SWARMBUG: Debugging Configuration Bugs in Swarm Robotics

Chijung Jung
University of Virginia
Charlottesville, Virginia, USA
cj5kd@virginia.edu

Ali Ahad
University of Virginia
Charlottesville, Virginia, USA
aa5rn@virginia.edu

Jinho Jung
Georgia Institute of Technology
Atlanta, Georgia, USA
jinho.jung@gatech.edu

Sebastian Elbaum
University of Virginia
Charlottesville, Virginia, USA
selbaum@virginia.edu

Yonghwi Kwon
University of Virginia
Charlottesville, Virginia, USA
yongkwon@virginia.edu

ABSTRACT

Swarm robotics collectively solve problems that are challenging for individual robots, from environmental monitoring to entertainment. The algorithms enabling swarms allow individual robots of the swarm to plan, share, and coordinate their trajectories and tasks to achieve a common goal. Such algorithms rely on a large number of configurable parameters that can be tailored to target particular scenarios. This large configuration space, the complexity of the algorithms, and the dependencies with the robots' setup and performance make debugging and fixing swarms configuration bugs extremely challenging. This paper proposes SWARMBUG, a swarm debugging system that automatically diagnoses and fixes buggy behaviors caused by misconfiguration. The essence of SWARMBUG is the novel concept called the degree of causal contribution (DCC), which abstracts impacts of environment configurations (e.g., obstacles) to the drones in a swarm via behavior causal analysis. SWARMBUG automatically generates, validates, and ranks fixes for configuration bugs. We evaluate SWARMBUG on four diverse swarm algorithms. SWARMBUG successfully fixes four configuration bugs in the evaluated algorithms, showing that it is generic and effective. We also conduct a real-world experiment with physical drones to show the SWARMBUG's fix is effective in the real-world.

CCS CONCEPTS

• **Computer systems organization** → **Robotics**; • **Software and its engineering** → *Development frameworks and environments.*

KEYWORDS

debugging, configuration bug, swarm robotics

ACM Reference Format:
Chijung Jung, Ali Ahad, Jinho Jung, Sebastian Elbaum, and Yonghwi Kwon. 2021. SWARMBUG: Debugging Configuration Bugs in Swarm Robotics. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468601

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468601

1 INTRODUCTION

In robotics, a swarm is a group of cooperative robots that is able to solve complex tasks through their collective behavior [24]. Swarms are being used to solve many real-world problems, from environmental monitoring and emergency response to entertainment [73]. Key enablers of such success are the algorithms that allow the individual robots of the swarm to plan, share, and coordinate their trajectories and tasks to achieve a common goal [17].

Despite the potential of swarms, developing robust swarm algorithms is challenging. (1) Swarm algorithms are dependent on a large number of related parameters and inputs that can significantly change the behavior of the swarms. The swarm algorithm controls multiple robots adding an order of magnitude in complexity to a large number of parameters used to configure each robot (e.g., ArduCopter [5] has hundreds of configuration parameters). (2) Swarm operations are highly dynamic, compounding the variability and sensitivity of all its robots to the environment. (3) Swarm algorithms have variables and code blocks that are highly inter-dependent. The algorithms are often a closed-loop (feedback) control system [29, 54] which continuously computes robots' new states using new inputs and their previous states.

In our conversation with developers of swarm algorithms [2, 77] and observation from public forums [42, 49, 79, 80, 92], one of the common challenges in swarm algorithms and robotics development is to find appropriate values for configurable parameters. A slightly misconfigured parameter can cause a buggy behavior, which we call *configuration bugs*. This paper focuses on *configuration bugs* in swarm algorithms (i.e., bugs caused by misconfiguration of the algorithms and robots), causing incorrect swarm states (such as crashing drones) in a particular deployment scenario.

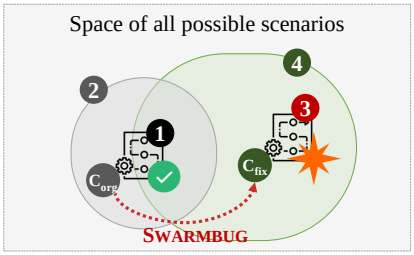


Figure 1: Illustration of a configuration bug and SWARMBUG.

Configuration Bugs. Figure 1 illustrates a high-level concept of the configuration bug and the SWARMBUG's ultimate objective. Given the space of all possible scenarios (S_{all}) of a swarm, there is a configuration for the swarm (C_{org}) that can result in a successful scenario (S_{succ}) denoted by ❶. ❷ denotes scenarios that can be successfully covered by C_{org}. A configuration bug happens when a swarm operates under a new scenario resulting in a failure S_{fail} because it is not covered by C_{org}.

Challenges. A typical debugging approach for a configuration bug might be tracking each parameter's value propagation to the robot's decision that caused a faulty scenario. Unfortunately, the aforementioned complexity of swarm algorithms makes this approach impractical. For example, parameters often go through a number of complex computations with other variables, including matrix multiplications. Precisely tracking a variable's impact after those computations is an extremely challenging task. Another typical approach is trial-and-error. A developer inspects a particular variable's value, modifies its value, and tests whether it will fix the bug. The debugging process typically requires non-trivial manual effort due to many configurable parameters and complex dependencies. Without proper guidance on each trial-and-error, this approach is rather impractical. Moreover, even after the developer identifies a potential fix (i.e., a new value for a configurable parameter), testing the fix in various scenarios is time-consuming and challenging due to the large space of possible swarm behaviors.

Our Approach. This paper proposes SWARMBUG, a swarm debugging approach for configuration bugs. As illustrated in Figure 1, it aims to find a new configuration which we call a fix C_{fix} that can cover more scenarios (❹). While not guaranteed, SWARMBUG prioritizes C_{fix} that are close to the C_{org}, which can potentially cover some of the scenarios already covered by C_{org} (❷) (as per the overlapping area of ❷ and ❹).

In particular, SWARMBUG targets bugs that are caused by misconfiguration of the swarm algorithm or robot's parameters (i.e., configuration variables). It aims to (1) find key variables that caused a buggy behavior, (2) identify possible fixes for the bug via systematic testing, and (3) rank the fixes that preserve the behavior of the original execution. SWARMBUG's key enabling technique is the novel concept of the degree of causal contribution (DCC). It creates alternative executions with and without critical factors (e.g., objects) that affect the swarm's behavior to understand which factors are causally contributing to the buggy behavior. SWARMBUG then finds variables that can configure swarm algorithms to adjust the DCC of the factors. The contributions of this research are as follows:

- We develop a swarm robotics debugger for configuration bugs.
- We propose the concept of DCC to understand the degree of causal contribution of each variable to swarm behavior and use it to precisely pinpoint critical variables that contribute to bugs.
- We evaluate our algorithm on 4 real-world swarm algorithms and automatically identified 7 valid bug fixes, including physical flight experiments with real-world drones to empirically show that the generated fixes are effective in real-world scenarios.
- We have communicated and confirmed all the configuration bugs and our fixes with the authors of the swarm algorithms.
- We publicly release the source code and data of SWARMBUG on *https://github.com/swarmbug/src*.

2 MOTIVATING EXAMPLE

We use the Adaptive Swarm [2] algorithm to illustrate SWARMBUG's operation. We run the algorithm for four drones: one leader and three follower drones (F1~F3). The algorithm's goal is to safely move the swarm to a destination while maintaining a diamond-shape formation as shown in Figure 2-(a). The arrows with borders (either blue or gray) indicate the drone's flight direction. Orange arrows are the vectors caused to avoid obstacles (including other drones). Gray arrows represent the vector to maintain the diamond formation. When there are multiple vectors considered, the blue arrows with borders indicate the final flight directions.

Configuration Variables. In this example, there are two types of configuration variables: environment and swarm configuration variables. Environment configuration variables represent objects such as robots and obstacles (e.g., *followers[0~ 1].sp*, *self.sp.x*, and *obstacle[8]* in Figure 3). Swarm configuration variables are parameters for swarm algorithm and robots. For example, circles surrounding drones visualize a parameter *infl_radius* that determines the maximum sensing distance for objects. *interrbt_dist* is another parameter that represents the desired distance between drones.

Configuration Bug. Figure 2-(b)~(e) show such a scenario where F3 crashes with an obstacle due to a *configuration bug*. First, the *moving obstacle* approaches F1, which is also moving, in (b) and makes F1 move towards the south-west, leading F1 to get close to F3. In (c), the obstacle forces F1 and F3 closer. In (d), the obstacle approaches now F3 which fails to avoid it because the other four forces come into play: three forces to avoid F1, F2, and obstacles (oranges), and the force to maintain the formation. This causes F3 to move just slightly from its current position, not enough to avoid the obstacle, leading to a crash in (e). A cause for the failure is that, in (d), F3 was too close to adjacent drones which interfere with the decision of F3 to avoid the obstacle.

Debugging Attempts without SWARMBUG. A typical debugging approach of the given bug is to trace the value propagation from the obstacle (i.e., the cause of the crash) to the drone to understand how the obstacle and other variables affect the drone's faulty decision. For example, one may use existing program analysis techniques such as taint analysis [7, 18, 40, 67, 76] to trace *obstacle[8]* which is an environment configuration variable (defined as a global variable) representing the obstacle. Each drone in the swarm reads this variable to determine whether they are close to the moving obstacle or not. However, tracking the value propagation of the variable is challenging as it goes through complex computations.

