2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) | 978-1-6654-0296-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE43902.2021.00119

Understanding Bounding Functions in Safety-Critical UAV Software

Xiaozhou Liang*, John Henry Burns*, Joseph Sanchez*, Karthik Dantu†, Lukasz Ziarek† and Yu David Liu*
*SUNY Binghamton, Binghamton, New York
Email: {xliang24,jburns11,jsanch49,davidL}@binghamton.edu
†SUNY Buffalo, Buffalo, New York
Email: {kdantu,lziarek}@buffalo.edu

***Abstract*—Unmanned Aerial Vehicles (UAVs) are an emerging computation platform known for their safety-critical need. In this paper, we conduct an empirical study on a widely used open-source UAV software framework, Paparazzi, with the goal of understanding the safety-critical concerns of UAV software from a bottom-up *developer-in-the-field* perspective. We set our focus on the use of Bounding Functions (BFs), the runtime checks injected by Paparazzi developers on the range of variables. Through an in-depth analysis on BFs in the Paparazzi autopilot software, we found a large number of them (109 instances) are used to bound safety-critical variables essential to the cyber-physical nature of the UAV, such as its thrust, its speed, and its sensor values. The novel contributions of this study are two fold. First, we take a static approach to classify all BF instances, presenting a novel *datatype-based* 5-category taxonomy with fine-grained insight on the role of BFs in ensuring the safety of UAV systems. Second, we dynamically evaluate the impact of the BF uses through a *differential* approach, establishing the UAV behavioral difference with and without BFs. The two-pronged static and dynamic approach together illuminates a rarely studied design space of safety-critical UAV software systems.**

***Index Terms*—unmanned aerial vehicles, bounding functions, safety**

I. Introduction

Unmanned aerial vehicles (UAVs) are an emerging platform with promising applications such as infrastructure inspection, precision agriculture, disaster search-and-rescue, and merchandise delivery. Traditionally designed as a robotics and embedded system with minimal software support, the software stack of UAVs in recent years has been significantly enriched, making them a "flying" computer system in the genuine sense. Beyond the excitement, the main hurdle against the broader adoption of this promising technology is their stringent requirement on safety: any crash of the UAV is not only a computer safety problem, but also a public safety hazard.

Even though the safety-critical nature of UAVs is universally recognized, there is no universal definition of what safety really means for UAVs. Broadly, any behavior that deviates from the "intended behavior" is a safety violation. Existing research [1]–[7] generally takes a "top-down" approach: an expert may provide a specification of the intended behavior, either through domain knowledge, or through the wisdom from the broader domains of cyber-physical systems (CPS) or robotics. Once the specification is given – whether in the form of invariants, constraints, pre-/post- conditions, or logic – the safety of a UAV system can be verified, monitored, or enforced.

A. An Empirical Perspective on UAV Software Safety

In this paper, we take a bottom-up approach to empirically study the safety of UAV software. In a nutshell, we choose to listen to the UAV software developers *in the field*, and reverse-engineer what they believe the most safety-critical software components are. Despite early UAV systems often being developed in a proprietary fashion, recent trends in open-source development for UAV systems present an opportunity. For example, the software framework that serves as the focus of our empirical study, Paparazzi [1] [8], is a popular open software (and hardware) ecosystem with more than a decade of development and numerous active contributors. It provides unified software support from autopilot to ground station, with diverse support for multi-copters, fixed-wing, helicopters and hybrid aircraft. If domain experts are the best source for understanding the "intended behavior," what can we learn about UAV safety from UAV software developers themselves?

We focus on how *Bounding Functions* (BF) are used in the Paparazzi autopilot software, arguably the most safety-critical components of the UAV software. A BF is a dynamic check inserted by programmers to ensure a variable – which we call a *Bounded Variable* (BV) – stays within a prescribed range. For example, variable `gv_z_ref` in Paparazzi's navigation guidance module is frequently bounded by a BF within the range $[\texttt{cur\_z} - \texttt{GC\_MAX\_Z\_DIFF}, \texttt{cur\_z} + \texttt{GC\_MAX\_Z\_DIFF}]$. Here, the bounded variable `gv_z_ref` represents the altitude the UAV is guided to for the next time interval; variable `cur_z` represents the current altitude of the UAV, and `GC_MAX_Z_DIFF` is a constant. Intuitively, this BF instance says that the UAV should not alter its altitude by `GC_MAX_Z_DIFF` or more within a time interval. This is aligned with our high-level understanding on UAV safety that an excessive change in altitude may jeopardize the stability of the UAV.

The premise of our approach is that the use of BFs is aligned with a UAV-specific safety concern. After all, the semantics of bounding a variable is akin to introducing an *invariant* over

[1]https://wiki.paparazziuav.org/

978-1-6654-0296-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00119

the variable: the application of the bounding function is a no-op if the variable is already in the range, or an assignment to the variable with the bound value otherwise. If we take the programmer's perspective, the need to bound a variable is aligned with her concern that an "out-of-range" variable may cause errors in the program.

We take a *two-pronged* approach in validating our premise. *Statically*, we identify all BF instances in the source code, and provide a detailed *datatype-based* taxonomy of the BF uses. We find a large number of BF uses indeed reflecting the safety-critical concerns of UAV systems (§ III). *Dynamically*, we perform a *differential* simulation to illustrate the impact of BF uses on UAV behavior. We find BF uses, and their lack of use, do have impact on the dynamic trace of safety-critical values of the UAV, from trajectory to pose, and hence cyber-physical behavior of the UAV (§ IV). We now elaborate these two contributions in more detail.

B. A Datatype-based Taxonomy

A novelty of our empirical study is that we classify the use of BFs based on the *datatype* of the BVs they intend to bound. In UAV software, a large number of values have primitive types such as `int` or `float`. A key insight gained in our exploration is that the BVs fall into a *small* set of well-known UAV parameters reflecting their cyber-physical nature, which we call *physical variables*. For example, we find a large number of `float`-type variables representing the 3 pose parameters that define the orientation of a UAV: the pitch, the roll, and the yaw. In other words, these variables carry higher-level semantics more than a floating point number. This insight recalls the classic programming abstraction of *abstract data type* (ADT) [9]: the `float` value above indeed logically encapsulates the floating number and a specification on what a pitch (or roll or yaw) parameter of a UAV should conform to. In this study, we classify our BF instances based on the logical *datatypes* of their corresponding BVs, as follows:

- **Trajectory Management (TM)** BF instances that provide safe navigation to the UAV, mainly bounding physical variables such as *position*, *distance*, and *heading*.
- **Sensor Management (SM)** BF instances that provide valid sensor readings, bounding physical variables directly related to *sensor values*.
- **Speed and Acceleration Management (SAM)** BF instances that ensure safe *speed* and *acceleration* to engine, bounding these two physical variables.
- **Engine Management (EM)** BF instances that provide safety to the engine by bounding 2 physical variables: the *thrust* and *throttle* of the engine.
- **Pose Management (PM)** BF instances that maintain safety for UAV orientation. These BF instances mainly bound 3 physical variables, *pitch*, *roll*, *yaw* of the UAV.

