Exploration of Failures in an sUAS Controller Software Product Line

Salil Purandare
Iowa State University
Ames, IA, USA
salil@iastate.edu

Myra B. Cohen
Iowa State University
Ames, IA, USA
mcohen@iastate.edu

ABSTRACT

Small uncrewed aerial systems (sUAS) are growing in their use for commercial, scientific, recreational, and emergency management purposes. A critical part of a successful flight is a correctly tuned controller which manages the physics of the vehicle. If improperly configured, it can lead to flight instability, deviation, or crashes. These types of misconfigurations are often within the valid ranges specified in the documentation; hence, they are hard to identify. Recent research has used fuzzing or explored only a small part of the parameter space, providing little understanding of the configuration landscape itself. In this work we leverage software product line engineering to model a subset of the parameter space of a widely used flight control software, using it to guide a systematic exploration of the controller space. Via simulation, we test over 20,000 configurations from a feature model with 50 features and 8.88×10^{34} products, covering all single parameter value changes and all pairs of changes from their default values. Our results show that only a small number of single configuration changes fail (15%), however almost 40% fail when we evaluate changes to two-parameters at a time. We explore the interactions between parameters in more detail, finding what appear to be many dependencies and interactions between parameters which are not well documented. We then explore a smaller, exhaustive product line model, with eight of the most important features (and 6,561 configurations) and uncover a complex set of interactions; over 48% of all configurations fail.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Software testing and debugging**; **Feature interaction**; • **Computer systems organization** → **Embedded and cyber-physical systems**.

KEYWORDS

Software Product Lines, Configurability, sUAS

ACM Reference Format:
Salil Purandare and Myra B. Cohen. 2024. Exploration of Failures in an sUAS Controller Software Product Line. In *28th ACM International Systems and Software Product Line Conference (SPLC '24), September 02–06, 2024, Dommeldange, Luxembourg*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3646548.3672597

SPLC '24, September 02–06, 2024, Dommeldange, Luxembourg

ACM ISBN 979-8-4007-0593-9/24/09
https://doi.org/10.1145/3646548.3672597

1 INTRODUCTION

Small uncrewed aerial systems (sUAS), or drones, are already widely used in a variety of real-world scenarios, from surveillance to package delivery. They can play key roles in safety-critical scenarios, such as emergency search and supply delivery in areas where natural disasters have occurred. Their usage is expected to only increase in the future. For instance, the drone delivery company Wing performed over 1,000 deliveries of food and groceries per day in Australia in 2023, while another similar service, Zipline, revealed that it had made over 600,000 deliveries of medical supplies in Africa [33]. Major technology companies that deliver goods, like Amazon, are also introducing drone delivery services in regions like the UK and the US, promising one-hour delivery times [4, 30].

Regardless of their intended purpose, whether they are being used to deliver emergency aid or groceries, one common requirement for sUAS is safe and stable flight behavior. Stability refers to the ability of the drone to maintain its intended flight state without experiencing excessive oscillation or movement and is a key property of safe flight behavior. If a drone system fails to maintain stability, it is a danger to not only itself, but also to other vehicles in the airspace and to the rest of the environment. A crash can cause serious harm to people or property on the ground, which has led sUAS to be described as safety-critical [9].

As drone technology rises in popularity and usage, flight control systems are becoming increasingly important. A flight controller is a circuit board that is carried on the drone and is primarily responsible for monitoring sensor data and constantly making adjustments to ensure stable flight behavior. Flight control software must interface with the on-board controller as well as a ground control station, which allows the user to control the drone, either manually or programmatically [12].

Flight control software like PX4 [24] and ArduPilot [1] allow users to tune the values of control parameters to improve flight behavior. Each parameter can take one of a wide range of values, and may be associated with stability, sensors, battery, or other important aspects of the drone's behavior. There is a set of parameters responsible for *attitude* (defined by the vehicle's roll, pitch and yaw) and rate control, which is part of the PID (Proportional, Integral, Derivative) controller system used by the software [12, 34]. This controller continuously outputs the setpoint (target) and response (observed) curves for the current flight state, and the goal of tuning these parameters is to match the response curve to the setpoint curve as closely as possible. By tuning these parameters correctly, users can help ensure safe and stable flight behavior for the required mission. Documentation of the parameter space is limited, and valid value ranges provided in the documentation may contain suboptimal or even failure-inducing values. As a result,

the responsibility falls upon the drone operator to tune the sUAS correctly before flying. This requires an in-depth understanding of both the software and hardware of the system which many users do not have, and online forum posts indicate that failures caused by misconfiguration are a serious issue [25, 26, 28].

As highlighted by Kim et al. [19], these parameter range specification bugs can also be exploited by malicious actors to intentionally trigger failures or crashes in subtle, difficult-to-detect ways. This underscores the need for systematic exploration to improve the general understanding of the configuration space. sUAS operators can leverage this knowledge to tune their drones more safely and easily, as well as to investigate problematic behavior and identify unsafe configurations before deployment to prevent them from causing damage to themselves or their environment. In follow on work to Kim et al., we examined ways to reconfigure the controller and automatically adapt when instabilities are detected [27]; however, we used simple approaches for reconfiguration that were not cognizant of the full configuration space.

Given that controllers are complex, configurable software systems, it is not a straightforward task to determine the role of every feature in the configuration space. Modifying parameter values can lead to unexpected consequences that are difficult to trace back to their source, and the vast number of possible configurations makes the space impossible to cover in its entirety. At any point in time, any number of features could be interfacing with one another in unexpected ways, leading to unpredictable behavior. Furthermore, documentation and general knowledge about interactions between multiple parameters is more limited than information about individual parameters. For these reasons, a systematic exploration of this space is necessary and highly relevant to sUAS operators.

The use of software product line engineering has the potential to help with both the exploration of the space of configurations (for testing, debugging, and evolution), as well as in documenting dependencies and constraints for the operators.

In this paper we use a product line engineering approach and model the configuration space for one popular controller, PX4 [24], as a software product line. We use an iterative approach and begin by exploring a configuration space consisting of 50 parameters and 8.8×10^{34} configurations. We use a one-hop exploration (all single parameter changes from the default values) to understand which parameter values fail on their own and then examine interactions using a two-hop exploration. We find that only 15% of the one-hops fail, while nearly 40% of the two-hop explorations fail. We also uncover interesting interactions that lead to passing tests in the two-hop data for tests which failed with a single configuration change, demonstrating the complexity of this space. We then explore an exhaustive region of eight controller parameters to understand the overall landscape. Despite removing parameter values known to cause failures in one-hop testing from the exhaustive feature model, we still find that 48.7% (almost half) of the configurations fail.

The overall contributions of this paper are:

(1) A feature model (and artifacts) for a part of the parameter space of PX4, a widely used flight control system for sUAS;
(2) Experiments on more than 30,000 simulations to explore the impact of 50 parameters related to attitude and rate control for drones on flight success and failure; and
(3) An investigation of positive and negative interactions between features (or parameters) in our product line.

The rest of this paper is laid out as follows. In the next section we present a motivating example which argues for the need for this exploration. In Sections 3 and 4 we present our study and results. We present related work (Section 5) and end with conclusions and future work in Section 6.