Figure 3 shows a simplified value propagation graph. The arrows in Figure 3 show the data propagation paths. The source variable (*obstacle[8]*) is a 2×4 array and the values of its elements (along with other variables including *followers[0].sp* and *followers[1].sp* representing other drones) are used to generate each element of a 500×500 array, *d2*. Later, each element of *d2* is used to create another 500×500 array *repulsive* with *infl_radius* and *nu*. Then, each element of *repulsive* and *attractive* are added to create *total* (a 500×500 array). Finally, it computes a gradient of the matrix to create *gx* and *gy*. Finally, mean values of the gx and gy arrays to compute x (*self.sp.x*) and y (*self.sp.y*) coordinates. At this point, which part (of bytes) of the x and y coordinates are affected by the source variable *obstacle[8]* is challenging to know. Using taint

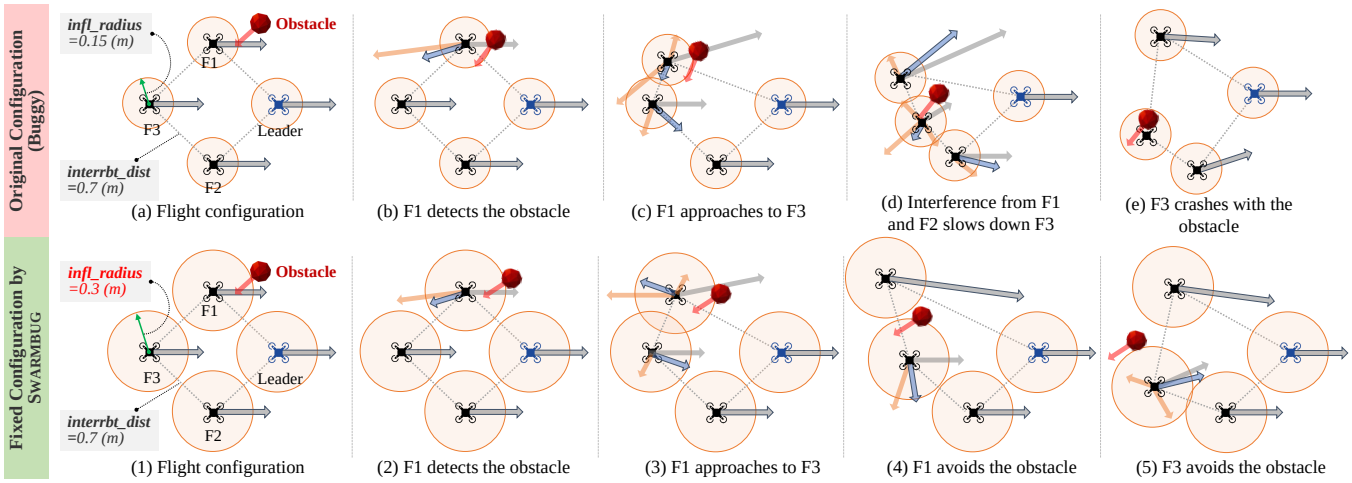


Figure 2: Swarm of four drones crashing an obstacle: (a)~(e). The same swarm mission with a fix by SWARMBUG: (1)~(5).

analysis would tell that every part of both coordinates depends on the source variable and other variables, which are not useful for debugging the configuration bug. Note that the graph is simplified. The complete graph of the swarm algorithm [2] is at least 10 times larger than Figure 3. A backward edge from the *self.sp.x* and *self.sp.y* to *followers[0].sp* and *followers[1].sp*, that forms cycles, are omitted.

Debugging with SWARMBUG. SWARMBUG (1) conducts a behavior causal analysis to find out environment configuration variables that caused the bug, (2) obtains bug fixes by mutating swarm configuration variables, and (3) ranks fixes that preserve the original behavior of the swarm.

(1) Cause Analysis: Given a definition of configuration variables provided by a user, SWARMBUG infers which configuration variables significantly contribute to the buggy behavior by leveraging a concept we call the *degree of causal contribution* (or DCC, details in Section 4.1.2). DCC essentially abstracts the impact (or contribution) of individual variables to a robot's decision.

DCC is computed as follows. Given the original execution (demonstrating the crash), SWARMBUG creates alternative executions by removing the impact of environment configuration variables (that are essentially related to surrounding objects and robots). Then, we compare the robots' behaviors of the original execution and the alternative executions. The difference of the robots' poses becomes a DCC value. Finally, we analyze the trends of DCC values around the time when the bug occurred to pinpoint the cause of the bug (e.g., whether some variable's contribution is insufficient or excessive). Note that SWARMBUG does not rely on tracking complex propagations of values, which existing techniques struggle to do, but rather analyzes values related to the robots' behavior as the environment is changed.

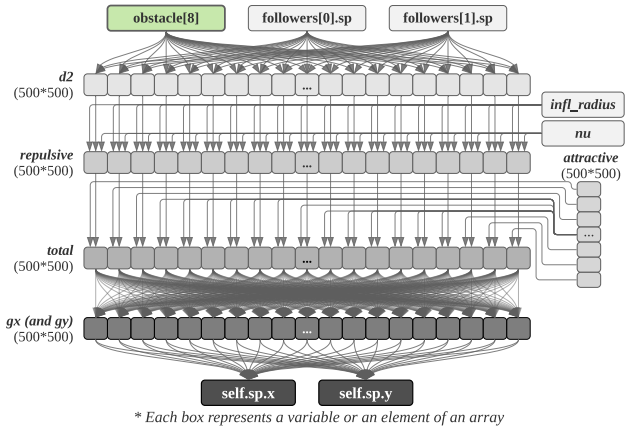


Figure 3: Illustration of simplified value propagation.

In the earlier example, SWARMBUG derives *alternative executions without each obstacle* by mutating environment configuration variables, to infer the causal relationship between an obstacle and the buggy behavior. Then, we compare each drone's poses observed during the generated alternative executions and the original execution, obtaining the difference that represents the impact of each removed obstacle to the buggy behavior. To this end, SWARMBUG identifies the most impactful variable: *obstacles[8]* (a moving obstacle).

(2) Finding Potential Configuration Fixes: From the environment configuration variable that contributes to the bug, SWARMBUG conducts a number of experiments that change each *swarm configuration* variable's value (e.g., a robot's parameter's value) to identify potential fixes for the bug. Specifically, it focuses on the trend of DCC values of the environment configuration variable. For example, we earlier noticed that the obstacle's contribution becomes more significant near the crash while other objects (e.g., other drones) also compete for the contribution.

To this end, SWARMBUG tries to *reinforce* (or intensify) the increasing trend of the moving obstacle's DCC value. With the change, we expect the drone to take the obstacle into account more significantly than the original execution. We then run multiple executions with mutated swarm configuration variables (e.g., increasing/decreasing their values) to find mutations that can reinforce the trend. Finally, we find concrete values for two swarm configuration variables (defined as global variables), leading to *two configuration fixes*: (i) *infl_radius*=0.3 and (ii) *interrbt_dist*=1.4.

(3) Validating the Robustness of Fixes: SWARMBUG tests the two fixes (i.e., *infl_radius* and *interrbt_dist*) exhaustively, by running a number of tests with diverse scenarios that SWARMBUG derived by profiling the variation of the target scenario (e.g., spawning the

swarm in various positions). To make each test more meaningful in terms of validating the robustness, SWARMBUG measures whether each run exercises observable new swarm behaviors using DCC values. Specifically, for each test, we collect DCC values and compute MSE scores against previous executions' DCC values. The testing is repeated until it does not observe new swarm behaviors (e.g., MSE scores of 100 consecutive executions are all smaller than 0.01) or reached a predefined timeout (e.g., 20 hours). In this example, both fixes successfully pass the testing, meaning that SWARMBUG did not observe any failures after 20 hours of testing while the fixes with *infl_radius* and *interrbt_dist* successfully finishes 3,880 and 1,211 tests respectively. Hence, both are considered as valid fixes.

(4) Finding Behavior-preserving Fixes: Some fixes may *disruptively* change the swarm behavior. For instance, in our example, changing *interrbt_dist* results in a bigger diamond formation, making the swarm look and behave quite differently. To avoid such fixes, SWARMBUG aims to identify a *behavior-preserving* fix which behaves similar to the original swarm. Specifically, we compare the DCC values from a fixed execution and the original execution to measure the differences between the two executions. If two swarm executions have similar DCC values, we consider that their behaviors are similar. In our example, the DCC values from the fix with *infl_radius* is more similar to the DCC values from the original run than the fix with *interrbt_dist*.

Chosen Fix: Figure 2-(1)~(5) show the flight with the *infl_radius* fix. It maintains the same formation, while individual drone detects and avoids the obstacle earlier, preventing the situation where multiple drones get too close (2)~(4). All the drones, including F3, avoid the obstacle successfully (5).

3 BACKGROUNDS, GOALS, AND SCOPE

3.1 Mobile Robot Software

Configurable Variables. A typical robot such as the drones we use in our studies can have hundreds of configurable parameters and each of the parameters can affect the robot's behavior significantly. A robot's decision-making process is typically implemented as a sequence of program statements that *continuously and iteratively* reads inputs from various sensors and computes the robot's next state, meaning that it is essentially a closed-loop system [98]. During the computation, the configurable parameters are also taken into account. As shown in Figure 3, variables in the algorithms are highly inter-dependent (e.g., most variables in the loop are dependent on their previous iteration's values), making it difficult to apply data-dependency analysis techniques.

Field Testing and Simulation-based Testing. Testing robotics algorithms is challenging because robots interact with the physical surroundings. While testing robots in the real-world (field testing or physical testing) is desirable and ultimately required, it is expensive and dangerous due to the cost of failures. As a result, simulation-based testing is a common alternative that can reduce development and validation costs. Still, given the dimension and complexity of the real-world, simulation-testing must identify what scenarios are worth validating and attempt to reduce the exploration of equivalent scenarios that render little value for testing.

3.2 Swarm Algorithms

Centralized and Distributed Swarm Algorithms. There are two main lines in constructing swarm algorithms [8, 10, 17, 37, 56]: centralized and distributed. A centralized algorithm [14, 22, 55] computes all the decisions of individual robots in a swarm in a centralized system. On the other extreme, a distributed swarm algorithm [6, 45, 94] runs the majority of the algorithm on individual robots, where robots are communicating via network channels. Existing approaches such as taint analysis have difficulty handling distributed algorithms while SWARMBUG works well on both centralized and distributed algorithms.

Local vs Global Goals. Swarm algorithms may have global goals for the entire swarm and local goals for individual robots at the same time, leading to *conflicting goals*. For instance, each robot may have a local algorithm to avoid obstacles, while a swarm algorithm aims to maintain a specific formation during the flight. When a robot in the swarm encounters an obstacle, the robot's local algorithm may hold back the swarm algorithm's progress as it prioritizes its local goal (i.e., avoiding the obstacle). Note that even if a swarm algorithm includes logic to balance the two goals (e.g., prioritizing local and global goals based on the current state and environment), the balancing logic may not be perfect, failing to balance the conflicting goals.

Complex Dependencies. As a swarm consists of multiple robots, the complexity of dependencies among variables and configurations has significantly increased compared to that of a single robot. During our experiments, we observe that the average number of data dependencies (i.e., the number of edges in the data dependence graph) in drone swarm algorithms [36, 50, 51, 61, 93] is '1,693+1,207∗n' where n represents the number of robots.[1] When n=5, the number is approximately 3.7 times the average number of dependencies of algorithms for a single drone which is 2,042 [15, 25, 27, 59, 68] (with n=10, the swarm algorithms' dependencies are 6.7 times bigger than the single drone algorithms). It means that applying the data dependency analysis to swarm algorithms is ineffective in practice.