Within each class, we perform an in-depth analysis on how BFs are used in Paparazzi, defined as *use scenarios*. Taken the view of ADTs, each use scenario can be viewed as a *specification* — in the form of a BF — of that datatype. Overall, our novel datatype-based taxonomy can be summarized as "not all floating point values (or integers) are created equal." By refining them into datatypes, their logical role in UAV software starts to emerge. As it turns out, except BF instances used for defining generic algorithms (such as control and geometry), the remaining BF instances *all* fall into the 5 categories above. In other words, despite the large code base of UAV software and despite the numerous instances of BFs, UAV developers concentrate their efforts of performing dynamic checks on a small set of physical variables. This cannot be accidental: it is a conscious reminder that this small set of physical variables are likely to play a pivotal role in defining what being safety-critical means for UAV systems.

C. A Differential Simulation

To cross-validate whether our discovered BFs indeed have an impact on the correctness of UAV behavior, we perform a fine-grained simulation on the impact of BFs. We adopt a *differential* approach: for each instance of BF use, we perform one simulation over the original Paparazzi program, and the other over the same program except that the BF is removed. At its core, our approach can be viewed as a form of A/B testing. The interesting design question lies in how difference is defined. Our approach relies on analyzing the difference over *the traces of physical variables*, such as position traces (trajectories), pose traces, and speed traces. This approach, black-box in essence, is aligned with our intuition on the safety of UAV systems: if the UAV behaviors with the BF and without the BF are *observably* different through the lens of physics, then the BF is likely impacting the safety of the UAV.

D. Research Questions and Results

In this paper, we report the first empirical study on the bounding function uses in UAV software. It complements existing top-down approaches with a bottom-up perspective focusing on answering two research questions:

- **RQ1**: Can the BF instances be classified to logically reflect the use of safety-critical physical variables?
- **RQ2**: Do BFs have impact on the dynamic cyber-physical behavior of UAV software?

We identified 241 BF instances through analyzing Paparazzi's 2049 source files in autopilot software modules. We grouped 109 instances related to physical variables into the 5 categories (described earlier) most relevant to the safety of UAVs. Our dynamic differential analysis reveals that numerous BFs have observable impact on the trace of physical variables. More specifically, 30 out of 64 simulatable cases show difference in flight trajectory, pose, etc. This provides experimental justification for our BF-based approach: the use of BFs coincides with safety-critical behavior of UAVs. While conducting the trace-based analysis, we also uncovered a bug in Paparazzi, whose fix has been accepted.

Broadly and philosophically, our study is a quest for answers on what makes UAV software *safety-critical*. The top-down approach taken by verification frameworks and tools defines safety as *a priori* properties or invariants. To do so, one needs

to resort to domain experts to come up with the definitions of these properties or invariants first. Our bottom-up developer-in-the-field approach identifies the use of BFs with a call for attention from developers, and the deviation in the dynamic traces of physical variables with a cause of safety concern "as the developer's program says so." Overall, our approach and the existing approach complement each other: our approach discovers candidate invariants related to safety (but some may be deemed not by an "oracle" domain expert), whereas the existing approach focuses on invariants agreed upon *a priori* (but they may be incomplete in the eyes of the "oracle" domain expert). The two approaches together converge on revealing the elusive essence of safety in UAV software.

Overall, this paper makes the following contributions:

- the first "developer-in-the-field" empirical study on the safety-critical components of UAV software, based on bounding functions
- a datatype-based taxonomy on bounding function uses, focusing on physical variables
- a systematical differential analysis on the impact of BFs in UAV behavior through comparing and aligning traces of physical variables
- a tool `PBF-Detector` (Paparazzi Bounding Function Detector) for automatically identifying BF instances in a real-world code base with complex compilation schemes (decentralized compilation with 78 makefiles mixed with pre-processing code generation)

II. A Primer on UAV Flight Control

The most widely known UAVs fall into two categories: fixed-wing aircraft and rotary-wing aircraft. Fixed-wing aircraft are featured with special-shaped wings that can make use of forward airspeed to generate lift [10], while rotary-wing aircraft, also referred to as rotorcraft, use rotating wings called blades to fly [11].

A. Engine and Pose

The driving force produced by the engine is commonly referred to as *thrust* or *throttle*. Engine management is directly associated with the *speed* and the *acceleration* of the UAV. UAVs are rigid bodies operating in 3-D space. Therefore, their position can be represented by three numbers (x, y, z) in a 3-D coordinate system. Similarly, their *pose* (orientation) is represented by three angles (also known as Euler angles) in the 3-D coordinate system. These angles are *roll, pitch and yaw*. The pose is also referred as the *attitude*. An illustration of the three angles can be found in Figure 1. Fixed-wing aircraft vary their attitude by utilizing flight control surfaces. Rotorcraft vary the attitude by varying the rotational speeds of the motors spinning in opposite directions.

B. Navigation

Navigating a UAV is usually split into two steps - path planning and trajectory planning. Path planning is the step of taking the objectives of a fight task. Path planning is usually

Fig. 1: A Visualization of UAV control (Left: attitude angles; Center: their application on a fixed-wing aircraft; Right: their application on a quadrotor) [12]

application-dependent, written in the form of *flight plans* in Paparazzi. For example, a typical flight plan may include a step-by-step description of take-off, a circle navigation task, and then landing. The flight plan is translated into a *trajectory*. The trajectory is defined through a series of *waypoints*, positions in the 3-D space, with the Z-axis representing *altitude*. Trajectory planning takes the next waypoint to be visited and plans a thrust and pose to set the UAV to reach that waypoint. Given the required thrust and pose, the flight controller controls the actuators (such as engines) to achieve that thrust and pose. While in motion, the UAV points to a direction, which is called *heading*. A related concept is the *course*, the direction that the UAV moves toward. Due to conditions such as wind, heading and course are not always the same.