2 MOTIVATING EXAMPLE

To illustrate the impact of configuration changes on flight behavior and the importance of understanding them from the user perspective, we consider a small product line with five features: *MC_PITCHRATE_K, MC_PITCHRATE_MAX, MC_PITCHRATE_P*, *MC_ROLLRATE_P*, and *MC_ROLL_P* as shown in Figure 1. Each of these parameters control specific aspects of the drone's behavior. The documentation specifies that two of them, *MC_PITCHRATE_K* and *MC_PITCHRATE_P*, are related. They are used together by the controller to calculate pitch rate gain. The documentation does not describe any other relationships between any of the five parameters. However, this does not guarantee that no such relationships exist.

Consider the configuration in which *MC_PITCHRATE_K*, *MC_PITCHRATE_MAX*, *MC_PITCHRATE_P*, *MC_ROLLRATE_P* and *MC_ROLL_P* are set to 3.0, 110.0, 0.03, 0.08 and 0.1, respectively. During our exhaustive exploration using simulation (see RQ3) we saw some interesting (and unexpected) behavior. We highlight this in Figure 2, which shows screenshots of the flight using two different configurations (discussed next).

First, each of these values falls within the corresponding valid range specified in the documentation. Second, we observed that every individual parameter value in this configuration results in a successful mission if we change it while holding the other values at their defaults.

Combining *MC_PITCHRATE_P* = 0.03, *MC_ROLLRATE_P* = 0.08, and *MC_ROLL_P* = 0.1, which are all individually safe, however, results in extremely unstable behavior with the drone crashing soon after takeoff (left picture in Figure 2). There is nothing to connect these three parameters in the documentation, so this could cause problems for a user attempting pre-flight parameter tuning without this specific knowledge.

Interestingly, there are additional ways (higher order interactions) that affect the behavior of the drone under the influence of these parameters. Bringing the other two parameters, *MC_PITCHRATE_K* and *MC_PITCHRATE_MAX*, into the picture, we can mitigate the unsafe behavior caused by the interaction of the other three parameters to a noticeable degree. We found that setting the values of *MC_PITCHRATE_K* to 3.0 and *MC_PITCHRATE_MAX* to 110.0, which are also within their valid ranges, we can prevent the crashing behavior caused by the dangerous interaction. We show this flight on the right side of Figure 2. While the drone deviates to some degree from its expected flight path, the mission was able to complete and the drone landed safely.

Although the flight is still unstable, this mitigation may enable mission success in many cases, and understanding interactions like this may allow drone operators to tune their systems more easily and reliably. However, it's difficult for most operators to discover the effects of configurations like this organically in a safe manner

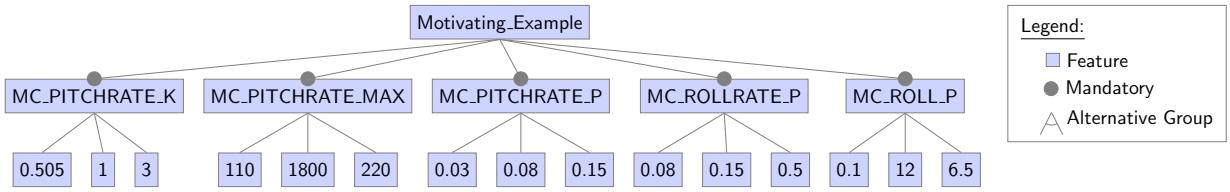

Figure 1: Software Product Line for our motivating example. This product line has five controller parameters as features, each of which can take one of three values.

Figure 2: Left: crashing behavior caused by a configuration consisting of MC_PITCHRATE_P = 0.03, MC_ROLLRATE_P = 0.08 and MC_ROLL_P = 0.1. Right: a flight using the same configuration as on the left but with the addition of two parameter values: MC_PITCHRATE_K = 3.0 and MC_PITCHRATE_MAX=110.0.

without endangering their hardware. Therefore, there is a need for more formal modeling and exploration of the configuration space to allow safer drone operation in the future. In this work, we tackle this problem and try to better understand the controller configuration space using a systematic product line engineering approach.

3 EXPERIMENTS

We aim to model and explore a large part of the controller configuration space systematically. We do this by asking three research questions. We provide artifacts on an external artifact website.[1]

- **RQ1:** Which features in our product line cause problematic flight behavior? To answer this question we compare our results with the existing (state of the art) research on sUAS controllers.
- **RQ2:** How do interactions between features affect flight success? For this question we examine both the positive and the negative changes in results between our one- and two-hop data sets.
- **RQ3:** How complex are the interactions in an exhaustive data set? For this experiment we try to understand the complexity of interactions.

In order to explore the parameter space of PX4, we first build a software product line that is broad and perform a series of trials similar to the one-dimensional mutation in Kim et al. [19] and the one-hop testing from our prior work [27]. We use the term one-hop in this work. The objective of these experiments is to determine which parameter values individually induce mission failures outside of the influence of other parameters. For each trial, the value of only one parameter is modified to a non-default value within its valid range as specified by the feature model. All other parameters are set to their default values. This allows us to determine the effect of only the current parameter on the outcome of the mission.

We follow this with two-hop interaction testing. Previous research has attempted a limited exploration of two-way interactions with small sets of parameters [19] or has used pairwise interaction testing [8, 11, 27]. The scope of these explorations was likely limited by the exponentially increasing number of trials required to test interactions between multiple parameters in the system. Moreover, Purandare et al. [27] found that most of the configurations that used pairwise interaction testing failed, likely due to the large distance from any default value.

In order to gain a more in-depth understanding, we perform a complete systematic set of two-hop trials that allows us to isolate and analyze the impact of pairs of parameter values on flight behavior. In these trials, we modify two parameters at a time, while all other parameters are kept at their default values. For every two-parameter pair from the set of parameters in the product line, we perform a run with every possible combination of the non-default values of the two parameters. The goal is to analyze the effects of

[1] https://github.com/LavaOps/splc2024

interactions between parameters which may not manifest when only one parameter is modified at a time.

3.1 Modeling the Product Line

We include all of the existing PX4 parameters used by Kim et al. [19], Han et al. [13], and Purandare et al. [27] in our product line. Two parameters used in [19] have since been deprecated in modern versions of PX4 and are thus no longer usable. We also incorporate new attitude and rate control-related parameters based on their similarity to the existing parameters in the documentation. This results in a total of 50 parameters in our study, which are listed in Table 1. While we use the same parameters, we use different partitions (parameter values). We choose four values for each parameter: the minimum allowed value, one value exactly halfway between the minimum and the default values, the maximum allowed value, and one value halfway between the default value and the maximum. In cases where the maximum value is not specified, we use the maximum value of closely related parameters with similar ranges, as in our prior work [27].

For each parameter in Table 1 we show the name, valid range, and default value. We don't show the other 4 values but have provided the full feature model on our artifact website. This product line has 5^{50} or 8.8×10^{34} configurations (or products). There are no constraints provided in the documentation, hence we assume that all products are potentially valid. The one-hop sample has 200 configurations - one for each non-default value for each parameter (4×50). The two-hop sample tests all pairs of non-default values and has $\binom{50}{2} \times 4 \times 4$, or 19,600 configurations.

3.2 Study Method

Different studies have used different approaches to determining flight success and failure for a given configuration. Examples include deviation from the expected flight state over a given threshold [19] or machine learning predictions [13, 14]. We apply the method described in our prior work [27], in which flight success is based on the outcome of the assigned mission. If the drone is able to reach every waypoint in the mission successfully within the allotted time with a given parameter configuration, that configuration is considered successful. If the configuration prevents the drone from completing the mission, it is classified as failure-inducing. We also keep track of whether or not the takeoff is successful in the case of failure. We want to differentiate between cases where the sUAS fails to fly (often due to the vehicle flipping over) and when it fails during flight. For our flight path, we simulate the same eight-waypoint mission plan as used in our prior work [27].