Dynamic Behaviors. In a swarm, individual robots' dynamic behaviors are often accumulated and amplified, leading to even more diverse swarm behaviors. For example, in our motivation example, Figure 2-(c) and (d) have a chain reaction to the obstacle, which is different from when an individual drone interacts with an obstacle. Hence, a significant challenge in swarm testing is obtaining test cases that can effectively cover various swarm behaviors and prioritizing test cases to cover diverse scenarios.

3.3 Goals and Scope

Goals of SWARMBUG. SWARMBUG aims to achieve the three major goals to effectively debug swarm algorithms as follows.

- **Goal-1**: *Developing effective causal analysis capabilities for swarm algorithms* to automatically identify root causes of configuration bugs and find fixes.
- **Goal-2**: *Developing an effective and efficient testing approach* to validate bug fixes for swarm algorithms by systematically covering various corner cases.

[1] As for '1,693' and '1,207', we use the data-dependency graph using Sourcetrail [75], with T as the total edges of the swarm algorithm and L as the number of edges for an individual drone algorithm. '1,693' is the average of the difference between T and L, and '1,207' is the average of L of all drones.

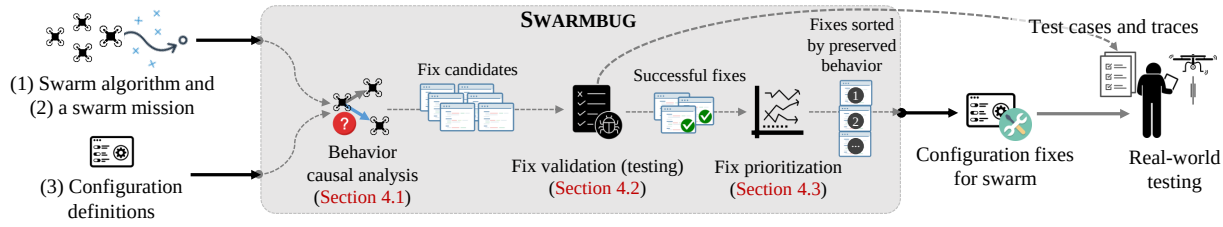


Figure 4: Overview of SWARMBUG

- **Goal-3**: *Understanding the impact of fixes and guiding how to choose fixes* that preserve the original swarm algorithm's behavior while correcting buggy behaviors.

Focus on Unmanned Air Vehicles (Drones). While our findings and insights are generic and applicable to various swarm robotics environments, our research focuses on swarm robotics algorithms for unmanned aerial vehicles. This is because (1) they are prevalent and used in various missions, and (2) they have one of the most sophisticated dynamics, leading to various challenges in debugging.

Generality of SWARMBUG's Fix. SWARMBUG generates fixes for a bug under a particular mission and algorithm's configuration. This means that the fixes may not work for a significantly different mission or scenario. For instance, a bug fix for a swarm mission with four drones may not work for a mission with eight drones. Also, a bug fix for a swarm avoiding obstacles may not work if the obstacles' speed changes (e.g., become faster).

4 DESIGN

Figure 4 shows the overall procedure of SWARMBUG. It takes three inputs: (1) a swarm algorithm's source code, (2) a swarm mission that triggers a buggy behavior, and (3) configuration definitions that include a list of configuration variables for the swarm and environment (e.g., certain obstacles, wind, etc.). SWARMBUG conducts a behavior causal analysis (Section 4.1) to find causes of buggy behaviors from environment configuration variables and generate fixes for swarm configuration variables. Then, SWARMBUG validates the fixes under various scenarios (Section 4.2) to obtain robust fixes. Further, it ranks the fixes based on the behavior similarity between the original swarm and the fixed swarm (Section 4.3). Finally, while it is not part of our main contribution, the test cases and traces can be used to conduct real-world testing as shown in Section 5.2.1.

4.1 Behavior Causal Analysis

4.1.1 Configuration Variables. Among the variables in a swarm algorithm, there are two types of variables that are important in understanding and controlling behavior: environment and swarm configuration variables. One of the SWARMBUG's inputs is the configuration definitions: a list of configuration variables with each variable's type (either environment or swarm configuration) and the value specification.

1. **Environment Configuration Variables** define the environment of the swarm that can be manipulated during simulation such as obstacles, robots, and wind. The value specification includes a value to eliminate the impact of the variable. For instance, if an obstacle is defined as a set of coordinates, coordinate values outside of the map will effectively remove the obstacle. We use the ∅ symbol to represent such a value.
2. **Swarm Configuration Variables** typically define parameters of drones and swarm algorithms. The specification includes the range of values (i.e., minimum and maximum values, distribution). For instance, the maximum drone velocity or the minimum distances between drones in a swarm.

Profiling for the Configuration Definitions. SWARMBUG expects a user to provide the configuration definitions[2], which may require non-trivial effort. To mitigate this, we present a set of profiling tools and supporting approaches on our project website [82] that can generate sketches of such configuration definitions for implementations like the ones we present later in our study [2, 16, 60, 88] to reduce such effort.

4.1.2 Degree of Causal Contribution (Dcc). Our analysis targets environment configuration variables that represent obstacles and other robots because *they directly affect the swarm behavior* and are crucial in understanding causes of bugs. A key innovation of SWARMBUG is the concept of *the degree of causal contribution (or Dcc)* of a variable to a robot's pose and propose its computation without relying on complex data propagation analysis techniques such as taint analysis. Dcc is computed by comparing differences between executions with mutations applied on the environment configuration variables.

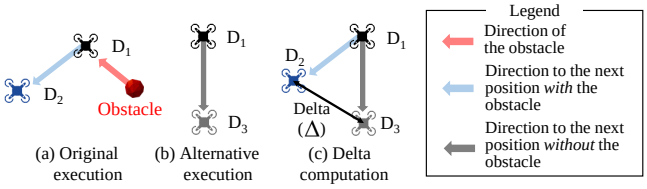


Figure 5: Example of computing a delta (Δ) value.

Computing Delta (Δ) via Alternative Execution. To understand the contribution of an environment variable, we first create a new (alternative) execution with a mutation on the variable that can essentially remove the variable's presence in the environment. Since the new execution negates the existence of the mutated variable, we call the new execution *alternative execution*. Figure 5 shows an example. Suppose that Figure 5-(a) shows an original execution that includes an obstacle, leading to the drone moving toward the

[2]Details of the configuration definitions and the real input file we use in this paper can be found on *https://github.com/swarmbug/src/tree/main/Input_Swarmbug*

Algorithm 1: Computing Dcc from the Delta values

```
Input  : M: a set of missions for robots. m_r ∈ M is a mission for robot r,
         T_e: the tick value of when the swarm mission finishes.
         V_ec: a set of environment configuration variables.
Output : Dcc(r, t): a set of tuples <v_ec, N> where v_ec is an environment configuration
         variable and N is the Dcc value of v_w at tick t for robot r
1  procedure ComputeSwarmDcc(M, V_w)
2      t ← 0
3      while t ≠ T_e do
4          for m_r ∈ M do
5              Dcc(r, t) ← ComputeRobotDcc (m_r, V_ec, t)
6          t ← t+Time-step // Time-step represents a single tick
7  procedure ComputeRobotDcc(r, V_ec, t)
8      Δtotal ← 0
9      P_org = GetRobotPose (r, V_ec, t) // Obtain a pose of r at t
       // Each source variable v_i representing a world object
10     for v_i ∈ V_ec do
11         tmp ← v_i // Save v_i
12         v_i ← ∅ // Removing the impact of an environment configuration variable v_i
13         P_i = GetRobotPose (r, V_ec, t) // Obtain a pose of r at t without v_i
14         Δi ← ||P_org − P_i|| // Δ for v_i via Euclidean Distance
15         Δtotal ← Δtotal + Δi
16         v_i ← tmp // Restore v_i
17     dccSet ← {}
18     for v_i ∈ V_ec do
19         dccSet ← dccSet ∪ <v_i, (Δi / Δtotal) >
20     return dccSet
```

south-west (from D_1 to D_2). A counterfactual execution is shown in Figure 5-(b) without the obstacle. The drone moves toward the south (from D_1 to D_3). As shown in Figure 5-(c), we obtain a delta by computing the Euclidean distance between drones' poses (D_2 and D_3) from the two executions.

Computing Dcc. The degree of causal contribution (or Dcc) is an aggregation of the delta (Δ) values of environment configuration variables. Specifically, we obtain Δ values of all environment configuration variables. Then, we compute the percentages for each variable, resulting in Dcc.

Algorithm 1 shows the details of Dcc computation. Given a swarm algorithm, it iterates over all the robots in the swarm and calls *ComputeRobotDcc* for every tick to obtain all Dcc values in the given mission M (lines 1-6). Then, it obtains the robot's pose (i.e., coordinate) in the original mission at the given tick t by calling *GetRobotPose* and stores the results to P_{org} at line 9. We remove each environment configuration variable's impact (i.e., v_i) by assigning ∅ to v_i. Next, we obtain a new robot's pose (line 13) without the object v_i, and store it to P_i. We compute delta Δi by calculating Euclidean distance between P_{org} and P_i (line 14). We modify v_i's value on each iteration to remove the object (line 12), and restore it (line 16). Finally, we construct a set of proportions of individual variables' deltas (line 19).

4.1.3 Temporal Analysis. We analyze how Dcc values change over time (i.e., trend) to identify the causes of a bug.

Time Window for Temporal Analysis. Robots typically have some lag in recognizing and reacting to changes in their surroundings. We call such time duration T_{win} (or time window for temporal analysis), and focus on the trend of Dcc values within the window. Note that different swarm algorithms may have different time windows so test missions are typically provided by the developers or can be obtained with slight changes of their configuration. Then, we identify when the current Dcc value is changed more than 10% than its previous tick's Dcc value (i.e., Dcc value is rapidly changing). Note that the 10% threshold is configurable[3].If such rapid changes are observed, we record how long the changing trend lasts. We calculate the average time they last and use it for the time window, T_{win}. In this paper, we measured T_{win} values of 7.6 ticks, 100 ticks, 6 ticks, and 3 ticks for Adaptive Swarm [2], Swarmlab [88], Fly-by-logic [60], and Howard's [16] respectively.