C. Paparazzi Flight Controller Software

Paparazzi UAV software suite is a collection of modules capable of flying on a variety of UAVs. It is highly configurable with various airframes, large suite of sensors, several controller algorithms as well as the ability to use the controller software in simulation and on real hardware.

The autopilot software is capable of integrating with several sensors, such as GPS, Inertial Measurement Unit (IMU), Sonar, and barometer. Sensor values are fed into the Inertial Navigation System (INS) that estimates position, speed, and acceleration of the UAV. Similarly, the Attitude and Heading Reference System (AHRS) performs attitude estimation. Together, the INS and the AHRS help the flight controller keep an estimate of the state of the UAV. This state is then used to control the UAV through the guidance and stabilization modules.

As is the case for all aerodynamic systems, control-theoretic algorithms are widely used to provide feedback control in UAVs' stability management and autonomous control. Two popular algorithms used by Paparazzi are Proportional Integral Derivative (PID) control [13] and Incremental Nonlinear Dynamic Inversion (INDI) [14].

III. Understanding Bounding Functions Statically

In this section, we describe our effort in understanding BF uses in Paparazzi through a detailed analysis on the source code, providing answers to **RQ1**. The centerpiece of this study is a *taxonomy* that classifies BF uses based on the physical variables they are applied to, in § III-C. Before we detail this

TABLE I: Bounding Functions in Paparazzi

Bound forms	Function names
double-ended bounds	Bound
	BoundInverted
	BoundWrapped
	VECT3_BOUND_CUBE
	VECT3_BOUND_BOX
	EULERS_BOUND_CUBE
	RATES_BOUND_CUBE
	RATES_BOUND_BOX
	Clip
absolute bounds	BoundAbs
	RATES_BOUND_BOX_ABS
	DeadBand
	ClipAbs
upper bounds	BoundUpper
normalization	FLOAT_ANGLE_NORMALIZE
	INT32_ANGLE_NORMALIZE
	INT32_COURSE_NORMALIZE
	NormRadAngle
special bounds	SATURATE_SPEED_TRIM_ACCEL

result, we start with a description of our taxonomy rationale in § III-A, and methodology in § III-B.

A. The Rationale of Classification

UAVs are cyber-physical systems that interact with the physical world. Their safety is defined with respect to this interaction, i.e., their behavior in the physical world. Our classification of BFs is based on this observation and thus derived from the *datatypes of physical variables* associated with the BFs. Our five-category taxonomy corresponds to the main functionalities of the UAV that define or impact interaction with the physical world. By organizing BF uses in this manner, we believe that this study will be useful for future UAV control software as they will still need to fundamentally interact with the physical world in the same manner: they will need to navigate (trajectory), control their navigation (speed and acceleration), understand their surroundings (sensors), understand their orientation with respect to their surroundings (pose), and manage their locomotion (motors). As our study shows, the vast majority of BFs in Paparazzi revolve around these five types of physical variables. This cannot be accidental: these five types of cyber-physical interactions are essential to the nature of UAV software.

B. Methodology

a) BF Identification: We have developed a compiler pass, implemented as a Clang plugin [2], to identify BF instances in the Paparazzi code base. Our plugin defines a baseline framework, `PBF-Detector`, for future research with advanced program analysis and optimization. Our analysis focuses on Paparazzi's autopilot software modules, in the `sw/airborne` directory, version v5.14.0_stable.

For our goal of identifying BFs, Paparazzi presents a unique advantage: a set of pre-defined BFs in the forms of C macros consistently used by Paparazzi developers. Our study focuses on the use of these macros, 19 in total as listed in Table I. These macros are manually identified by inspecting all `.h` files, and a macro qualifies if it bounds a variable within a given range. Some BFs are general, such as `Bound`, while others are more specific. Our Clang plugin parses C files to identify the 19 forms of BFs in the AST. One technical hurdle is that macros are expanded in Clang before the AST is generated. To address this, we have redefined 19 corresponding C functions to the macros in Table I. The Paparazzi source remains unchanged with a small number of exceptions that we documented at our project website (see URL in § VII).

b) Makefile-Aware Identification: A significant engineering challenge in analyzing Paparazzi's code base results from the complex compilation process inherent in Paparazzi. Unlike high-level applications where the compilation process is often a "one-off" process that reaches all files in all folders, embedded system software like Paparazzi must consider diverse configurations with complex customization and platform-dependent cross-compilation. Paparazzi adopts a hierarchical compilation with 78 C Makefiles distributed at various levels of the Paparazzi directories, and the dependencies between Makefile targets are complex. To further complicate the matter, many programs are generated on the fly during the compilation process with generators written in OCaml and Python.

Our compiler pass is *Makefile-aware*: we modified the decentralized Makefiles, and as a result, `PBF-Detector` can faithfully follow the same dependencies as in compilation. This not only allows us to reach all source code that can be reached by Paparazzi compilation, but also reach it in a semantic-aware manner: every name on the AST of every reachable file must have been defined (because the program compiles!). The `PBF-Detector` modification to handle hierarchical Makefiles in our compiler analysis was labor-intensive, but it is rewarding for building a toolchain for Paparazzi to integrate with Clang/LLVM.

c) BF Selection: In total, we identified 241 instances of BFs from autopilot program modules spanning 2049 files in 331K LOC of Paparazzi source code. We further cross-validated the number of instances through a text-based search. Among them, our study excludes instances not directly related to the safety of UAV software, which fall into 3 categories: (a) 71 BF instances in core control algorithms (PID/INDI). These BF instances are part of the algorithm design, such as PID and INDI; they are "generic" in nature and do not vary from a UAV implementation to a non-UAV implementation. As a standard robotics problem, bounding and tuning generic control parameters is an independent and well-studied problem [15]. It should be made clear that we only leave out generic control algorithm BF uses here: if a physical variable, say the roll value of the UAV, relies on the PID control and is bounded while interacting with the PID, it *is* included in our study. (b) 45 BF instances used for geometric transformation. These BFs occur as parts of the trigonometry-based algorithms solely related to geometry. For example, a common use is to normalize an angle within the range of (-2π , 2π). (c)

[2] https://clang.llvm.org/

15 BF instances in vision/image processing algorithms and 1 instance used for the remote control switch. For instance, BFs frequently occur for managing auto white balance, image refinement, sub-pixel resolution, and auto exposure. For any vision BF instances that impact control algorithms (e.g., optic-flow-based landing), we have included them into our study. Overall, our guiding principle here is to conservatively leave out BF instances unrelated to the safety-critical nature of UAV software, and when in doubt, an instance is included in our study. We have documented every BF instance for cross reference, including those we left out in the study on our project website.