Timing issues caused by external factors in the computing system (e.g. heavy server loads causing processes to get suspended) can occasionally lead to non-configuration-related failures. To avoid incorrectly labeling tests resulting from individual instances of unexpected behavior as configuration failures, we verified failing configurations by repeating multiple trials for every failing configuration. This might lead to some over approximation of success in the case of a flaky pass, however based on our experience, this is not common and we re-ran all failures. For the one-hop data, we ran each failure 4 additional times and for the two-hop data we ran each failure 2 more times. Any configuration that failed the majority of the time in the verification runs was classified as a failure.

Each individual experiment takes 4-8 minutes of computational time to run on our server (depending on the outcome), hence the base data (without re-runs) represents from 55 to 110 days of actual computing time just for the simulations. We used 50 nodes in parallel to run our experiments. We note that the actual time to run these experiments was significantly longer due to the use of a shared computing platform.

Table 1: Complete list of all 50 parameters in our study with valid ranges and default values. "?" indicates that no limit has been specified in the PX4 documentation. The full set of parameter-values can be found on our artifact website.

Parameter	Range	Default
COM_ARM_IMU_ACC	[0.1, 1.0]	0.7
COM_ARM_IMU_GYR	[0.02, 0.3]	0.25
COM_POS_FS_EPH	[-1, 400]	1
COM_VEL_FS_EVH	[0, ?]	1
EKF2_ABL_LIM	[0.0, 0.8]	0.4
MC_PITCH_P	[0.0, 12]	6.5
MC_PITCHRATE_D	[0.0, ?]	0.003
MC_PITCHRATE_FF	[0.0, ?]	0
MC_PITCHRATE_I	[0.0, ?]	0.2
MC_PITCHRATE_K	[0.01, 5.0]	1
MC_PITCHRATE_MAX	[0.0, 1800.0]	220
MC_PITCHRATE_P	[0.01, 0.6]	0.15
MC_PR_INT_LIM	[0.0, ?]	0.3
MC_ROLL_P	[0.0, 12]	6.5
MC_ROLLRATE_D	[0.0, 0.01]	0.003
MC_ROLLRATE_FF	[0.0, ?]	0
MC_ROLLRATE_I	[0.0, ?]	0.2
MC_ROLLRATE_K	[0.01, 5.0]	1
MC_ROLLRATE_MAX	[0.0, 1800.0]	220
MC_ROLLRATE_P	[0.01, 0.5]	0.15
MC_YAW_P	[0.0, 5]	2.8
MC_YAW_WEIGHT	[0.0, 1.0]	0.4
MC_YAWRATE_D	[0.0, ?]	0
MC_YAWRATE_FF	[0.0, ?]	0
MC_YAWRATE_I	[0.0, ?]	0.1
MC_YAWRATE_K	[0.0, 5.0]	1
MC_YAWRATE_P	[0.0, 0.6]	0.2
MIS_YAW_ERR	[0, 90]	12
MOT_SLEW_MAX	[0.0, ?]	0
MPC_ACC_HOR	[2, 15]	3
MPC_ACC_HOR_MAX	[2, 15]	5
MPC_ACC_UP_MAX	[2, 15]	4
MPC_JERK_AUTO	[1, 80]	4
MPC_JERK_MAX	[0.5, 500]	8
MPC_LAND_SPEED	[0.6, ?]	0.7
MPC_THR_MAX	[0, 1]	1
MPC_THR_MIN	[0.05, 0.5]	0.12
MPC_TILTMAX_AIR	[20, 89]	45
MPC_TILTMAX_LND	[10.0, 89.0]	12
MPC_TKO_SPEED	[1, 5]	1.5
MPC_XY_CRUISE	[3.0, 20.0]	5
MPC_XY_P	[0.0, 2.0]	0.95
MPC_XY_VEL_I_ACC	[0.0, 60.0]	0.4
MPC_XY_VEL_MAX	[0.0, 20.0]	12
MPC_XY_VEL_P_ACC	[1.2, 5.0]	1.8
MPC_Z_VEL_D_ACC	[0.1, 2.0]	0.2
MPC_Z_P	[0.0, 1.5]	1
MPC_Z_VEL_MAX_DN	[0.5, 4.0]	1
MPC_Z_VEL_MAX_UP	[0.5, 8.0]	3
MPC_Z_VEL_P_ACC	[2.0, 15.0]	4

3.3 Implementation Details

All experiments were performed using PX4. PX4-Autopilot [24] is a popular open source flight control software. It interfaces with the drone's onboard hardware controller and the ground control station (GCS) using the MAVLink messaging protocol. It is responsible for carrying out the mission plan and controls defined by the user, as well as sending flight status and sensor data back to the user through the GCS. We used the stable 1.12 version of PX4 and the Iris multicopter airframe. We performed the simulations using the widely used Gazebo simulator [20]. The test mission was defined using Python and the mission details and commands were sent to the sUAS using MavROS. ROS [32] is a popular open source software platform which provides tools and libraries for robotics applications. We used ROS Noetic for our implementation. The missions were defined within a Singularity [22] container with 2 CPUs and 2GB of memory. The missions were managed by a Python program using Python 3.9 on Ubuntu 20.04. All experiments were performed with 8 CPUs and 16GB of memory.

3.4 Threats to Validity

All tests in our study were performed using the Gazebo simulator. Although this is the most widely used simulator for sUAS, it cannot model reality perfectly and there are likely to be some differences in flight behavior in real-world systems [20]. This gap between simulation and reality may be narrowed by future tests with physical drone hardware, but due to the dangerous failure-inducing nature of these experiments, as well as the number of total flights required, the current best evidence of the relevance of these results are forum posts by users who encountered similar unstable behavior unintentionally [25, 26]. We incorporated parameters from these reports into our feature model.

The drone model used for all experiments was the Iris quadcopter airframe. This is the same airframe used in other related work using PX4 [13, 19, 27], which allowed us to compare the observed effects of parameter changes with those reported by other works fairly. However, ideal parameter value ranges differ between airframes, and these results are not necessarily generalizable to other airframes, including various real-world drone models. The mission used for all experiments was the same as the flight path described in our prior experiments [27], for fair comparison. This mission is a basic but nontrivial task which is intended to emulate a real-world drone mission. However, we acknowledge that it may pose varying levels of difficulty for different airframes, and other missions may lead to different outcomes for matching parameter configurations.

In addition, we note that the parameter space of PX4 is much larger than the subset we analyzed. We focused on impactful parameters related to attitude and rate control, which is an important aspect of sUAS stability and safety. We incorporated parameters tested by other research which are expected to have an impact on flight behavior, as well as related parameters from forum posts in order to ensure we were studying features that were relevant to drone operators for PID tuning. However, these are not the only parameters capable of affecting flight success, and there are likely other instability-inducing parameters in the PX4 software that remain to be studied. We cannot be sure that the results from our feature model generalize to other parts of the PX4 configuration-space.

Finally, as described in Section 3.2, unpredictable external factors related to the environment of the physical server that the simulations were performed on could have caused some incorrect failures. In order to limit the extent to which they skewed our results, we performed multiple trials of every initially failing test in our one- and two-hop testing to ensure that each failure was truly configuration-induced and not random. We also validated interesting interactions we observed during this study through additional repeated trials.

4 RESULTS

We now present the results of each of our research questions and summarize our findings.