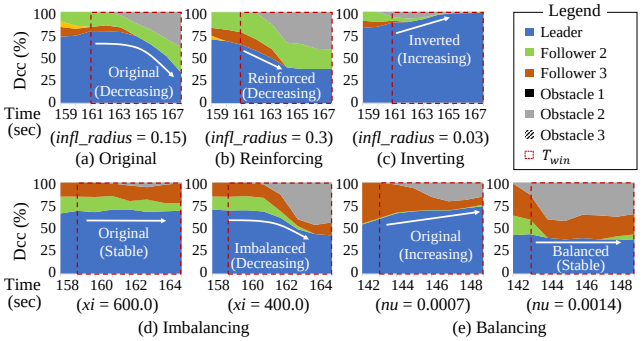


Figure 6: Examples of Dcc value trends and fixing strategies.

Fixing Strategies Based on Dcc Trends. With the identified T_{win}, we try to identify a temporal trend of Dcc values within a family of predefined templates, as shown in Figure 6, that reflect our experience in practice. Then, we apply a set of predefined fixing strategies depending on the matching temporal trend template. From the time that it causes a buggy behavior, T_{bug}, the time window for our temporal analysis starts at '$T_{bug} - T_{win}$' and ends at 'T_{bug}', as shown in Figure 6. Then, we apply the following four strategies.

1. **Reinforcing.** If the trend of Dcc values is either increasing or decreasing, we try to *reinforce* the trend (i.e., increasing or decreasing more). Figure 6-(a) shows an example of a decreasing trend of Dcc values. Figure 6-(b) is a fix obtained by changing the value of *infl_radius* (a swarm configuration variable that represents the maximum sensing distance for objects) to 0.3 from 0.15 (the original value shown in Figure 6-(a)).
2. **Inverting.** If Dcc values are increasing/decreasing, we generate a fix to invert (i.e., decrease/increase) the trend of Dcc values, respectively. For example, Figure 6-(c) inverts the trend of Dcc values from Figure 6-(a) by changing the value of *infl_radius* to 0.03 (from 0.15). This strategy is effective when a swarm overlooks an essential factor and focuses on trivial inputs. It would invert the focus so that the essential factor can be considered.
3. **Imbalancing.** If a Dcc value of the variable does not have noticeable changes, we try to introduce changes that can lead to different swarm behavior. We first try to imbalance (i.e., either increase or decrease) the Dcc values.
   For example, Figure 6-(d) introduces a decreasing trend by changing the value of *xi* (a swarm configuration variable) to 400 from 600. *xi* represents the non-leader robot's tendency of following the leader drone. Reducing this value allows robots to focus on other surroundings.
4. **Balancing.** Swarm algorithms may fail because they accidentally take some inputs into the computation more or less than they should be. This strategy will try to reduce the impact of

[3]The optimal for each algorithm can be profiled. Details can be found in [82]. We profile the four algorithms we evaluated, and find that 10% works for all of them.

overly-prioritized objects in algorithms. For example, Figure 6-(e) changes the value of *nu* from 0.0007 to 0.0014. *nu* swarm configuration variable representing the priority of avoiding obstacles over other goals (e.g., following the leader). The fix prevents the drone from being overly considering the leader.

4.2 Fix Validation

4.2.1 Profiling Spatial Variations. It is common to observe a swarm behaves differently between each test. A robust fix should be tested under such diverse behaviors. To understand the variation of a given swarm algorithm, we profile the drone's poses from tests.

Aligning Spatial Coordinates. Spatial coordinates of the swarm can vary across the test runs. For example, two relatively identical flights can have different coordinates if the entire swarm's poses are shifted. To identify the variation of drones' poses in the swarm, it is necessary to align the drones' poses based on common coordinate system. Specifically, we set the spatial coordinates of the swarm on the drone that caused a bug (e.g., a crash). Other objects including other drones and obstacles are referenced accordingly.

Computing Spatial Variations. We run n sets of tests where each set includes N tests (N = 10 in this paper), until we reach a fixed point of the spatial variation. We measure the spatial variation of the drones' poses from all the test runs on each test set. For measuring the spatial variation SV, we leverage the concept of circular/spherical error probable (CEP/SEP) [20] to identify the area that can include 90% of coordinates from the total tests.

On the i^{th} test set, we measure the spatial variation of the drones' poses (SV_i) from all the test runs executed at this point ($i * 10$ tests). We repeat the process until we observe SV_{i-1} and SV_i do not differ more than 5%. In general, we reach the fixed point with 10 test sets, meaning that we run 100 tests in total. Details can be found on [82].

To this end, we obtain a map called SVMAP (Spatial Variation Map) that shows the aligned spatial variations of individual robots and objects. Figure 7 shows an example SVMAP obtained from Adaptive Swarm [2]. In the map, observed robots are presented as points. Solid contour lines indicate areas that are estimated as the same density. The contour lines represent areas that contain the sample's population from 10% to 90%, where the outmost area includes 90%, and each inner area has 10% less population.

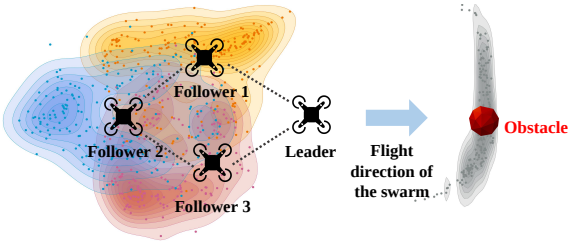


Figure 7: Spatial Variation Map (SVMAP)

4.2.2 Feedback-driven Fuzzing. We validate the generated fixes by testing them under various scenarios. We use SVMAP, which represents the spatial variation of the swarm under test. We aim to spawn robots and obstacles within the regions shown SVMAP.

Initially, we spawn them in inner layers more than outer layers (because more drones were observed there during the profiling). During the tests, we record Dcc values. If the Dcc values of the current test differ by *more than 10% from all the previously observed Dcc values*, we consider the test covered some new swarm behaviors, hence a meaningful test covering a new scenario. In this case, we prioritize creating new tests that are similar to the current one. If the Dcc values from the current testing are similar to Dcc values from previous tests, we prioritize the other layers. Note that we essentially use Dcc values as feedback representing the behavior of the swarm. If we tried all the layers and cannot find new Dcc values that are more than 10% different from the previous tests, we extend the layers to cover larger spaces.

The process terminates (1) when the test fails (e.g., robots crashing to obstacles or walls) or (2) reaches a predefined timeout. If we reach the timeout without a failure, we consider the fix is valid. During the testing, if we observe any crashes or runs that fail to reach the original goal, we consider them unsuccessful runs, and the corresponding fixes are discarded.

4.3 Fix Prioritization

The fixes by SWARMBUG may affect different aspects of the swarm behavior in an undesirable way. For example, a fix may resolve a crash by changing the swarm's formation significantly (increasing the distances between drones). In such a case, the swarm with the fix may look very different from the original one.

To this end, we *rank* the fixes by how much they preserve the original algorithm's behavior. Specifically, for each fix, we compare Dcc values from the swarm with the original configuration and fixed configuration. Then, we rank the fixes with smaller differences higher because they preserve the original behavior of the swarm more than those with larger differences in Dcc values. In many cases, a higher-quality fix does not significantly change the swarm's behavior while eliminating the fault bug. Note that we essentially use Dcc to approximate the swarm behavior.

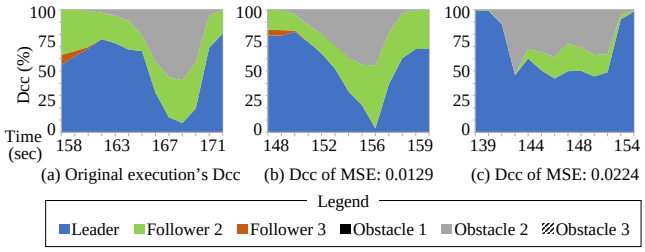


Figure 8: Example MSE scores

Measuring Distances of Dcc Values. To compare Dcc values from different runs, we leverage the Mean Squared Error (MSE). Figure 8-(a) shows Dcc values from the original execution, and Figure 8-(b) and (c) are Dcc values from executions with two different fixes. We rank the one with a smaller MSE value (0.0129) higher than the other with MSE value 0.0224. Note that we make them have the same length using interpolation (i.e., applying linear interpolation to the shorter sequence), then calculate the MSE to handle Dcc values of different time periods.

5 EVALUATION

Implementation. We prototype two versions of SWARMBUG to support four swarm algorithms. One in Python (742 lines) to support Adaptive Swarm [2] and another one in Matlab (536 lines)

to support Swarmlab [88], Fly-by-logic [60], and Howard's [16]. We also modified existing simulators/emulators. Our analysis for SVMAP (Section 4.2) is written in R (632 lines).

Environment Setup. We performed our evaluation on an Intel i7-9700k 3.6Ghz and 16GB RAM, and 64-bit Linux Ubuntu 16.04. For the real-world experiment in Section 5.2.1, we use six Crazyflies [13].

Table 1: Selected Algorithms for Evaluation

Name	SLOC	Drones	Objective
Adaptive Swarm [2]	3,091	20	Flight avoiding static & dynamic obst.
Swarmlab [88]	13,213	20	Flight avoiding static obstacle
Fly-by-logic [60]	13,244	6	Optimizing path avoiding unsafe zone
Howard's [16]	1,989	20	Flight avoiding static obstacle

Swarm Algorithms. As shown in Table 1, we use four representative and diverse swarm algorithms. To select the four algorithms, we search total 23 swarm-related research papers with open sourced algorithms and 54 public GitHub repositories related to swarm robotics from 2010 to 2020. Among these, 25 came with runnable code from which we pruned out 12 that were just off-line planning algorithms not reactive to the environment, and 9 algorithms that did not exhibit collective behaviors (e.g., collections of individual drones without cooperative interactions). Finally, we end up with the selected four swarm algorithms. Details can be found in [82].

Note that while there are many swarm algorithm papers, the viable implementations are limited. We found that many repositories do not include the full implementations to support the swarm [33, 38, 62, 65, 78, 91] or do not release enough details for usage [3, 70, 99]. Others just include rudimentary implementations that do not provide basic swarm functionality or testing environments (e.g., maintaining formation, avoiding obstacles) [9, 26, 32, 53, 72, 87, 95, 96].