There are two take-away messages from our BF selection process. On one hand, it shows some BF instances are not aligned with our intuition of safety-critical concerns. In that sense, these instances are the "false positives" to the premise of our empirical study. On the other hand, the more striking observation is that once the well-carved categories of (a)(b)(c) BF instances are removed, *every* remaining instance fits nicely with one of 5 categories intimately linked to the safety of UAVs, as we shall see next.

C. A Taxonomy of Bounding Function Uses

For the remaining 109 BF instances, we conducted an in-depth manual inspection, understanding the functionality of the program fragment each appears, and the purpose of each BF. As it turns out, all fit nicely into the 5-category taxonomy, which we present in Table II. In this table, observe that we further refine each category into a number of *use scenarios*. If each category is intuitively viewed as an ADT, each use scenario serves as a specified behavior of that ADT. In the rest of this section, we focus on trajectory management and sensor management as examples to demonstrate our approach. A description of all categories with the same level of detail can be found in a technical report at our project website.

1) Trajectory Management: To follow a trajectory, the UAV needs to follow waypoints, including turning occasionally (in the horizontal direction) and changing altitude (in the vertical direction). The physical variables related to trajectory management are *distance* and *heading* (change). We identified 11 instances of BFs applied for trajectory management, which we divided into 4 use scenarios.

a) Safe Homing: (a) Use Context: After performing the flight task, the UAV should go back to the ground station. (b) Datatype: distance (X and Y axis). (c) The Need for BFs: to avoid catastrophic consequences due to battery drain, a UAV (generally) should not fly too far way from the ground station. (d) Example: In this code snippet [16], the distance between the UAV waypoint and the home waypoint is computed, and bounded by variable `max_dist_from_home`, the maximum distance between them. (e) Occurrence: 3 instances.

b) Safe Altitude Change: (a) Use Context: UAV systems fly in a 3-D space; altitude change is a basic task. (b) Datatype: distance (Z axis). (c) The Need for BFs: a drastic change in altitude may affect the stability of the UAV. (d) Example: In this code snippet [17] , the altitude change between two iterations of the control loop is bounded by `GV_MAX_Z_DIFF`, the maximum distance between the previous waypoint and the current waypoint on the Z axis. (e) Occurrence: 1 instance.

TABLE II: Classification of Bounding Function Uses

Category	Use Scenario	Datatype	Occurrence
TM	Safe Leg Distance in Guidance	Distance (X and Y Axis)	4
	Safe Heading Change	Heading Change	3
	Safe Homing	Distance (X and Y Axis)	3
	Safe Altitude Change	Distance (Z Axis)	1
SM	Safe Sensor Fusion	Weight for Sensor Fusion	6
	Safe Sensor Reading Interval	Time Interval	1
	Safe Sensor Readings	Sensor Reading	1
SAM	Safe Acceleration Request as Engine Input	Acceleration	8
	Safe Acceleration for Navigation	Acceleration	4
	Safe Remote User Speed Input	Speed	3
	Safe Wind Speed	Speed	2
EM	Safe Motor Mixing	Thrust/Throttle	8
	Safe Landing	Thrust/Throttle	7
	Safe Motor Speed Change	RPM	5
	Collision Avoidance	Thrust/Throttle	1
PM	Safe Pose Change Rate	Pitch/Roll/Yaw	37
	Safe Pose Maintenance	Pitch/Roll/Yaw	12
	Safe Pose Change Time Interval	Pitch/Roll/Yaw Change Time Interval	2
	Safe Turn Coordination	Roll	1

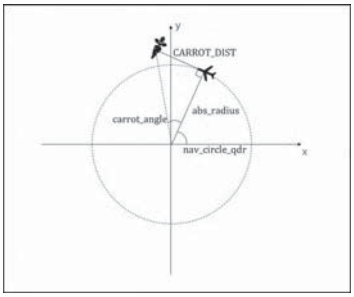


Fig. 2: Carrot-Based Guidance for Heading Change

c) Safe Heading Change in Guidance: (a) Use Context: Paparazzi follows the widely used *carrot-based* approach [18] for trajectory management: a virtual, continuously updated waypoint not far from the current position of the UAV to guide the next "leg" of movement of the UAV, similar to using a carrot to attract a mule to move forward. As shown in Figure 2, the carrot-based guidance implemented by Paparazzi for circle navigation assumes a constant distance between the current position of the UAV and the carrot, as `CARROT_DIST`. By

adjusting the `carrot_angle`, the UAV may change its heading. (b) Datatype: heading (change). (c) The Need for BFs: a drastic change in heading may affect the stability of the UAV, and affects the correctness of the circle trajectory. (d) Example: In Listing 1 which concerns circle navigation, the `carrot_angle` is bounded to the range of $[\frac{\pi}{16}, \frac{\pi}{4}]$. The rest of the variables are illustrated in Figure 2. (e) Occurrence: 3 instances.

```
void nav_circle(struct EnuCoor_i *wp_center, int32_t radius
    )
{
  ...

  // direction of rotation
  int8_t sign_radius = radius > 0 ? 1 : -1;
  // absolute radius
  int32_t abs_radius = abs(radius);
  // carrot_angle
  int32_t carrot_angle = ((CARROT_DIST << INT32_ANGLE_FRAC)
      / abs_radius);
  Bound(carrot_angle, (INT32_ANGLE_PI / 16),
     INT32_ANGLE_PI_4);
  carrot_angle = nav_circle_qdr - sign_radius *
     carrot_angle;

  ...
}
```

Listing 1: Safe Heading Change in Guidance [19]

d) Safe Leg Distance in Guidance: (a) Use Context: For linear trajectories that do not involve heading change, Paparazzi also uses carrot-based guidance. In this setting, the distance between the starting point of the leg and the carrot, which is called *leg distance*, is dynamically adjusted. (b) Datatype: distance (X and Y axis). (c) The Need for BFs: if the leg distance is set too long, the UAV may go "past" the waypoint of the target point. Deviating from the planned trajectory is a correctness concern. (d) Example: In this code snippet [20] which concerns route (i.e., linear) navigation, the `nav_leg_progress` is bounded to guarantee that the next leg of flight does not surpass the target waypoint. (e) Occurrence: 4 instances.