4.1 RQ1: Individual Feature Failures

For this research question we performed one-hop testing, in which a single parameter was modified at a time while all other parameters in the product line were kept at their default value, with four values for each parameter. In these trials, we found that 30 configurations out of 200 that involved single-parameter changes were failure-inducing. These were spread across 24, or approximately half, of the parameters.

All runs with failure-inducing parameter values were repeated four times, for a total of five trials. If the majority of runs failed, we considered it a failure. A comparison of every failing value in our one-hop trials with the result for the same value in the CICADA [27] (blue star) and RVFuzzer [19] (red triangle) datasets, as well as takeoff success data, is shown in Figure 3. The graph shows each of the 30 failing configurations on the `y-axis`. The `x-axis` has four columns. The first (fail) shows which of the other techniques (CICADA and/or RVFuzzer) also failed on these parameters.

The n/a column indicates which technique did not test this parameter and the success column shows which techniques returned a successful result. There was one parameter from RVFuzzer that we were unable to determine the outcome for (this parameter was in their dataset but the presentation of the outcome was inconclusive), hence we put this in its own column.

The black triangle (Takeoff) is not a technique. Rather, a takeoff in the success column indicates that the sUAS was able to complete takeoff in our experiments, despite the overall mission failure. Parameters with a black triangle in the fail column mean the sUAS failed to even complete takeoff.

These results indicate that a relatively small number of the features are responsible for most, if not all, configuration-related failures in the system. Many parameters simply don't influence flight behavior in a noticeable way. There were, in contrast, some parameters that seemed especially relevant with respect to instability and failure-causing behavior. The parameter that failed for the largest range of its valid values was *MPC_THR_MAX*, which is responsible for limiting the drone's maximum thrust. This parameter triggered a failure for every non-default value. The default value of this parameter is also its maximum allowed value, suggesting that at least for the quadcopter airframe used in these experiments, the sUAS does not support any values of this parameter far below its maximum.

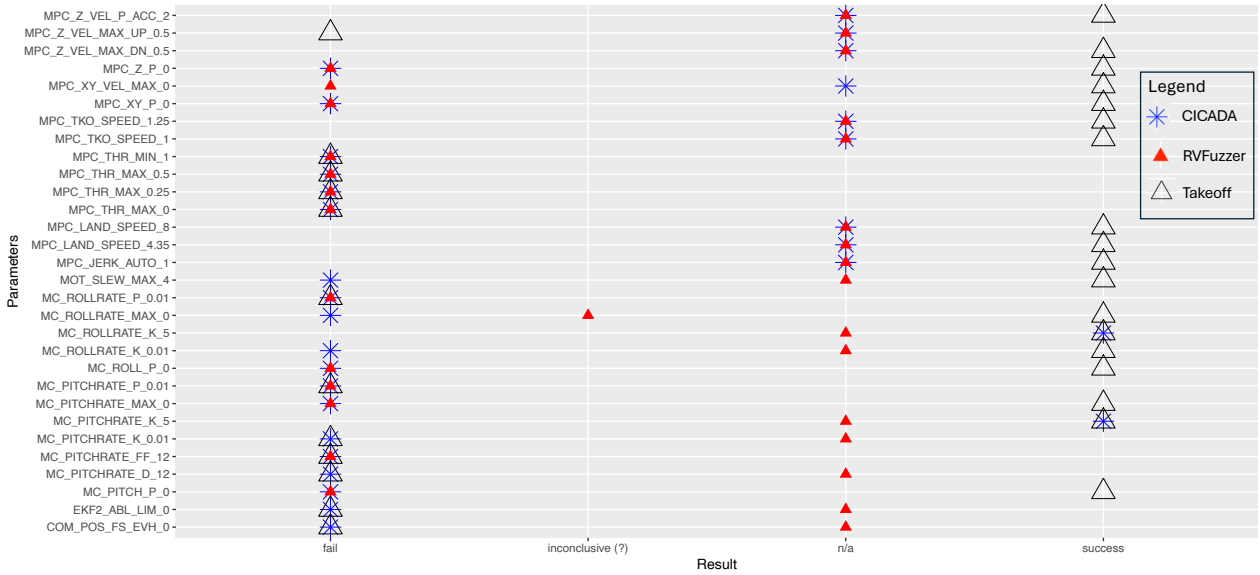

Figure 3: A comparison of the failure-inducing configurations in one-hop testing with the outcomes for the same value in CICADA [27] (blue star) and RVFuzzer [19] (red triangle). Failures indicate that the associated technique failed. N/A means the other technique did not test a particular configuration. Inconclusive indicates we were unable to determine pass or fail in their technique. The black triangle refers to the takeoff phase of our experiments. A failure indicates the sUAS never took off. A success indicates it took off and failed during flight.

Two other parameters which caused failures for multiple values were *MC_PITCHRATE_K* and *MC_ROLLRATE_K*, which control the pitch and roll rate controller gain respectively. However, for both of these parameters, the values that were failing were the minimum and the maximum value, while the values in between the two extremes allowed successful mission completion. This is especially interesting because it appears to contradict the *monotonic property of control instability* which has been reported in previous work in this area [19]. This property states that changing the value of a parameter will cause a monotonic change in stability, such that instability will increase or decrease in a single direction for any given parameter. The one-hop testing of parameters like *MC_PITCHRATE_K* and *MC_ROLLRATE_K* suggests that certain features in the space may have greater complexity and cannot be treated as monotonic.

The failure-inducing values tended to lie at extreme ends of the range of valid values as specified in the documentation. These results generally agree with the observations made by CICADA [27] and RVFuzzer [19]. However, there are some notable differences in the results. For example, although we found in our one-hop testing that both the minimum and maximum values for the *MC_PITCHRATE_K* and *MC_ROLLRATE_K* were failure-inducing, only the minimum value failed for each parameter in CICADA's one-hop tests. This may be due in part to differences in the machines used for simulation.

12 of the 30 failures involved failing takeoffs. Configurations that provoke takeoff failure errors are likely to crash and damage the hardware, while configurations that fail later in the mission often prevent the drone from being able to move to its intended target without instantly crashing. Although mid-flight failures are still serious, recent work has indicated that it may be possible to take advantage of the additional time spent in the air to detect anomalies and reconfigure the drone or change its flight plan to recover from the misconfiguration [16, 27].

Summary of RQ1. Only 24 of the 50 features (30 out of 200 values) in this product line led to a failure when combined with the rest of the feature's default values. When compared with state of the art techniques we saw similar patterns; however, we observed more nuanced behavior with respect to the monotonic properties previously described. We also saw some features which are more likely to fail than others.

4.2 RQ2: Interactions Between Features

While individual failure-causing parameter values are useful, one-hop testing fails to capture the impact of interactions between multiple parameters. In order to explore this aspect of our product line, we followed up with two-hop testing. For every value of each parameter in our model, we paired it with every value of all other parameters in the model and tested the resulting configuration. This resulted in 19,600 total two-hop configurations, which we then tested in the same way as the one-hop missions to determine failures and successes. Of these, we found 7,814 that were failure inducing. Similarly to the one-hops, we repeated trials for failure-causing configurations, in this case running them a total of three times to reduce the risk of random failures caused by external factors.

Table 2: Comparison of one-hop, two-hop, and exhaustive results in terms of mission outcome and takeoff success. The exhaustive set is based on a smaller feature model.