Table 1 shows the SLOC (Source Line of Code) of algorithms and the number of drones we used for the evaluation. We use 20 drones for all the algorithms, except for the Fly-by-logic as it does not support a swarm with up to 6 drones. The last column briefly describes the objective of each algorithm. Among the four algorithms, Adaptive Swarm is the only algorithm that enforces a particular formation during the mission. Swarmlab tries to match the speed with other robots during the mission, while the other three algorithms consider other robots as an object to avoid. Swarmlab implements two swarm algorithms: Olfati-Saber's [58] and Vicsek's [86]. We use Olfati-Saber's algorithm because Vicsek's algorithm has a bug (all the robots are disappearing after a mission starts).

5.1 Effectiveness

Buggy Behaviors. During the evaluation, we aim to fix four bug classes as shown in Figure 9 by using SWARMBUG: (a) A drone fails to avoid a moving obstacle in Adaptive Swarm, leading to a crash, (b) Drones fail to avoid the second static obstacles they encounter, crashing to the pillar structure which is a round shape object in the figure, (c) The first drone fails to avoid the unsafe zone (represented as the red cube) that the algorithm aims to go around, and (d) A drone (the green sphere) crashes into an obstacle (the red sphere).

5.1.1 Behavior Causal Analysis. Table 2 shows the result of SWARMBUG's causal analysis. "Trend" shows the identified trends of Dcc values as described in Section 4.1.3. Note that the variable name is

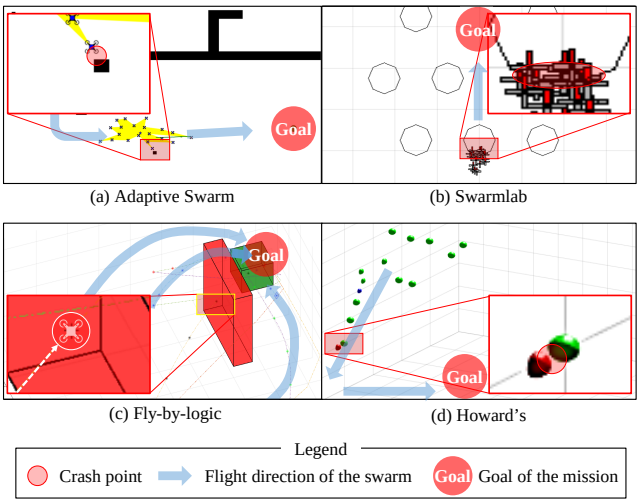


Figure 9: Buggy behaviors in the four selected algorithms

the one that dominates the Dcc values. "Strategies" shows all the fixing strategies applied. "Swarm Configuration" shows the swarm configuration variables (and their initial values) we mutate to apply the fixing strategy (e.g., reinforcing or inverting the trend of Dcc values). To achieve a target Dcc trend, SWARMBUG tries both (1) increasing the value by two times and (2) decreasing the value by 80%, and chooses one that achieves the target Dcc trend. Note that we omit several swarm configuration variables[4] that could not lead to any fixing strategies. Also, some strategies cannot be done by mutating a particular environment variable (e.g., mutating *drone_vel* does not reinforce the trend in the Adaptive Swarm's case). In such a case, we consider the strategy is not applicable and mark it as ✖. Also, there are some cases where the strategies are well achieved while the resulting execution always crashes. To check such a case, we run 10 runs for sanitization purposes. Those that fail to pass the sanitization test (e.g., drones crashing into other objects/drones) are marked as ⊗. All successfully applied strategies are annotated by ✔. It does not include the imbalancing strategy which requires the Dcc trends to be balanced, while all the observed Dcc trends are decreasing.

5.1.2 Testing Fixes. "Profiling" presents the results of 100 tests we run for spatial variation profiling (Section 4.2.1). It took approximately 25.2 (for Adaptive Swarm), 2.8 (for Swarmlab), 0.4 (for Fly-by-logic), and 0.3 (for Howard's) hours for run 100 tests. Note that they are naive testing runs where SWARMBUG further conducts fuzz testings (shown in the "Fuzzing" column) guided by MSE scores of Dcc values. In general, our fuzz testing finds more crashes (lower rates of successful runs) than the naive profiling tests, meaning that it is effective in discovering more diverse testing scenarios.

SWARMBUG initially generates 11 (for Adaptive Swarm), 6 (for Swarmlab), 4 (for Fly-by-logic), and 3 (for Howard's) fixes. Gray cells represent fixes that do not fail any tests during the profiling step. "Fuzzing" shows the number of successful tests during the fuzz-testing out of 30 hours for Adaptive Swarm and Swarmlab, 10 hours for Fly-by-logic and Howard's. Gray cells mean the fixes that

[4] In Table 2, we omit 8, 11, 4 swarm configuration variables from Swarmlab, Fly-by-logic, and Howard's respectively.

Table 2: Effectiveness of SWARMBUG

Algorithm	Trend	Behavior causal analysis: Swarm Configuration		Strategies: Reinforcing	Strategies: Inverting	Fix validation: Profiling[1] R²	Fix validation: Profiling[1] In²	Fix validation: Fuzzing[3] Reinforcing	Fix validation: Fuzzing[3] Inverting	Fix prioritization: MSE score (Rank) R²	Fix prioritization: MSE score (Rank) In²	Dev. cfm.[5]
Adaptive Swarm	Decreasing (*robot1.sp*)	*w*	(=20.0)	✔ (+20.0)	✔ (-18.0)	34	16	1195/4292 (28%)	601/4282 (14%)	-	-	-
		xi	(=400.0)	✔ (-380.0)	✔ (+400.0)	100	13	4281/4324 (99%)	477/4333 (11%)	0.024 (2)	-	✔
		nu	(=1.4E-03)	✔ (+1.4E-03)	✔ (-1.12E-03)	100	51	4060/4215 (96%)	1858/4424 (42%)	0.031 (3)	-	✔
		int_dist	(=0.7)	✔ (+0.7)	✔ (-0.56)	88	58	3922/4466 (88%)	2131/4441 (48%)	0.053 (4)	-	✔
		infl_radius	(=0.3)	✔ (-0.24)	✔ (+0.3)	11	100	411/4190 (10%)	4199/4199 (100%)	-	0.022 (1)	✔
		drone_vel	(=4.0)	✖	✔ (+4.0)	-	76	-	3052/4788 (64%)	-	0.061 (5)	✔
Swarmlab	Decreasing (*p_swarm.u_ref*)	*c_vm*	(=3.0)	✪	✔ (-2.4)	-	15	-	669/4554 (15%)	-	-	-
		b	(=5.0)	✔ (-4.0)	✖	22	-	1019/4323 (24%)	-	-	-	-
		r0	(=10.0)	✔ (+10.0)	✖	100	-	4508/4537 (99%)	-	0.021 (1)	-	✔
		c_pm_obs	(=5.0)	✖	✔ (-4.0)	-	57	-	2311/4661 (50%)	-	-	-
		d_ref	(=10.0)	✖	✔ (-8.0)	-	29	-	1167/4551 (26%)	-	-	-
		v_ref	(=6.0)	✔ (-4.8)	✖	100	-	3811/4088 (93%)	-	0.023 (2)	-	✔
Fly-by-logic	Decreasing (*obs*)	*max_vel*	(=0.8)	✔ (+0.8)	✖	78	-	3776/4896 (77%)	-	0.021 (2)	-	✔
		max_accl	(=1.0)	✔ (+1.0)	✖	60	-	2808/4888 (57%)	-	0.025 (3)	-	✔
		C	(=50.0)	✔ (+50.0)	✔ (-40.0)	100	23	4808/4901 (98%)	-	0.015 (1)	-	✔
Howard's	Decreasing (*wypt*)	*dist_thresh*	(=2.0)	✔ (+2.0)	✖	26	-	1444/6281 (23%)	-	-	-	-
		obst_pot_c[4]	(=1000.0)	✔ (+1000.0)	✔ (-800.0)	100	14	5697/6311 (90%)	831/6211 (13%)	0.011 (1)	-	✔

1: Data in Profiling column indicates the number of successful mission for 100 tests. 2: R and In indicate Reinforcing and Inverting, respectively. 3: Data in Fuzzing column indicates the number of successful mission over the number of fuzz testing in given time and success rate. 4: The program has hardcoded constants instead of variables. We assign a conceptual name to them. 5: Checkbox in this column indicates whether the bugs and fixes are confirmed by developers or not.

are most successful (e.g., more than 90% of them are successful). We run Adaptive Swarm and Swarmlab longer than the other two because a single run from the first two algorithms is much slower than the other two.

5.1.3 Fix Prioritization. As explained in Section 4.3, we obtain MSE scores of the fixes and rank them according to the scores. The most promising fixes are ranked the first in all cases. Two fixes are ranked second: *xi* and *v_ref* in Adaptive Swarm and Swarmlab, respectively. Our manual inspection shows that they are still valid fixes while they are ineffective compared to the fix ranked first.

However, *nu* in Adaptive Swarm, which is ranked third, shows abnormal behavior: it often makes robots stall or even move backward when they recognize obstacles (even if the obstacles are quite far away from them). Our manual inspection reveals that the fix prioritizes avoiding obstacles significantly more than other goals.

Confirmation from the Algorithm Authors. Throughout our research project, we have communicated with the authors of all four swarm algorithms [2, 16, 60, 88] regarding the configuration bugs we find. The bugs and fixes for the three algorithms are confirmed and acknowledged by the authors. The authors also agreed that the higher-ranked fixes are better than those that are lower-ranked.

5.2 Case Study

5.2.1 Real-world Experiment of a Fix from SWARMBUG. To show that a fix generated and validated by SWARMBUG is effective in real-world environment (e.g., with various noises), we conduct a physical experiment that uses the fixed configuration (*nu*) of Adaptive Swarm to reproduce the same flight.

Setup and Presentation. We use 6 Crazyflies [13] and leverage CrazySwarm [66] as a controller for swarming. We use a local position system (called LPS [12]) supported by Crazyflies to precisely locate drones' 3D positions in space. We conduct the experiments in the lab environment where the space is 3m × 4m × 3m (in width × length × height). We use the same trajectory (which includes drones' poses) from the Adaptive Swarm mission shown in Figure 9-(a).