2) Sensor Management: As important components of a UAV, sensors play an irreplaceable role in UAV's state estimation, e.g., UAV's current attitude (pitch/roll/yaw). An accurate estimation based on sensor data is also critical for UAV safety. The physical variables related to sensor management are *sensor readings*, the *time interval* among readings, and the *weight* when multiple sensor readings are weighted. We identified instances of BFs applied for sensor management, which we divide into 3 use scenarios.

a) Safe Sensor Readings: (a) Use Context: The raw sensing data may be unreliable, either because the sensor is faulty, or because the reading may only reflect a transient state. (b) Datatype: sensor reading. (c) The Need for BFs: The need for bounding is sensor-specific. Take the current sensor for example. Due to overflow on high current spikes (fast electrical transients in current), the reading may be magnitudes higher than normal readings. This would impact battery estimation, crucial for estimating the remaining flight time. (d) Example: In this code snippet [21] , the current sensor keeps its readings in `electrical.current`, which is in turn bounded to a safe range $[-65000, 65000]$. (e) Occurrence: 1 instance.

b) Safe Sensor Reading Interval: (a) Use Context: In UAVs, sensors are continuously reading. In some scenarios, the time interval between different readings plays a crucial role in physical estimation. For example, as an application of Kalman filter [22], the UAV can use data from GPS and barometer at different time intervals to estimate its vertical position and velocity. (b) Datatype: time interval. (c) The Need for BFs: if there is a significant delay between two intervals, the estimation may be inaccurate, which in turn severely impacts the decision-making process of the UAV. (d) Example: In this code snippet [23] , the variable `dt` represents the time interval between two GPS readings. It is bounded into the range $[0.02, 2]$ seconds. The variable is used by Kalman filter (`alt_kalman`) for the estimation of the UAV's altitude and vertical speed. (e) Occurrence: 1 instance.

c) Safe Sensor Fusion: (a) Use Context: Complementary filter [24] combines sensor readings from the accelerometer and the gyroscope to estimate UAV attitude (pitch/roll/yaw). (b) Datatype: weight for sensor fusion (c) The Need for BFs: To ensure that data collected from both sensors are considered adequately, their proportions in attitude estimation need bounding in order to reach a balance between these two components. (d) Example: In Listing 2, `ahrs_fc.weight` computed at line 9 reflects the role of accelerometer plays in attitude estimation, which is influenced by `fabs(1.0 - g_meas_norm)`, the deviation between the measured gravitational acceleration and 1g. In the case of vibrations, large deviations from 1g may cause a decrement of the weight for the accelerometer data if bound is not introduced, ultimately causing the attitude estimate to drift [25]. Attitude estimation is critical for the safety of UAVs. In the aviation history, a catastrophe with the same root cause is Lion Air Flight 610, which was caused by incorrect angle-of-attack sensing (and consequent activation of the anti-stall software to repeatedly pitch the plane downward) [26]. (e) Occurrence: 6 instances.

```
void ahrs_fc_update_accel(struct FloatVect3 *accel, float
    dt)
{
  ...

  // compute ratio between measured gravitational
     acceleration and the standard value
  const float g_meas_norm = float_vect3_norm(&
     filtered_gravity_measurement) / 9.81;

  // compute the weight of accelerometer in attitude
     estimation
  ahrs_fc.weight = 1.0 - ahrs_fc.gravity_heuristic_factor *
      fabs(1.0 - g_meas_norm) / 10.0;

  Bound(ahrs_fc.weight, 0.15, 1.0);

  ...
}
```

Listing 2: An Example of Sensor Fusion [27]

IV. Understanding Bounding Functions Dynamically

In this section, we experimentally evaluate the impact of BFs on UAV behavior, answering **RQ2**. We start with a description of our rationale in § IV-A and on experiment setup in § IV-B, and the core results from differential simulation will be described in the rest of the section with a summary and several more detailed case studies.

A. The Rationale of Differential Simulation

As we stated earlier, UAVs are cyber-physical systems that interact with the physical world. In UAV software, the traces of UAV physical properties — pitch/roll/yaw, trajectory, or altitude as time series — are essential for capturing their observable behavior. When the removal of BFs leads to observable difference in the trace of these physical variables, it should be a concern for attention.

Our differential simulation aims at achieving two goals. First, it helps confirm that the BF instances indeed impact the dynamic physical behavior of UAVs. A premise with the "developer-in-the-field" approach is that we *trust* the experience and wisdom of the developers. From the perspective, the dynamic approach here serves as the *trust but verify* step: we would like to confirm BFs do make a difference in defining the physical behavior of UAVs. With that, answers to **RQ2** serve as an evidence of the significance of our taxonomy proposed for **RQ1**. Second, the dynamic approach also serves as a quantitative study of the safety-critical impact of BFs. It complements the qualitative study of our static (taxonomy) approach by answering *how much* impact BFs have on the safety of UAV software.

B. Experiment Setup

We use Paparazzi's built-in simulator for recording flight trajectories. We further use Paparazzi's log plotter to generate traces on real-time physical variables, such as speed, altitude, and roll-pitch-yaw values. The two complement each other, with the former useful for elucidating macro-level navigation patterns, and the latter useful for characterizing micro-level time-dependent physical behavior.

Among the 109 BF instances, we are able to conduct simulation for 64 of them. Some programs with BF instances require manual radio control (RC) inputs. We have developed a script to ensure RC inputs are programmably given, so that for repetitions of the same experiment, identical RC commands with identical timing are inputed. The not-simulatable cases fall into two categories. First, the compilation and execution of some program fragments are hardware-dependent, such as requiring camera or sensor support. The Paparazzi simulator does support physical simulation, but it does not include features such as optical flow (for cameras) and some low-level sensors. Second, some code fragments where BFs occur are experimental features that cannot be built with any compatible aircraft. For example, no existing aircraft in Paparazzi is compatible with the module `stabilization_float_euler`, so we cannot simulate any BF instances in that module.

TABLE III: Simulation Result Summary (S-Diff: Single-BF Simulation Different Results; M-Diff: Multi-BF Simulation Different Results; Same: No Difference in Results; Non-Sim: Not Simulatable)

Category	S-Diff	M-Diff	Same	Non-Sim	Total
TM	7	0	2	2	11
SM	3	0	4	1	8
SAM	5	4	8	0	17
EM	2	0	5	14	21
PM	3	6	15	28	52

TABLE IV: Selected Differential Analysis BF Instances

Label	File Name	BF Line Number
A	ahrs_float_cmpl	253
B	common_nav	135
C	nav_gls	150
D	nav_gls	181
E	nav_smooth	174
F	attitude_ref_saturate_naive	79, 82, 83, 84 (multi-BF)
G	guidance_h_ref	240, 241 (multi-BF)

For each simulatable BF instance, we perform two experiments: (1) a simulation of the autopilot with a pre-defined flight plan (see § II-B) where the code with the BF instance is called; (2) a simulation with the same flight plan with the BF is removed. We compute whether the traces from the two experiments are different, where *difference* is defined as the relative error in the trace values of physical variables (roll, pitch, yaw, thrust, etc.) from the two simulations above. We repeat each pair of simulations 5 times. The data across the 5 runs are averaged out with respect to timestamps.