	# Configs	Failures	Failure Rate	Failing Takeoffs	Takeoff Failure Rate
One-hop	200	30	15.0%	12	6%
Two-hop	19,600	7,814	39.9%	2,576	13.1%
Exhaustive	6,561	3,194	48.7%	2,448	37.3%

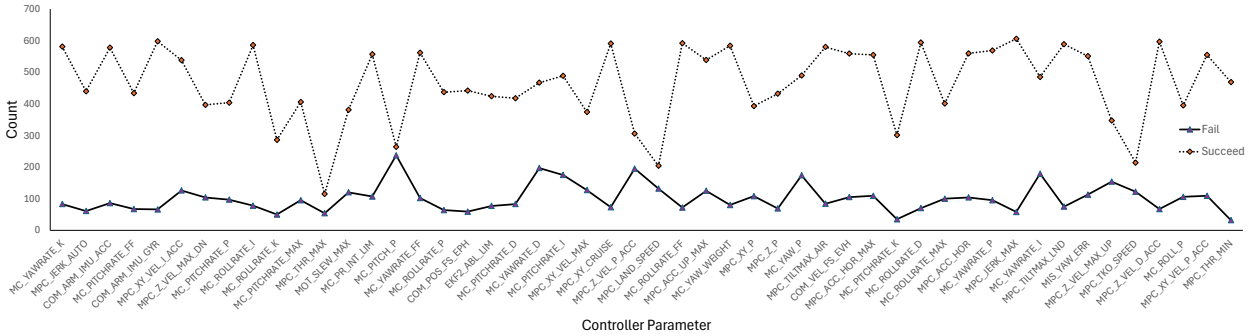

Figure 4: Negative interactions: two-hop failures and successes by parameter, considering only values that did not cause failures in one-hop testing. The top line shows those parameters which pass, the bottom line indicates failures.

We note that the failure rate is considerably higher in two-hop testing. The primary reason for this is interactions between parameters causing unstable flight behavior. Although some interactions and relationships are explicitly documented in PX4 [12], it is evident that many more such interactions exist. We observed 2,530 instances in which two parameter values which were both individually non-failure-inducing in one-hop testing triggered failures when set together. Therefore, out of the 7,814 total failures observed in two-hop testing, 32.4% were caused by negative interactions between otherwise stable parameters, suggesting that this is a prevalent issue in drone parameter tuning. We show this data in the first two rows of Table 2. This shows the number of configurations and failures, followed by the failure rate, number of failing takeoffs and the failing takeoff rate.

Interestingly, most parameter values that were not individually failure-inducing in one-hop testing ultimately led to a failure when paired with some other individually harmless (non-failure-inducing) parameter in the feature model. Figure 4 shows the failures and successes for all parameter values which did not fail individually in one-hop testing. The graph groups features by parameter (we exclude individual values to make this more readable). Failures in this graph indicate what we term a negative interaction; the test goes from passing to failing by the addition of a second parameter that does not fail when changed in the 1-hop experiments.

We found that *MC_PITCH_P* was the parameter with the highest number of failures in this set. *MC_PITCH_P*, which controls pitch proportional gain, is an attitude control parameter that is typically not instability-inducing except when set to extremely low values, but it appears to be prone to failure-causing interactions with other parameters. One reason for this may be that many of the parameters in our feature model are tied to the rate controller, which operates closer to the hardware than the attitude controller. This means the other controllers send commands through the rate controller and as a result, the tuning of rate control parameters could interact with attitude control parameters like *MC_PITCH_P*.

There were several parameters which experienced very few negative interactions, including *MPC_JERK_MAX*, *MPC_TILTMAX_LND*, *MPC_XY_CRUISE*, and *MPC_Z_VEL_D_ACC*. There are two likely explanations for this behavior. First, the low failure rate of these parameters in one-hop testing indicates that these are generally all very stable parameters. Furthermore, many of these parameters are only relevant at specific periods during the mission. For example, *MPC_TILTMAX_LND* is engaged exclusively during landing, and *MPC_Z_VEL_D_ACC* only impacts vertical movement, i.e., during takeoff and landing. For the rest of the flight, the other parameter in the pair will be the only factor in determining the success of the flight, resulting in minimal interaction for most of the mission.

Some other parameters seem resistant to negative interactions: *MC_ROLLRATE_D*, *MC_ROLLRATE_I*, and *MC_YAWRATE_FF*. This may be due in part to their status as rate control parameters, i.e. they are closest to the hardware and are not sending commands through another controller. Additionally, these are generally stable parameters across their value spectrum, as demonstrated in one-hop testing, which increases their representation in the graph. We repeated this analysis for a normalized set which included the same number of values for every feature, and these parameters still had some of the lowest failure rates.

Another direction of interest is *positive* parameter interactions, in which the influence of a beneficial parameter was able to mitigate the failure-causing impacts of another parameter. We did not expect many cases, since previous work in this area has not described this phenomenon [19, 27]. During our two-hop testing, we identified 168

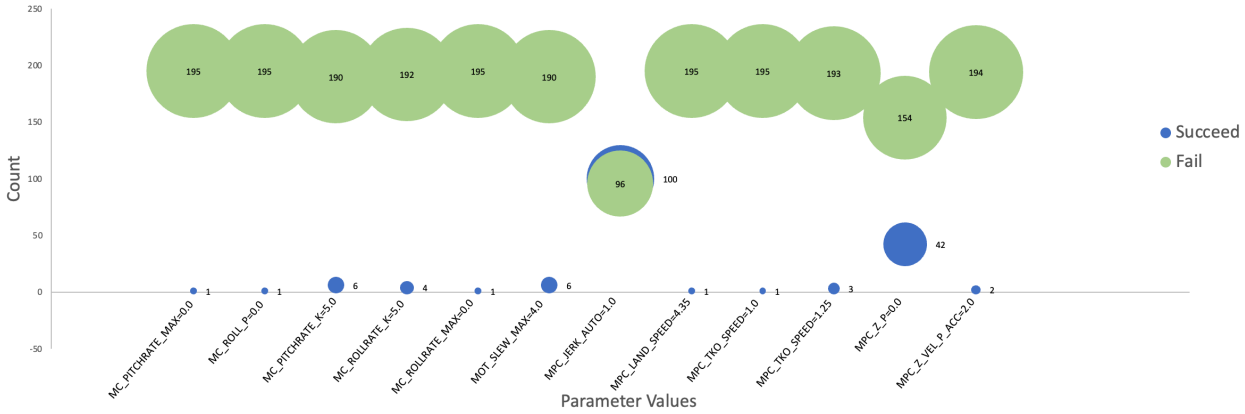

Figure 5: Positive interactions. Parameter values on the x-axis failed in one-hop testing, but succeeded when paired with some other parameter value in two-hop testing. Failing bubbles indicate no change in two-hop testing. Success bubbles indicate the mission succeeded when an additional parameter was changed. Sizes of bubbles indicate the number of successes or failures.

instances of configurations in which at least one parameter value triggered failure individually during one-hop testing, but ultimately succeeded when paired with another parameter value in the product line. These instances involved 12 distinct parameter values. Many of these positive interactions were between two parameters that had no apparent relationship based on the documentation of the source code [12]. All positive interactions discovered in two-hop testing are summarized in Figure 5. In this graph each of the 12 distinct parameters and their values are shown on the `x-axis`. We show two bubbles for each parameter. The `y-axis` indicates the number of times a succeed or fail occurs and the size of the bubble provides a visual clue to the size as well. In most cases the two bubbles are distinct. For one parameter (*MPC_JERK_AUTO* = 1.0) we see an even split of successes and failures.