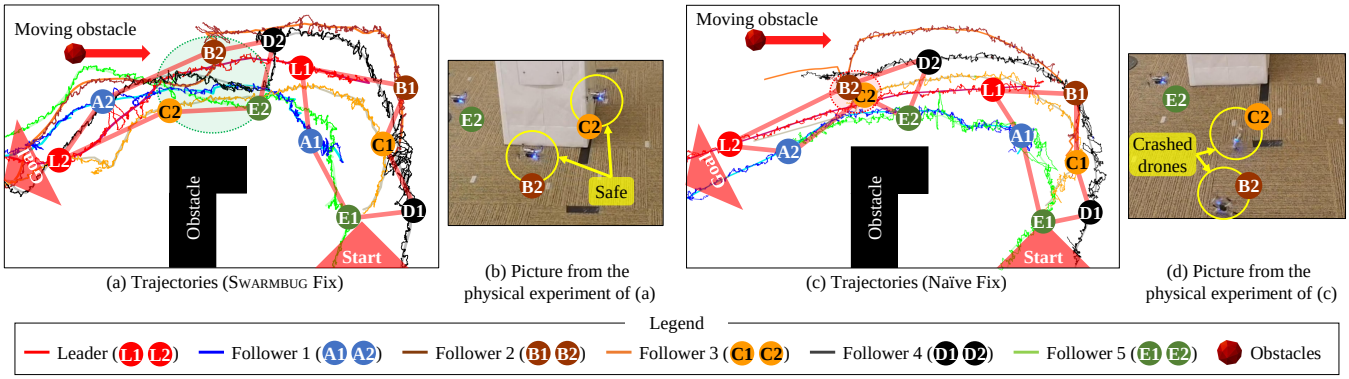
Figure 10 illustrates the results. Drones start from the right-bottom side of the map (marked as 'Start') and move toward the left (marked as 'Goal'), while avoiding obstacles. There is an L-shape static obstacle which we use two white boxes in our physical experiment. Moving obstacle (i.e., red symbol) is approaching the drones from the left to right direction in the upper side of the map. Thick lines are trajectories computed by swarm algorithms, and thin lines with jitters are the traces of the real physical drones' movements from the motion capture system [12]. The physical aerodynamics and noise may have caused these variations (i.e., jitters). Along the trajectories, we visualize instances of drones at two different time ticks. Circled letters represent drones, where 'L' means the leader, and A~E means follower 1~5. The symbol is followed by a number that represents the time tick of the instances. For instance, 'L1 and A1~E1' represent the drones' positions at the time tick 1 while 'L2 and A2~E2' are positions of the same drones at the time tick 2. The red transparent lines between drones visualize a group of drones at the same time tick.

Result. Figure 10-(a) shows partial traces of the drones using SWARMBUG's fix "*infl_radius* = 0.6" (from the original value 0.3), which safely finishes the mission without crashing. Figure 10-(b) shows a picture of the physical experiment, while safely passing the obstacle (the box behind the drones). With the SWARMBUG's fix, drones maintain a sufficient safe distance. A video of this physical experiment is available on [82].

Finding a Fix without SWARMBUG. To provide a comparison point for the quality of the fix generated by SWARMBUG, we conduct a small additional experiment that tries to come up with a fix by manually changing the parameters without SWARMBUG. First of all, it would take a lot of time to pick the right configuration variable for the fix (i.e., *infl_radius*), without any guidances such as DCC and MSE values used in SWARMBUG. Even if we assume that the desired variable, *infl_radius*, is chosen, finding a good value for the fix is difficult. Assume that 0.4 is chosen (the original value is 0.3). The fix is tested by running the simulations 200 times that are all successfully finished without any crashes.

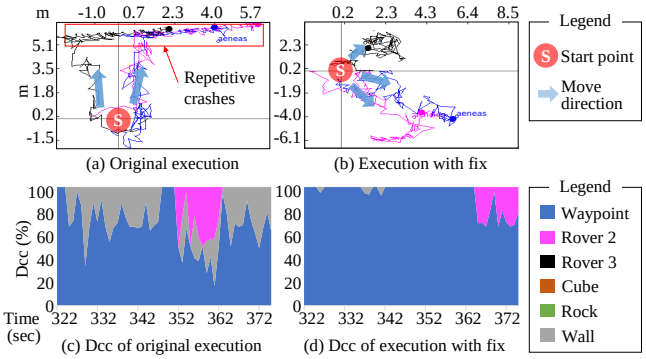
To this end, we run a physical experiment with the fix as shown in Figure 10-(c). Observe that Follower 2 (B2) and Follower 3 (C2) crash each other, meaning that while it passes the naive testing (200 times), the fix is not effective in real-world scenarios.

5.2.2 Debugging a Ground Vehicle Swarm. In this case study, we show how SWARMBUG is used to debug a ground vehicle swarm algorithm's configuration bug. We use a swarm algorithm [85]



(a) Trajectories (SWARMBUG Fix) (b) Picture from the physical experiment of (a) (c) Trajectories (Naïve Fix) (d) Picture from the physical experiment of (c)

Figure 10: Trajectories of 6 drones during our physical experiment.



(a) Original execution (b) Execution with fix (c) Dcc of original execution (d) Dcc of execution with fix

Figure 11: Applying SWARMBUG to Swarmathon

submitted to an annual robot competition funded by NASA: Swarmathon [57, 81]. The algorithm [85] took third place in the competition and was selected as the authors identified the bug with a few test runs. The goal of the algorithm is to leverage the swarm robots to gather resources spread throughout the map quickly.

During the mission, there is a buggy behavior that rovers keep crashing on the north border of the map and get stuck into the north-east corner, as shown in Figure 11-(a).

SWARMBUG identifies the Dcc trends as shown in Figure 11-(c) with seven environment configuration variables. Fluctuating Dcc values for the wall (i.e., the gray area) in the graph represent the crashes. SWARMBUG applies the *balancing* strategy based on the trend, identifying 14 potential fixes (from 38 swarm configuration variables). Among these fixes, "*M_PI_2 = rand()+pi()/2*" ranked the first (the original value for the variable is "*pi()/2*"). The fix yields the Dcc trend shown in Figure 11-(d). Most of the gray area is removed as the fix reduces the number of crashes. As shown in Figure 11-(b), the execution with the fix does not show the buggy behavior (e.g., drones stuck in the corner).

6 DISCUSSION

Overhead. During the operation, SWARMBUG runs a number of tests and conducts various analyses (e.g., computing Dcc and MSE values) on the collected data from the tests. Note that the analyses are done offline. We also instrument existing simulators to collect values for Dcc computation and the instrumentations incur less than 5% overhead at runtime.

Applicability of SWARMBUG's Fuzz Testing. While this paper focuses on finding and fixing configuration bugs, SWARMBUG's fuzz testing can find other types of bugs as well.

Specifically, while fuzz-testing the Adaptive Swarm, we find a bug in the algorithm that may rarely appear at runtime. That is, when a follower drone and the leader drone get very close to each other, the leader does not try to avoid the follower, leading to a crash. Our manual analysis shows that the leader drone's algorithm does *not consider follower drones as an object to avoid*. This is odd because follower drones have the logic to avoid the leader drone if they get too close. Our conversation with the developer confirmed that the developer assumed that the leader will always be far ahead of other drones and do not need to implement code to avoid a collision. Even testing three days without SWARMBUG does not reveal the bug. SWARMBUG's fuzz-testing identified such a scenario and exposed the defect, thanks to the guidance via Dcc and MSE values. We also validated this can happen in the real-world and the issue is confirmed by the author of the algorithm as well. More details can be found on our project page [82].

We also find that Dcc can be used to identify buggy logic in the swarm algorithm. Specifically, when we initially evaluate Howard's algorithm, we find that SWARMBUG could not find any possible fix. We investigate the Dcc values produced during the experiment further and notice that the observed Dcc values are extremely stable, except for slight variations observed in *obst_pot_c* just before the drone crashes. As we trace back to code related to *obst_pot_c*, we found that it detects the obstacle only after a crash happens. To properly avoid objects before it crashes, the algorithm should detect the object before it gets too close.

To fix this, we modify the algorithm so that it can detect objects early. After we patch the algorithm (can be found on our project page [82]), we conduct our evaluation on the algorithm again, and SWARMBUG successfully finds a possible fix as shown in Table 2.

Scalability and Usability of SWARMBUG. Our design is general and applicable to other swarm algorithms while it requires some engineering effort. Specifically, to support a new swarm algorithm, two tasks are required: (1) identifying configuration-variables (we provide a profiling tool for this in Section 4.1.1) and 5 thresholds (e.g., mission completion time, time-window, MSE thresholds), (2) instrumenting the algorithm to integrate SWARMBUG (e.g., changing 289 SLOC for Adaptive Swarm). In our evaluation, it took 10~18

hours (by a graduate student with moderate experience in drones) to complete the two tasks for an algorithm. The effort is non-trivial, but it is required one time for each algorithm. For example, besides the four evaluated algorithms, we have applied SWARMBUG to Swarmathon (see Section 5.2.2), taking about 10 hours (identifying 38 configuration-variables, the 5 thresholds, and changing 152 SLOC for the integration).

Future Directions. We envision future directions of our paper along two dimensions: empirical and technical. For the empirical aspect, applying SWARMBUG to more diverse swarm algorithms/systems (e.g., ground vehicle swarm) and more complicated scenarios (e.g., drones navigating a city landscape) and analyzing the cost and benefits of it can be the future work. Also, further analysis support to complete some of the semi-automated processes such as identifying key parameters used as inputs in SWARMBUG can be the future work as the technical aspect.

7 RELATED WORK

Testing Autonomous Robotics. Several testing methods are proposed [4, 31, 34] and studied [1] to solve and understand diverse challenges in testing autonomous robots. To evaluate the exploration of the system under test (SUT), coverage-driven verification (CDV) guides the testing process with an automated and systematic aspect; thus developers generate a broad range of test cases [4]. ASTAA [34] proposed an automated system specialized in stress and robustness testing and then discovered hundreds of bugs. Timperley *et al.* empirically studied and found that the majority of bugs in autonomous systems can be reproduced by software-based simulations [83]. Hildebrandt *et al.* integrated dynamic physical models of the robot to generate physically valid yet stressful test cases [31].

Alternatively, formal validation and verification are rigorously studied [35, 52] and used to prove properties of the testing programs such as correctness, functionality, and availability. Bensalem *et al.* developed a toolchain for specifying and formally modeling the functional level of robots [11], and Halder *et al.* implemented a system for checking the model of robots. Deeproad [23] validated inputs for testing autonomous driving systems.

Unlike previous studies, SWARMBUG aims to debug swarm algorithms, which is an order of magnitude more complex, by using the novel concept of the degree causal of contribution (DCC).

Testing/Debugging Approaches. Delta debugging [97] isolates the difference between a passing and a failing test case, by running mutated test cases and observing the execution results. BugEx [69] and Holmes [39] leverage a similar approach to understand the cause of bugs. In addition, Coz [19] introduces additional delays to infer possible optimization opportunities. LDX [46] perturbs program states at runtime to infer causality between system calls.