It is noteworthy that when there is more than one BF instance in the same function, removing one may have no impact on the trajectory or physical variable traces, but removing multiple can. From now on, we refer to the experiments that involve the removal of multiple BFs in the same function at the same time as *multi-BF differential simulation*, and refer to the one-BF-a-time experiments as *single-BF differential simulation*. Multi-BF differential simulation is performed when single-BF differential simulation for each BF in a function does not show any difference.

C. Result Summary

The results from the experiments fall into 4 categories, which we summarize in Table III. If our simulation shows difference in a single-BF differential simulation, we classify the involved instance as "S-Diff". Otherwise, if difference is shown in a multi-BF differential simulation, we classify the involved instances as "M-Diff". The rest of simulatable instances are classified as "Same", and non-simulatable instances are classified as "Non-Sim". There is no overlap between the categories. For repeated experiments, we only mark an instance as "different" when 5 repeated experiments all show a difference exists: in physical simulation, small variations

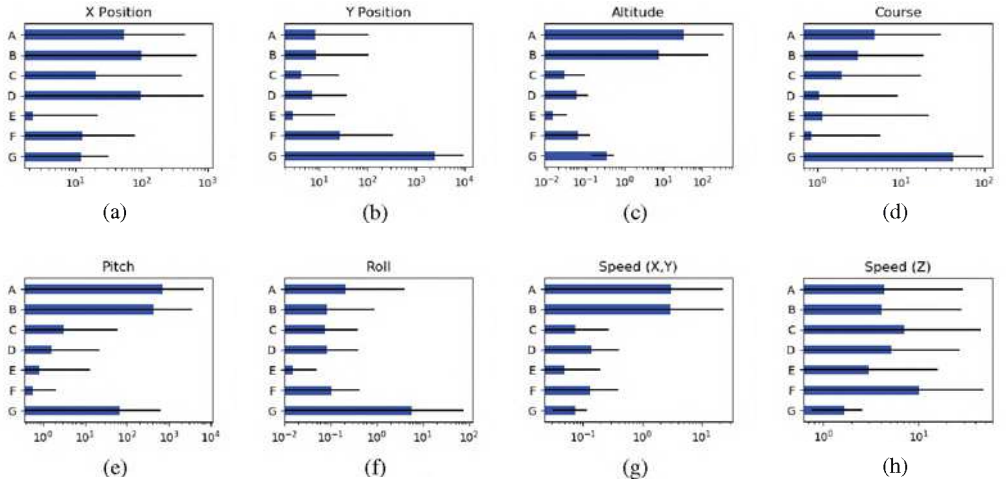


Fig. 3: Relative Errors in Differential Analysis (Each sub-figure represents a distinct physical variable. Each bar represents a BF case, whose height is the mean and the range line is the standard deviation. The label to the left of each bar indicates a BF instance, whose details are described in Table IV. Data is presented in log scale, where 10^0 (1) implies 100% relative error.)

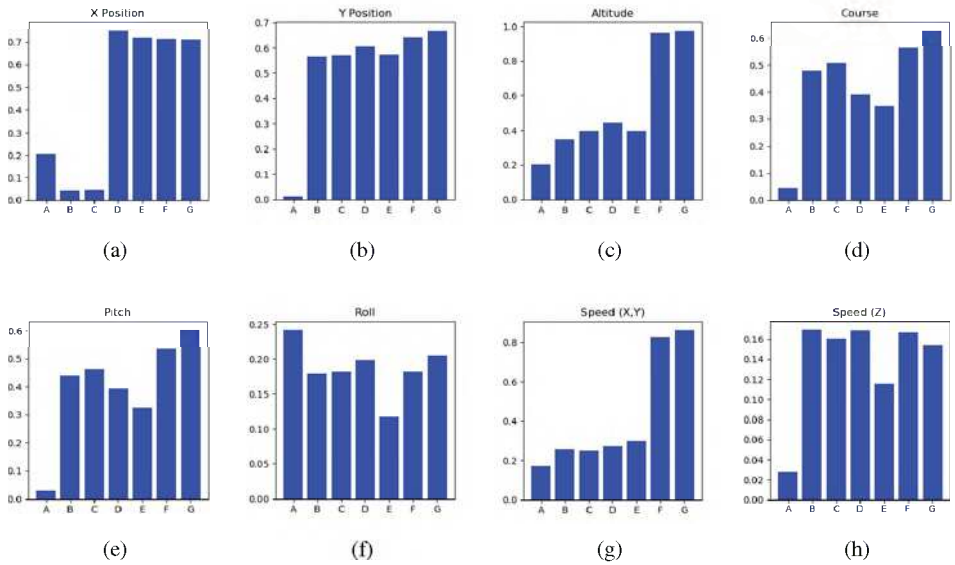


Fig. 4: Pearson's Correlation Coefficients (PCCs) in Differential Analysis (Each sub-figure represents a distinct physical variable. Each bar represents a BF case, whose height is the PCC. The label at the bottom indicates a BF instance, whose details are described in Table IV. A PCC value over 0.7 empirically indicates strong correlation.)

are common, so we wish to be conservative to make sure all repeated experiments agree.

As we can see in the S-Diff and M-Diff columns, nearly half of the instances we can simulate produce different results when comparing executions with or without BFs. In other words, the BFs indeed play an important role in safeguarding the correctness of programs and consequently the safety of UAVs.

In our differential analysis, we compute the averaged relative error between the measured physical variable value of the program with the BF, and the one without. We elide yaw data for brevity. Figure 3 shows the result for a subset of BF instances, whose details can be found in Table IV. The complete results are included in the repository. Three concrete observations can be made. *First*, BFs have non-equal impacts on physical variables. For example, we can observe that BF cases `A`, `B`, and `G` have large impacts on the majority of physical variables, whereas BF case `E` has a minor impact on nearly all variables. *Second*, the same BF instance may have different impacts on different physical variables. For example, BF case `A` has a larger impact on the Z axis of positioning

(altitude) (see Fig. 3c) than the Y axis (see Fig. 3b). As another example, the relative error of BF case G stands out in trajectory-related physical variables (see Fig. 3d for example) than speed-related variables. *Third*, the same physical variable may be impacted by different BF instances in different degrees. For example, speed is significantly impacted by BF cases A and B, but not others (see Fig. 3g).