We believe this knowledge of positive interactions can have meaningful applications for sUAS operators. For example, some drone adaptation frameworks [2, 27] attempt real-time configuration-based mitigations to unstable flight behavior. One of these, CICADA, applies adaptation techniques to some of the parameters included in our study. However, for some parameter values, for example, *MPC_Z_P* = 0, the adaptation attempts were unsuccessful. Our two-hop exploration identified multiple parameters which have positive interactions with *MPC_Z_P*, and setting one of these beneficial parameters instead of trying to modify the value of the failure-causing parameter itself could be an alternative adaptation strategy which may be preferable in some scenarios.

Summary of RQ2. We observed a large number of interactions and more failures when we explored the two-hop data. We observed both negative and positive interactions. All parameters with individually succeeding one-hop values did ultimately fail in combination with some other succeeding parameter value, while 12 parameters that failed during the one-hop exploration passed when paired with another parameter.

4.3 RQ3: Exhaustive Exploration

As a final experiment, we created a smaller product line and exhaustively tested it. Since any attempt at exhaustive coverage of the entire space is impossible, as it would require a number of trials in the order of 10^{50}, we chose to limit our focus to a subset of the feature model which is most relevant to drone operators and which we believed could yield useful complex interactions. We selected eight parameters which were all related to pitch and roll behavior: *MC_PITCHRATE_P, MC_PITCH_P, MC_ROLLRATE_P, MC_ROLL_P, MC_PITCHRATE_D, MC_PITCHRATE_K, MC_ROLLRATE_D*, and *MC_PITCHRATE_MAX*, based on their prevalence in forum reports [25, 26, 28], as well as their behavior in individual and interaction tests. In order to further optimize the number of runs that this would entail, we limited the number of values each parameter could take to three instead of five as in the previous runs. Importantly, since our goal was to maximize our chances of discovering non-trivial interactions, we decided against allowing parameter values which were known to be individually failure-inducing based on the one-hop tests. However, we did include values that were close to failure-inducing values in order to facilitate interesting flight behavior. Our feature model for the exhaustive trials is included in Figure 6. Most of the non-leaf features are collapsed for readability. (the full feature model is on our supplemental website). It consists of 3^8 or 6,561 total configurations. While still small, it is larger than previous exhaustive analyses.

Figure 7 summarizes the interactions between this set of parameters. This is a tree created by the R rpart package [35]. It uses recursive partitioning. We also ran this data through the Weka J48 algorithm [29] and the two trees agree at the top level, however the J48 tree is more complex, hence we chose to show this version for understanding. It suggests that *MC_PITCHRATE_P*, *MC_PITCHRATE_K*, and *MC_PITCHRATE_D* are the most influential features in this set. Interestingly, *MC_PITCHRATE_P*, which controls the pitch rate proportional gain, appears to have the greatest impact on flight outcome. This is surprising because the formula for the global gain of the controller is scaled

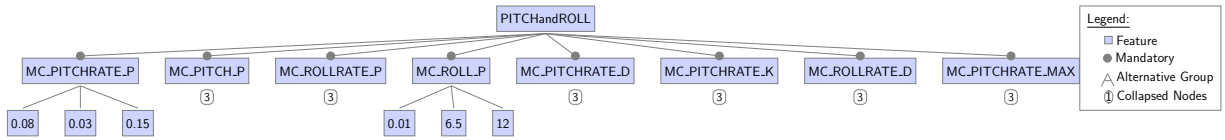

Figure 6: Feature model for pitch and roll consisting of 6,561 products

by *MC_PITCHRATE_K*, so it might be expected to have an effect as large as the *MC_PITCHRATE_P*, *MC_PITCHRATE_I*, and *MC_PITCHRATE_D* parameters combined. However, we find that this is not the case, and that *MC_PITCHRATE_P* has a more meaningful impact on flight success. This may be due to the specific values in our feature model for the exhaustive set. Although all of the values were known to be individually non-failure-inducing, the range selected for *MC_PITCHRATE_P* might have included more provocative values than the range for *MC_PITCHRATE_K*. It is worth noting that the most influential parameters are all operating at the rate controller level. This is likely explained by the fact that the rate controller is the closest controller in the system to the hardware, and has potential to interact with most other controllers [12]. Therefore, they may have a strong effect on attitude control parameter and ultimately on flight behavior.

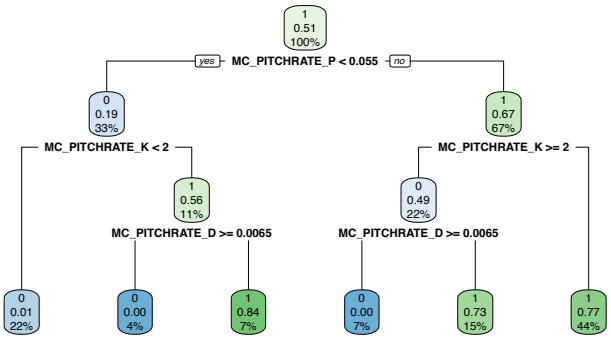

Figure 7: Effects of interactions between parameters in the exhaustive set, where 1 represents success and 0 represents failure. Parameters higher in the tree have a greater impact on flight success.

Ultimately, we found that the pattern of increasing interactions leading to increasing numbers of failures continued. Nearly half of all runs in the exhaustive trials failed (48%) as shown in the last row of Table 2, which is even more striking given that all individually failure-causing values were removed from this model. Had all such values been included, the number of failures would have been even higher (we have observed similar behavior in smaller exhaustive sets which retain those values).

Interestingly, the rise in the percentage of failing takeoffs in the exhaustive tests compared to the one- and two- hops was higher than the rise in the overall failure rate. This may indicate that more complex interactions, those with more parameters involved, are more likely to lead to severe and immediate flight failures which prevent the drone from even being able to reach flying altitude.

Summary of RQ3. We found that the number of failures was higher (48%) in the exhaustive data set despite having removed known failing one-hop values, and the complexity of interactions was also apparent. We could not find any parameters which always failed or failed with a simple 2-way interaction, leading to complicated trees describing the behavior.

4.4 Discussion

Throughout this exploration we have observed several interesting phenomenon that we believe can benefit from a tighter integration with product line engineering. First, the use of a feature model allows us to perform systematic exploration and has identified potential dependencies (or interactions) among parameters (and parameter values) in this work, such as those between *MC_PITCHRATE_P*, *MC_ROLLRATE_P*, and *MC_ROLL_P*. We believe we can then utilize the learned information and add constraints and attributes back to the feature model for improved documentation. Complex interactions are important for drone operators to understand prior to parameter tuning. They can have unexpected and sometimes non-intuitive effects on flight behavior, including both negative and positive interactions, as highlighted in Section 2. Additionally, they may also have applications in adaptation and failure prevention.

Additionally, research on drone adaptation [16, 27] has highlighted the importance of speed in adapting to failure-inducing parameters. Faster response times can make the difference between a successful recovery and a disastrous outcome. Therefore, in situations where a complex interaction between multiple parameters is causing dangerous flight behavior, it may be advantageous to attempt a lightweight adaptation with one or two positively interacting parameters instead of correcting all of the negatively interacting parameters.

Second, documentation can be helpful to both developers as well as operators who need to configure before takeoff. A tighter integration between the human and the feature model would be beneficial. For instance, if we provided the operator with a prompt filling in the missing documentation (as they selected one parameter and configured it), this could prevent potential crashes.

Last, we believe we can leverage the feature model for more sophisticated testing, such as by guiding prioritization and selection based on the importance of various slices of the model. We plan to explore some of these ideas in future work.