SWARMBUG uses a similar idea of mutating environment configuration variables to conduct behavior causal analysis. However, SWARMBUG handles swarm algorithms where inputs are essentially streams of data, while other techniques may need a non-trivial amount of modifications to handle such input data. SWARMBUG also leverages the DCC values to create a fuzz testing system.

Researchers leveraged random testing techniques (e.g., fuzzing) to continually improve the quality of test cases [47, 64, 84]. PySE [43] used a reinforcement learning-based approach to find a worst-case scenario. There are also model-based approaches inferring the actual program state [71, 74] or input types [89].

Automated Program Repair. There is a line of research focused on fixing buggy programs automatically [28, 30, 41, 48, 90]. In particular, [48] leverages a genetic programming approach [44] to repair a buggy program. [30, 90] proposes an automated program repair technique for programming assignments. While the previous works and SWARMBUG share the same goal of fixing a bug, SWARMBUG aims to fix configuration bugs in complex swarm algorithms running multiple robots. It fixes bugs by changing the swarm configuration variables' values, while the previous works change the program code to repair. QLOSE [21] leverages program distances to come up with solutions for program repairing. SemCluster [63] defines a new metric based on the input data space and uses the metric to cluster programs. SWARMBUG leverages DCC to guide the analysis and testing for swarm algorithms.

8 CONCLUSION

We proposed SWARMBUG, a debugging approach for resolving configuration bugs in swarm algorithms. SWARMBUG automatically identifies the causes of configuration bugs by creating new executions with mutated environment configuration variables. It compares the new executions with the original execution to find the causes of the bug. Then, given the cause, SWARMBUG applies four different strategies to fix the bug by mutating swarm configuration variables, resulting in fixes for the configuration bugs. Our evaluation shows that SWARMBUG is highly effective in finding fixes for diverse configuration bugs in swarm algorithms.

ACKNOWLEDGMENTS

We thank the anonymous referees for their constructive feedback. The authors gratefully acknowledge the support of NSF 1916499, 1908021, 1850392, 1853374, and 1924777. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

REFERENCES

[1] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. 2020. A Study on Challenges of Testing Robotic Systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 96–107. https://doi.org/10.1109/ICST46399.2020.00020

[2] Ruslan Agishev. 2019. *Adaptive Control of Swarm of Drones for Obstacle Avoidance*. Master's thesis. Skolkovo Institute of Science and Technology, Moscow, Russia.

[3] Javier Alonso-Mora, Stuart Baker, and Daniela Rus. 2015. Multi-robot navigation in formation via sequential convex programming. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 4634–4641. https://doi.org/10.1109/IROS.2015.7354037

[4] Dejanira Araiza-Illan, David Western, Anthony G Pipe, and Kerstin Eder. 2016. Systematic and realistic testing in simulation of control code for robots in collaborative human-robot interactions. In *Annual Conference Towards Autonomous Robotic Systems*. Springer, 20–32.

[5] Ardupilot. 2020. ArduCopter. https://ardupilot.org/copter/docs/introduction.html.

[6] H. Asama, M. Habib, I. Endo, K. Ozaki, A. Matsumoto, and Y. Ishida. 1991. Functional distribution among multiple mobile robots in an autonomous and decentralized robot system. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation*. IEEE Computer Society, Los Alamitos, CA, USA, 1921,1922,1923,1924,1925,1926. https://doi.org/10.1109/ROBOT.1991.131907

[7] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 237–250.

[8] Erkin Bahceci, Onur Soysal, and Erol Sahin. 2003. A review: Pattern formation and adaptation in multi-robot systems. *Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-03-43* (2003).

[9] Boldizsár Balázs, Gábor Vásárhelyi, and Tamás Vicsek. 2020. Adaptive leadership overcomes persistence–responsivity trade-off in flocking. *Journal of the Royal Society Interface* 17, 167 (2020), 20190853. https://doi.org/10.1098/rsif.2019.0853

[10] Jan Carlo Barca and Y. Ahmet Sekercioglu. 2013. Swarm robotics reviewed. *Robotica* 31, 3 (2013), 345–359. https://doi.org/10.1017/S026357471200032X

[11] Saddek Bensalem, Lavindra de Silva, Félix Ingrand, and Rongjie Yan. 2013. A verifiable and correct-by-construction controller for robot functional levels. *arXiv preprint arXiv:1309.0442* (2013).

[12] bitcraze. 2019. A local positioning system. https://www.bitcraze.io/products/loco-positioning-system/.

[13] bitcraze. 2020. A lightweight, open source flying development platform based on a nano quadcopter. https://www.bitcraze.io/products/crazyflie-2-1/.

[14] Alexandre Santos Brandão and Mário Sarcinelli-Filho. 2016. On the guidance of multiple uav using a centralized formation control scheme and delaunay triangulation. *Journal of Intelligent & Robotic Systems* 84, 1 (2016), 397–413. https://doi.org/10.1007/s10846-015-0300-5

[15] Gino Brunner. 2019. autonomous-drone. https://github.com/szebedy/autonomous-drone.

[16] Christian Howard. 2020. Algorithms developed to make drone swarm move together. https://github.com/choward1491/SwarmAlgorithms.

[17] Soon-Jo Chung, Aditya Avinash Paranjape, Philip Dames, Shaojie Shen, and Vijay Kumar. 2018. A Survey on Aerial Swarm Robotics. *IEEE Transactions on Robotics* 34, 4 (2018), 837–855. https://doi.org/10.1109/TRO.2018.2857475

[18] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/1273463.1273490

[19] Charlie Curtsinger and Emery D Berger. 2015. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 184–197. https://doi.org/10.1145/2815400.2815409

[20] Daniel Wollschlaeger. 2020. Analyzes shooting data with respect to group shape, precision, and accuracy. https://cran.r-project.org/web/packages/shotGroups/index.html.

[21] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*. Springer, 383–401. https://doi.org/10.1007/978-3-319-41540-6_21

[22] Celso De La Cruz and Ricardo Carelli. 2006. Dynamic modeling and centralized formation control of mobile robots. In *IECON 2006-32nd Annual Conference on IEEE Industrial Electronics*. IEEE, 3880–3885. https://doi.org/10.1109/IECON.2006.347299

[23] Ankush Desai, Shaz Qadeer, and Sanjit A Seshia. 2018. Programming safe robotics systems: Challenges and advances. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 103–119. https://doi.org/10.1007/978-3-030-03421-4_8

[24] Marco Dorigo, Dario Floreano, Luca Maria Gambardella, Francesco Mondada, Stefano Nolfi, Tarek Baaboura, Mauro Birattari, Michael Bonani, Manuele Brambilla, Arne Brutschy, et al. 2013. Swarmanoid: a novel concept for the study of heterogeneous robotic swarms. *IEEE Robotics & Automation Magazine* 20, 4 (2013), 60–71. https://doi.org/10.1109/MRA.2013.2252996

[25] Jan Dufek. 2019. Multi-UAV Cooperative Surveillance. https://github.com/jan-dufek/multi-uav-surveillance.

[26] Francesco. 2016. VRepRosQuadSwarm. https://github.com/merosss/VRepRosQuadSwarm.

[27] Kshitij Gajapure. 2018. Drone Simulation with realistic controls made using Unity. https://github.com/Kshitij08/Drone-Simulation.

[28] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[29] Volker Grabe, Heinrich H Bülthoff, and Paolo Robuffo Giordano. 2012. On-board velocity estimation and closed-loop control of a quadrotor UAV based on optical flow. In *2012 IEEE International Conference on Robotics and Automation*. IEEE, 491–497. https://doi.org/10.1109/ICRA.2012.6225328

[30] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices* 53, 4 (2018), 465–480. https://doi.org/10.1145/3296979.3192387

[31] Carl Hildebrandt, Sebastian Elbaum, Nicola Bezzo, and Matthew B Dwyer. 2020. Feasible and stressful trajectory generation for mobile robots. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 349–362. https://doi.org/10.1145/3395363.3397387

[32] Jun S Huang, Siqing Ma, Gao Li, Oliver W Yang, and Chang Shao. 2020. An Artificial Swan Formation Using the Finsler Measure in the Dynamic Window Control. *Int J Swarm Evol Comput* 9 (2020), 186.

[33] Ziyao Huang, Weiwei Wu, Feng Shan, Yuxin Bian, Kejie Lu, Zhenjiang Li, Jianping Wang, and Jin Wang. 2020. CoUAS: Enable Cooperation for Unmanned Aerial Systems. *ACM Transactions on Sensor Networks (TOSN)* 16, 3 (2020), 1–19. https://doi.org/10.1145/3388323

[34] Casidhe Hutchison, Milda Zizyte, Patrick E Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. 2018. Robustness testing of autonomy software. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 276–285. https://doi.org/10.1145/3183519.3183534

[35] Félix Ingrand. 2019. Recent trends in formal validation and verification of autonomous robots software. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*. IEEE, 321–328. https://doi.org/10.1109/IRC.2019.00059

[36] Florida Space Institute. 2020. EZ-RASSOR. https://github.com/FlaSpaceInst/EZ-RASSOR.

[37] Luca Iocchi, Daniele Nardi, and Massimiliano Salerno. 2000. Reactivity and deliberation: a survey on multi-robot systems. In *Workshop on Balancing Reactivity and Social Deliberation in Multi-Agent Systems*. Springer, 9–32. https://doi.org/10.1007/3-540-44568-4_2

[38] Alex Jinlei. 2018. Autonomous UAVs Swarm Mission. https://github.com/AlexJinlei/Autonomous_UAVs_Swarm_Mission.

[39] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal testing: understanding defects' root causes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 87–99. https://doi.org/10.1145/3377811.3380377

[40] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (London, England, UK) *(VEE '12)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/2151024.2151042

[41] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I. Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. 2016. Apex: Automatic Programming Assignment Error Explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 311–327. https://doi.org/10.1145/2983990.2984031

[42] kitz. 2021. Position controller instability at yaw angles close to 180 degrees. https://forum.bitcraze.io/viewtopic.php?t=4079.