To gain a finer-grained analysis, we further computed the similarity of the two traces in a timestamp-wise manner. Figure 4 shows the Pearson's Correlation Coefficients (PCCs) of the with-BF and without-BF traces. The most important observation is that PCC is rarely over 0.7, the golden standard for "strong" correlation. In other words, without BFs, a program would cause noticeable behavioral change to the UAV in a large number of BF instances, as manifested through location or pose or speed.

Take BF case E for example. Recall that in the earlier relative error figure, this BF has a small relative error; the PCC results however tell a different story: a timestamp-wise alignment of traces is poor for the majority of physical variables, especially altitude, pitch, row, and speed on the X/Y dimensions. In this example, the BF is used to bound the physical variable of ground speed. With its removal, the UAV not only has significant ground speed fluctuation, but also leads to functuations in other physical variables. The difference between relative error and PCC as metrics is that the latter is time-dependent. As a result, PCC can capture behavior differences in the presence of (time-dependent) fluctuation despite the "mean" remains stable, a goal the relative error cannot achieve.

Together, these experiments show that BFs do significantly impact UAV behavior. As physical variables play a pivotal role in safety-critical UAV behavior – from trajectory management to pose management and so on – our experiments demonstrate the importance of BFs in safety-critical UAV software.

In the rest of this section, we highlight 3 BF instances and their impact on preserving the UAV behavior.

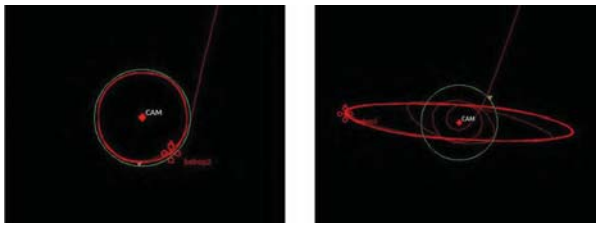
D. Case Study: Turning Angles

In § III-C1c, we discussed the bounded variable `carrot_angle` bounded within the range $[\frac{\pi}{16}, \frac{\pi}{4}]$ in function `nav_circle`, which is used by Paparazzi to perform a circle task. With the safeguard of the bounding function, the UAV circles around normally as is shown in Figure 5a, where the red actual trajectory fits nicely with the green desired trajectory.

However, if we remove the bounding function, as is shown in Figure 5b, the trajectory is irregular at the beginning and later follows a stable oval orbit. This deviation stems from the drastic angle variation `carrot_angle`.

E. Case Study: Takeoff Speed

As an example of multi-BF differential simulation, Listing 3 shows a code snippet where removing only one BF does not make a difference while removing both does. In this example, `sp` represents the vertical speed setpoint computed by the PID algorithm, and `incr` represents its deviation from the



(a) (b)

Fig. 5: A Case Study on Turning Angles (a) with-BF trajectory (b) without-BF trajectory

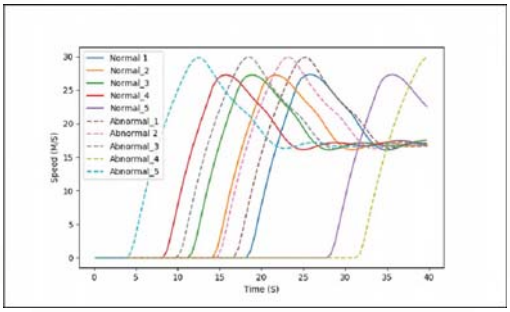


Fig. 6: Multi-BF Differential Simulation on Takeoff Speed

current speed setpoint `v_ctl_climb_setpoint`. `incr` is added to `v_ctl_climb_setpoint` in the end. If we only remove the BF on line 7, the excessive value would be bounded on line 10. Similarly, if we remove the latter, since the former has already bounded `sp`, the following `incr` is thus not likely to be excessive. However, when we remove both, `v_ctl_climb_setpoint` can grow by a sharp increment.

```
1  void v_ctl_altitude_loop(void)
2  {
3    ...
4
5    float sp = v_ctl_altitude_pgain * v_ctl_altitude_error +
       v_ctl_altitude_pre_climb ;
6
7    BoundAbs(sp, v_ctl_max_climb);
8
9    float incr = sp - v_ctl_climb_setpoint;
10   BoundAbs(incr, 2 * dt_navigation);
11   v_ctl_climb_setpoint += incr;
12 }
```

Listing 3: A Multi-BF Simulation Example [28]

Figure 6 shows the UAV speed when taking off based on the UAV's flight logs. The solid lines show the speed when both BFs are kept, while the dashed line show the speed when both are removed. In the first 40 seconds, the without-BF runs (named as "abnormal" in the Figure) reach a higher speed during take-off: observe that the dashed lines show a higher speed than those of the solid lines. This agrees with our source code inspection above.

F. Case Study: Navigation Progress

In §III-C1d, we discussed another bounded variable `nav_leg_progress` in function `nav_route`, and this

variable reflects the navigation progress which is bounded within the range $[0, \texttt{prog\_2}]$. As is shown in Figure 7a, an oval trajectory consists of two straight lines and two semi-circles. The function `nav_route` is called in the navigation task on both straight lines. If we remove the BF from variable `nav_leg_progress`, the UAV may "flee" and move in the opposite direction when approaching the waypoint where the straight routine begins, as is shown in Figure 7b.

The more intriguing question is why the UAV would change its behavior as radically as this. Let us have a close look at the source code on how `nav_leg_progress` is computed:

```
nav_leg_progress = (pos_diff.x * wp_diff.x
       + pos_diff.y * wp_diff.y) /
             nav_leg_length;
```

Here, `wp_diff` and `pos_diff` are two-dimensional variables representing the horizontal distance between two waypoints `p1` and `p2` (as shown in the figure) and between the UAV and the start waypoint `p1` respectively. The types of their members `x` and `y` are both *signed* integers. However, since `nav_leg_length` is an *unsigned* integer, the result computed within the parentheses must be implicitly converted to unsigned integer before divided by `nav_leg_length`. When the UAV is approaching the waypoint `p1`, the result computed within the parentheses happens to be a negative value whose most significant bit is set to 1, and thus it is interpreted as a large value after conversion to the unsigned integer. After being divided by `nav_leg_length`, the most significant bit becomes 0 and therefore, when the final result is converted back to the signed integer and assigned to `nav_leg_progress`, it is still a large positive value which goes far beyond the range $[0, \texttt{prog\_2}]$. It further impacts the computation of the position of navigation target and consequently the UAV flees eccentrically.