5 RELATED WORK

There has been work on exploring controller parameters for sUAS in the past few years. Kim et al. presented RVFuzzer [19], a system for discovering range implementation and specification bugs in robotic vehicles. RVFuzzer uses a binary search to refine valid

parameter value ranges beyond those defined in documentation based on a metric of flight state deviation. However, it considers a limited section of the relevant configuration space and focuses on preventing malicious parameter changes by attackers rather than investigating parameter behavior in-depth for the benefit of operators. Purandare et al. proposed CICADA [27], an adaptation framework for sUAS which investigates the impact of parameter changes on flight stability and introduces adaptation mechanisms to counteract unstable flight behavior. Although this work investigates the impact of parameter changes and interactions between parameters to some extent, it does not focus primarily on the behavior of specific parameters and uses sampling techniques to cover the configuration space which lose interaction-related information. We have incorporated all existing PX4 parameters from both RVFuzzer and CICADA into our feature model.

In recent work, Cleland-Huang et al. [7] performed an exhaustive exploration of a small subset of PX4's configuration space consisting of four flight control parameters and built classification trees to show the potential interactions. However, their product line consisted of only 625 products and they did not perform an in-depth analysis of the interactions. This subset of parameters is also included in our feature model.

Han et al. introduced LGDFuzzer [14] and ICSEARCHER [13], which use genetic algorithms to search for problematic configurations in sUAS. They leverage a machine learning-based instability predictor and multi-objective optimization to search for incorrect parameter value ranges. However, these works do not systematically explore the parameter space or focus on the impact of specific parameters in the system. Furthermore, their dependence on the instability prediction model means that many complex and unpredictable interactions in the vast configuration space may be overlooked. Similarly, Chang et al. [5] use a genetic algorithm-based approach called APFuzzer to search for configurations that send drones into incorrect states. APFuzzer is simulation-based, but it is designed to find a diverse set of bad configurations in an efficient manner, rather than investigating the behavior of any features or interactions between them in depth. It does not systematically explore a large number of configurations, and uses the Ardupilot flight control software [1] rather than PX4 in its implementation, which has an entirely different set of parameters.

Cleland-Huang et al. [6] proposed a requirements-based approach for creating a software product line of configurable scenarios for sUAS, and Islam et al. [15] present a software product line configuring drone missions. These, however, are concerned with mission plans as the features in their system, and do not deal with flight control parameters. Khatiri et al. presented SURREALIST [17] and AERIALIST [18], simulation-based test case generation tools for sUAS. Given a flight log as input, these tools are able to replicate the flight in a simulator and generate slightly modified test cases. The modifications made are primarily to the external environment, especially by introducing obstacles in the flight path. These works do not evaluate flight control parameters.

Last, there has been research on software product line engineering in cyberphysical systems [10, 21] and the robotics domain [3, 23, 31, 36], however, none of this work focuses on controllers. Rather, they model other aspects of their respective systems.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we modeled part of the configuration space of a flight control software as a product line and systematically explored the effects of the features on sUAS flight behavior. We studied a product line consisting of 50 flight control parameters in the widely used PX4 software leading to 8.8×10^{34} configurations. We first investigated individual features by performing a series of one-hop tests, consisting of 200 configurations. Using a simulated test mission, we identified a set of 30 parameter values that induced mission failure by preventing the drone from flying as intended, and compared our results to those presented in recent related works.

We continued our exploration with two-hop testing, in order to determine the effects of interactions between features. We tested 19,600 configurations with this approach and discovered 7,814 failing configurations. We further found 2,530 configurations that induced failure specifically due to negative interactions between parameters, and found 168 instances of positive interactions in which one parameter mitigated the failure-causing effects of another parameter. The majority of these interactions were not captured by the existing documentation of the system.

Finally, we performed an exhaustive exploration of a subset of our feature model consisting of eight parameters, with three values each. This test set consisted of 6,561 configurations, of which 3,194, or nearly half, were discovered to be failure-inducing. We allowed only values that did not fail in one-hop testing, which meant that every failure we encountered was due to a negative interaction between otherwise harmless parameter values. We also discovered that in some special cases, positively interacting parameters were able to mitigate failure-inducing behavior caused by negative interactions between other parameters.

We believe that this exploration of the sUAS control parameter product line can meaningfully benefit drone operators. Misconfigurations of flight control parameters are a commonly reported issue, and enabling users to gain a better understanding of the system can help prevent the occurrence of dangerous behaviors like instability and crashing. By modeling this space as a product line, we can discover not only the behavior of individual parameters but also interactions between multiple parameters, both positive and negative, which may have applications for research in sUAS stability and configuration-based adaptation.

In future work, we plan to expand our exhaustive exploration to cover more parameters. We also want to incorporate more complex tests into our experiments, including multiple missions and airframes, and to measure the impact of complexity of interactions in greater depth. We further plan to design methods for tighter integration of software product line engineering and configuring the sUAS controller, using the feature model for advanced test generation, and incorporate learned constraints into our model. We also plan to explore the use of the feature model to help in documentation for operator parameter tuning.

ACKNOWLEDGMENTS

This work was funded in part by the National Aeronautics and Space Administration Grant 80NSSC21M0185 and the National Science Foundation Grants CCF-1909688, CNS-2234908. We thank the anonymous reviewers for their useful comments.

REFERENCES

[1] Ardupilot. Last Accessed 04/22/24. ArduPilot Open Source Autopilot. https://ardupilot.org/

[2] Victor Braberman, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel. 2015. MORPH: A reference architecture for configuration and behaviour self-adaptation. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*. 9–16.

[3] Davide Brugali. 2021. *Software Product Line Engineering for Robotics*. Springer International Publishing, Cham, 1–28. https://doi.org/10.1007/978-3-030-66494-7_1

[4] Zach Buchanan. 2024. Coming soon to the West Valley: alarmingly large Amazon drones. *Phoenix New Times* (April 2024). https://www.phoenixnewtimes.com/news/amazon-drones-phoenix-huge-18837322

[5] Zhiwei Chang, Hanfeng Zhang, Yue Yang, Yan Jia, Sihan Xu, Tong Li, and Zheli Liu. 2024. Fuzzing Drone Control System Configurations Based on Quality-Diversity Enhanced Genetic Algorithm. In *Artificial Intelligence Security and Privacy*, Jaideep Vaidya, Moncef Gabbouj, and Jin Li (Eds.). Springer Nature Singapore, Singapore, 499–512.

[6] Jane Cleland-Huang, Ankit Agrawal, Md Nafee Al Islam, Eric Tsai, Maxime Van Speybroeck, and Michael Vierhauser. 2020. Requirements-driven configuration of emergency response missions with small aerial vehicles. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A (SPLC '20)*. Article 26, 12 pages. https://doi.org/10.1145/3382025.3414950

[7] Jane Cleland-Huang, Nitesh Chawla, Myra B. Cohen, Md Nafee Al Islam, Urjoshi Sinha, Lilly Spirkovska, Yihong Ma, Salil Purandare, and Muhammed Tawfiq Chowdhury. 2022. *Towards Real-Time Safety Analysis of Small Unmanned Aerial Systems in the National Airspace*. https://doi.org/10.2514/6.2022-3540

[8] Myra B. Cohen, Peter.B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. 2003. Constructing test suites for interaction testing. In *International Conference on Software Engineering, 2003. Proceedings.* 38–48. https://doi.org/10.1109/ICSE.2003.1201186