[43] Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi. 2019. Pyse: Automatic worst-case test generation by reinforcement learning. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 136–147. https://doi.org/10.1109/ICST.2019.00023

[44] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA. https://doi.org/10.1007/BF00175355

[45] C Ronald Kube and Hong Zhang. 1993. Collective robotics: From social insects to robots. *Adaptive behavior* 2, 2 (1993), 189–218. https://doi.org/10.1177/105971239300200204

[46] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. LDX: Causality Inference by Lightweight Dual Execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 503–515. https://doi.org/10.1145/2872362.2872395

[47] Xuan-Bach D Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. 2019. SAFFRON: Adaptive grammar-based fuzzing for worst-case analysis. *ACM SIGSOFT Software Engineering Notes* 44, 4 (2019), 14–14. https://doi.org/10.1145/3364452.3364455

[48] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72. https://doi.org/10.1109/TSE.2011.104

[49] Eric Liu. 2019. Crazyflie cannot be stable when take off, it flipped onto the ground. https://github.com/USC-ACTLab/crazyswarm/issues/150.

[50] Yang Liu. 2019. Swarm formation sim. https://github.com/yangliu28/swarm_formation_sim.

[51] Yang Liu. 2020. Swarm robot ros sim. https://github.com/yangliu28/swarm_robot_ros_sim.

[52] Matt Luckcuck, Marie Farrell, Louise A Dennis, Clare Dixon, and Michael Fisher. 2019. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)* 52, 5 (2019), 1–41. https://doi.org/10.1145/3342355

[53] Li Ma, Weidong Bao, Xiaomin Zhu, Meng Wu, Yuan Wang, Yunxiang Ling, and Wen Zhou. 2020. O-Flocking: Optimized Flocking Model on Autonomous Navigation for Robotic Swarm. In *International Conference on Swarm Intelligence*. Springer, 628–639. https://doi.org/10.1007/978-3-030-53956-6_58

[54] N Harris McClamroch and Danwei Wang. 1987. Feedback stabilization and tracking of constrained robots. In *1987 American Control Conference*. IEEE, 464–469. https://doi.org/10.1109/9.1220

[55] Dejan Milutinović and Pedro Lima. 2006. Modeling and optimal centralized control of a large-size robotic population. *IEEE Transactions on Robotics* 22, 6

(2006), 1280–1285. https://doi.org/10.1109/TRO.2006.882941
[56] Yogeswaran Mohan and SG Ponnambalam. 2009. An extensive review of research in swarm robotics. In *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*. IEEE, 140–145. https://doi.org/10.1109/NABIC.2009.5393617
[57] Luong A Nguyen, Thomas L Harman, and Carol Fairchild. 2019. Swarmathon: a swarm robotics experiment for future space exploration. In *2019 IEEE International Symposium on Measurement and Control in Robotics (ISMCR)*. IEEE, B1–3. https://doi.org/10.1109/ISMCR47492.2019.8955661
[58] Reza Olfati-Saber. 2006. Flocking for multi-agent dynamic systems: Algorithms and theory. *IEEE Transactions on automatic control* 51, 3 (2006), 401–420. https://doi.org/10.1109/TAC.2005.864190
[59] Ori. 2020. DroneSimLab. https://github.com/orig74/DroneSimLab.
[60] Yash Vardhan Pant, Houssam Abbas, Rhudii A Quaye, and Rahul Mangharam. 2018. Fly-by-logic: control of multi-drone fleets with temporal logic objectives. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 186–197. https://doi.org/10.1109/ICCPS.2018.00026
[61] Jungwon Park. 2020. Trajectory generation and simulation for multi-agent swarm. https://github.com/qwerty35/swarm_simulator.git.
[62] Jungwon Park, Junha Kim, Inkyu Jang, and H Jin Kim. 2020. Efficient multi-agent trajectory planning with feasibility guarantee using relative bernstein polynomial. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 434–440. https://doi.org/10.1109/ICRA40945.2020.9197162
[63] David M Perry, Dohyeong Kim, Roopsha Samanta, and Xiangyu Zhang. 2019. SemCluster: clustering of imperative programming assignments based on quantitative semantic features. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 860–873. https://doi.org/10.1145/3314221.3314629
[64] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2155–2168. https://doi.org/10.1145/3133956.3134073
[65] Peyje. 2020. SWARMulator. https://github.com/Peyje/SWARMulator.
[66] James A Preiss, Wolfgang Honig, Gaurav S Sukhatme, and Nora Ayanian. 2017. Crazyswarm: A large nano-quadcopter swarm. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3299–3304. https://doi.org/10.1109/ICRA.2017.7989376
[67] Feng Qin, Cheng Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 135–148. https://doi.org/10.1109/MICRO.2006.29
[68] Nishanth Rao. 2019. ROS-Quadcopter-Simulation. https://github.com/NishanthARao/ROS-Quadcopter-Simulation.
[69] Jeremias Roßler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. 2012. Isolating failure causes through test case generation. In *Proceedings of the 2012 international symposium on software testing and analysis*. 309–319. https://doi.org/10.1145/2338965.2336790
[70] Dibyendu Roy, Arijit Chowdhury, Madhubanti Maitra, and Samar Bhattacharya. 2018. Multi-robot virtual structure switching and formation changing strategy in an unknown occluded environment. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 4854–4861. https://doi.org/10.1109/IROS.2018.8594438
[71] Charitha Saumya, Jinkyu Koo, Milind Kulkarni, and Saurabh Bagchi. 2019. XSTRESSOR: Automatic generation of large-scale worst-case test inputs by inferring path conditions. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 1–12. https://doi.org/10.1109/ICST.2019.00011
[72] Fabrizio Schiano and Paolo Robuffo Giordano. 2017. Bearing rigidity maintenance for formations of quadrotor UAVs. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 1467–1474. https://doi.org/10.1109/ICRA.2017.7989175
[73] Melanie Schranz, Martina Umlauft, Micha Sende, and Wilfried Elmenreich. 2020. Swarm Robotic Behaviors and Current Applications. *Frontiers in Robotics and AI* 7 (2020), 36. https://doi.org/10.3389/frobt.2020.00036
[74] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. Rescue: Crafting regular expression dos attacks. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 225–235. https://doi.org/10.1145/3238147.3238159
[75] Coati Software. 2020. Sourcetrail. https://www.sourcetrail.com/.
[76] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Information Systems Security*, R. Sekar and Arun K. Pujari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1
[77] Enrica Soria, Fabrizio Schiano, and Dario Floreano. 2020. SwarmLab: a Matlab Drone Swarm Simulator. (2020), 8005–8011. https://doi.org/10.1109/IROS45743.2020.9340854
[78] Siddharth Swaminathan, Mike Phillips, and Maxim Likhachev. 2015. Planning for multi-agent teams with leader switching. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 5403–5410. https://doi.org/10.1109/ICRA.2015.7139954
[79] swarm5. 2020. ESTKALMAN: State out of bounds, resetting. https://github.com/USC-ACTLab/crazyswarm/issues/259.
[80] swarm5. 2021. The motor has inconsistent performance. https://github.com/USC-ACTLab/crazyswarm/issues/289.
[81] Swarmathon. 2019. NASA Swarmathon. http://nasaswarmathon.com/.
[82] Swarmbug. 2021. Source Code Release. https://github.com/swarmbug/src.
[83] Christopher Steven Timperley, Afsoon Afzal, Deborah S Katz, Jam Marcos Hernandez, and Claire Le Goues. 2018. Crashing simulated planes is cheap: Can simulation detect robotics bugs early?. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 331–342. https://doi.org/10.1109/ICST.2018.00040
[84] Luca Della Toffola, Michael Pradel, and Thomas R Gross. 2018. Synthesizing programs that expose performance bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 314–326. https://doi.org/10.1145/3168830
[85] Jackson State University. 2018. Swarmathon Code of Team JSU. https://github.com/BCLab-UNM/Swarmathon-JSU-Public.
[86] Gábor Vásárhelyi, Csaba Virágh, Gergő Somorjai, Tamás Nepusz, Agoston E Eiben, and Tamás Vicsek. 2018. Optimized flocking of autonomous drones in confined environments. *Science Robotics* 3, 20 (2018). https://doi.org/10.1126/scirobotics.aat3536
[87] Tamas Vicsek. 2019. *Autonomous Mission Control of Drone Flocks*. Technical Report. EOTVOS Lorand Tudomanyegetem Budapest Hungary.
[88] Anthony De Bortoli Victor Delafontaine, Andrea Giordano. 2020. A drone swarm simulator written in Matlab. https://github.com/lis-epfl/swarmlab.
[89] Di Wang and Jan Hoffmann. 2019. Type-guided worst-case input generation. *Proc. ACM Program. Lang.* 3, POPL, Article 13 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290326
[90] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 481–495. https://doi.org/10.1145/3192366.3192384
[91] Shirley Wang, Nicholas Anselmo, Miller Garrett, Ryan Remias, Matthew Trivett, Anders Christoffersen, and Nicola Bezzo. 2020. Fly-Crash-Recover: A Sensor-based Reactive Framework for Online Collision Recovery of UAVs. In *2020 Systems and Information Engineering Design Symposium (SIEDS)*. IEEE, 1–6. https://doi.org/10.1109/SIEDS49339.2020.9106654
[92] William Warke. 2019. Crazyflie 2.1 rotating frantically and crashing at specific Yaw-Angle. https://github.com/USC-ACTLab/crazyswarm/issues/149.
[93] Frank Willeke. 2021. FlockModifier. https://github.com/FlaSpaceInst/EZ-RASSOR.
[94] Sean Wilson, Paul Glotfelter, Li Wang, Siddharth Mayya, Gennaro Notomista, Mark Mote, and Magnus Egerstedt. 2020. The robotarium: Globally impactful opportunities, challenges, and lessons learned in remote-access, distributed control of multirobot systems. *IEEE Control Systems Magazine* 40, 1 (2020), 26–44.
[95] Kun Xiao, Lan Ma, Shaochang Tan, Yirui Cong, and Xiangke Wang. 2020. Implementation of UAV Coordination Based on a Hierarchical Multi-UAV Simulation Platform. *arXiv preprint arXiv:2005.01125* (2020).
[96] Yxiao 2020. SwarmSim. https://github.com/yxiao1996/SwarmSim.
[97] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498
[98] Ganwen Zeng and Ahmad Hemami. 1997. An overview of robot force control. *Robotica* 15, 5 (1997), 473–482. https://doi.org/10.1017/S026357479700057X
[99] Hai Zhu, Jelle Juhl, Laura Ferranti, and Javier Alonso-Mora. 2019. Distributed Multi-Robot Formation Splitting and Merging in Dynamic Environments. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 9080–9086. https://doi.org/10.1109/ICRA.2019.8793765