[9] Andrea Di Sorbo, Fiorella Zampetti, Aaron Visaggio, Massimiliano Di Penta, and Sebastiano Panichella. 2023. Automated Identification and Qualitative Characterization of Safety Concerns Reported in UAV Software Platforms. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 67 (apr 2023), 37 pages. https://doi.org/10.1145/3564821

[10] Hafiyyan Sayyid Fadhlillah, Antonio M. Gutiérrez Fernández, Rick Rabiser, and Alois Zoitl. 2023. Managing Cyber-Physical Production Systems Variability using V4rdiac: Industrial Experiences. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A* (Tokyo, Japan) *(SPLC '23)*. Association for Computing Machinery, New York, NY, USA, 223–233. https://doi.org/10.1145/3579027.3608994

[11] Michael Forbes, James Lawrence, yu Lei, Raghu Kacker, and D. Kuhn. 2008. Refining the In-Parameter-Order Strategy for Constructing Covering Arrays. *Journal of Research of the National Institute of Standards and Technology* 113 (09 2008). https://doi.org/10.6028/jres.113.022

[12] PX4 Development forum. Last Accessed 04/24/24. PX4 Documentation. https://docs.px4.io/master/en/

[13] Ruidong Han, Siqi Ma, Juanru Li, Surya Nepal, David Lo, Zhuo Ma, and Jianfeng Ma. 2024. Range Specification Bug Detection in Flight Control System Through Fuzzing. *IEEE Transactions on Software Engineering* 50 (March 2024), 1–13. https://doi.org/10.1109/TSE.2024.3354739

[14] Ruidong Han, Chao Yang, Siqi Ma, JiangFeng Ma, Cong Sun, Juanru Li, and Elisa Bertino. 2022. Control parameters considered harmful: detecting range specification bugs in drone configuration modules via learning-guided search. In *Proceedings of the 44th International Conference on Software Engineering*. 462–473. https://doi.org/10.1145/3510003.3510084

[15] Md Nafee Al Islam, Muhammed Tawfiq Chowdhury, Ankit Agrawal, Michael Murphy, Raj Mehta, Daria Kudriavtseva, Jane Cleland-Huang, Michael Vierhauser, and Marsha Chechik. 2023. Configuring mission-specific behavior in a product line of collaborating Small Unmanned Aerial Systems. *Journal of Systems and Software* 197 (2023), 111543. https://doi.org/10.1016/j.jss.2022.111543

[16] Md Nafee Al Islam, Michael Vierhauser, and Jane Cleland-Huang. 2024. ADAM: Adaptive Monitoring of Runtime Anomalies in Small Uncrewed Aerial Systems. In *2024 IEEE/ACM 19th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 44–55. https://doi.org/10.1145/3643915.3644092

[17] Sajad Khatiri, Sebastiano Panichella, and Paolo Tonella. 2023. Simulation-based Test Case Generation for Unmanned Aerial Vehicles in the Neighborhood of Real Flights. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 281–292. https://doi.org/10.1109/ICST57152.2023.00034

[18] Sajad Khatiri, Sebastiano Panichella, and Paolo Tonella. 2024. Simulation-based testing of unmanned aerial vehicles with Aerialist. In *Proceedings of the International Conference on Software Engineering: Companion Proceedings*. 134–138. https://doi.org/10.21256/zhaw-29678

[19] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFuzzer: Finding Input Validation Bugs in Robotic Vehicles through Control-Guided Testing. In *Proceedings of the 28th USENIX Conference on Security Symposium*. 425–442.

[20] N. Koenig and A. Howard. 2004. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vol. 3. 2149–2154 vol.3. https://doi.org/10.1109/IROS.2004.1389727

[21] Jacob Krüger, Sebastian Nielebock, Sebastian Krieter, Christian Diedrich, Thomas Leich, Gunter Saake, Sebastian Zug, and Frank Ortmeier. 2017. Beyond Software Product Lines: Variability Modeling in Cyber-Physical Systems. In *Proceedings of the International Systems and Software Product Line Conference - Volume A*. 237–241. https://doi.org/10.1145/3106195.3106217

[22] Gregory M. Kurtzer, Dave Trudgian, cclerget, Michael Bauer, Ian Kaneshiro, David Godlove, Vanessasaurus, Yannick Cote, Carlos Eduardo Arango Gutierrez, Geoffroy Vallee, DrDaveD, Justin Cook, Jason Stover, Adam Hughes, Brian P Bockelman, Marcelo Magallon, Jacob Chappell, Mike Frisch, Daniele Tamino, Carl Madison, Sasha Yakovtseva, Satrajit Ghosh, Amanda Duffy, VP, Tru Huynh, Mike Gray, Yaroslav Halchenko, and Felix Abecassis. 2024. *sylabs/singularity: Singularity*. https://doi.org/10.5281/zenodo.10782908

[23] Niloofar Mansoor, Jonathan A. Saddler, Bruno Silva, Hamid Bagheri, Myra B. Cohen, and Shane Farritor. 2018. Modeling and testing a family of surgical robots: an experience report. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. 785–790. https://doi.org/10.1145/3236024.3275534

[24] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. 2015. PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 6235–6240. https://doi.org/10.1109/ICRA.2015.7140074

[25] PX4 Forum poster. Last accessed 4/16/24; Posted Oct 2019. Strange Harrier D7 crash. https://discuss.px4.io/t/strange-harrier-d7-crash/13480

[26] PX4 Forum poster. Last accessed 4/24/24; Posted Dec 2019. Drone not stable in flight. https://discuss.px4.io/t/drone-not-stable-in-flight/14452

[27] Salil Purandare, Urjoshi Sinha, Md Nafee Al Islam, Jane Cleland-Huang, and Myra B. Cohen. 2023. Self-Adaptive Mechanisms for Misconfigurations in Small Uncrewed Aerial Systems. In *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 169–180. https://doi.org/10.1109/SEAMS59076.2023.00030

[28] PX4-Autopilot. Last Accessed 4/16/24. PX4 Online Discussion forum. https://discuss.px4.io/

[29] Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA.

[30] Emma Simpson. 2023. Amazon pledges parcels in an hour using drone deliveries. *BBC News* (October 2023). https://www.bbc.com/news/business-67132527

[31] Andrés Felipe Solis Pino, Jose García Alonso, Enrique Moguel, Cristina Vicente-Chicote, Julio Ariel Hurtado Alegria, and Pablo H. Ruiz. 2022. Software Product Lines for Industrial Robots: A Pilot Case with Arduino. In *Gerontechnology IV*, José García-Alonso and César Fonseca (Eds.). Springer International Publishing, Cham, 55–66.

[32] Stanford Artificial Intelligence Laboratory et al. [n. d.]. *Robotic Operating System*. https://www.ros.org

[33] Katie Tarasov. 2023. Amazon's 100 drone deliveries puts Prime Air far behind Alphabet's Wing and Walmart partner Zipline. *CNBC* (May 2023). https://www.cnbc.com/2023/05/18/amazons-100-drone-deliveries-puts-prime-air-behind-google-and-walmart.html

[34] ArduPilot Dev Team. Last Accessed 04/25/24. ArduPilot Documentation. https://ardupilot.org/plane/docs/

[35] Terry Therneau, Beth Atkinson, and Brian Ripley. 2023. rpart: Recursive Partitioning and Regression Trees. https://CRAN.R-project.org/package=rpart

[36] Kentaro Yoshimura, Yuta Yamauchi, and Hideo Takahashi. 2023. Managing Variability of Logistics Robot System: Experience at Hitachi. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A (SPLC '23)*. 234–241. https://doi.org/10.1145/3579027.3608995