With a BF in place, `nav_leg_progress` is at least bounded within a range, so that a radically unexpected trajectory such as Figure 7b does not occur. However, note that always adjusting its value to the upper bound `prog_2` is not reasonable either: the variable should not have shown a completed progress before straight navigation begins.

In other words, the program contains a bug. To help fix this bug, we added an explicit type conversion for `nav_leg_length`:

```
nav_leg_progress = (pos_diff.x * wp_diff.x
       + pos_diff.y * wp_diff.y) /
         (int32_t)nav_leg_length;
```

After this bug fix, we repeated our simulation without the BF. As is shown in Figure 7c, the oval trajectory is preserved. However, as is analyzed in our discussion in § III-C1d, the carrot, namely the navigation target denoted as a yellow inverted triangle in the figure, exceeds the intended trajectory without the BF.

This case study is interesting for two reasons. First, the use of the BF indeed reflects the developer's concern that an unbounded variable may significantly alter the UAV behavior. Second, the developer appears to be unaware of the latent bug: the BF use somewhat masks the severity of the bug. Observe however, it is the use of the BF that led our attention to this code snippet, and it is the simulation of BF removal that helps us uncover the bug. We reported this bug to Paparazzi, and our bug fix has been accepted. The updated code has now been merged into Paparazzi's GitHub repository.

V. Threats to Validity

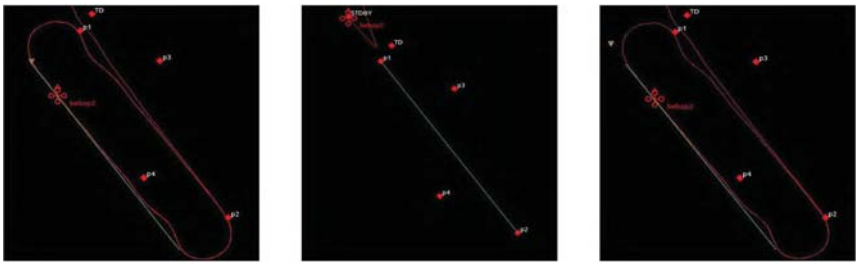
Our analysis is empirical in nature. We based our analysis on the BFs injected by the Paparazzi developers. We assume the correct amount of functions is leveraged to achieve safety-criticality. In this sense, a fundamental limitation of our approach is that it may only be as good as the programming skills of the developers. In reality, developers may miss BFs and may make mistakes. Incompleteness in enumerating all safety-critical scenarios is inherent to our approach.

UAVs, and embedded systems in general, are real-time systems driven by their onboard sensors. As such, achieving simulations that faithfully cover all flight scenarios is challenging. Our simulation environment can replay navigation commands and replay at a specific rate. However, due to timing of the control software, perfect reproducibility is impossible.

Paparazzi is an influential UAV framework, but not the only one. We believe the taxonomy of safety-critical use scenarios and the methodology of our differential analysis may transcend the specifics of Paparazzi, but the concrete findings of our study — such as the number of use scenarios within each caregory, and the dynamic impact of individual cases — may not be representative for all UAV frameworks.

VI. Related Work

Verification for safety-critical software is a well-established area, and perhaps the best example of the "top-down" approach for studying safety of UAV systems. Blanchet et al. [1] propose a static analyzer based on abstract interpretation to verify a large class of properties in safety-critical software. Miller et al. [2] apply NuSMV [3], a symbolic model checker, to the verification of a flight control system. Kloetzer and Belta [4] provide a fully automated framework to develop feedback controllers for a linear system given its linear temporal logic over a set of linear predicates in its state variables. Kress-Gazit et al. [5] propose a linear temporal logic (LTL) based framework to automatically generate a hybrid controller that guarantees correct robot function given a high-level task specification as well as a class of admissible environments. Yoo et al. [6] introduce a formal-methods-based process that supports development, verification, and safety analysis for the nuclear power plant's reactor protection system, and develop Computer-Aided Software Engineering (CASE) tools for nuclear engineers to apply formal methods to safety verification. Similarly, runtime verification of UAV software is also an actively researched topic. Moosbrugger et al. [7] develop a real-time, Realizable, Responsible, Unobtrusive Unit (R2U2) to monitor security properties and diagnose security threats of Unmanned Aerial Systems (UAS) during run time. Its



(a) (b) (c)

Fig. 7: A Case Study on Navigation Progress (a) with-BF trajectory (original) (b) without-BF trajectory (original): UAV flees (c) without-BF trajectory (fixed): carrot off

supervision scope covers the on-board components, as well as inputs from the ground control station.

Software engineering for self-adaptive cyber-physical systems is an active research direction, where UAVs are often cited as a compelling use scenario [29], [30]. Testing cyber-physical systems (e.g., [31]) and development tools (e.g., [32], [33]) is well explored. Another family of self-adaptive systems that have received attention in recent years is autonomous/self-driving vehicles, with results on bug characterization (e.g., [34]) and testing (e.g., [35], [36]). Programming languages are proposed for supporting energy awareness of UAVs [37] and context adaptation for UAVs [38]. Copilot [39] is a stream-based dataflow language to perform hard real-time monitoring over safety-critical control systems by sampling variables in programs and computing properties over the sampled values. SafetyScrum [40] is a software development methodology that relies on a notion of "safety debt" to incrementally track the safety status of safety-critical UAV systems in agile software development and maintenance.

Broadly speaking, our datatype-based classification can be related to programming language efforts that refine primitive types. For example, dimension types [41] are designed so that value 1 can either mean one meter or one kilometer, and misuse among them can be eliminated by the type system. As another example, Osprey [42] is a constraint-based type inference to automatically detect misuse of measurement units.

Fundamental to the growth of UAVs is their ability to fly autonomously and not require human control at all times. Most modern UAVs, from high-end fixed wing aircraft to hobby quadcopters, come equipped with flight controllers, such as in PixHawk [43]. These systems use well-studied algorithms such as extended Kalman filter estimation to fuse the sensor values into a pose, and well-studied controllers to achieve the set commands.

VII. Concluding Remarks

This paper describes a novel empirical study on the use of bounding functions in UAV autopilot software. Our study shows that the use of bounding functions coincides with use scenarios where safety concerns of UAVs are addressed by UAV software developers. Our differential simulation further shows that bounding functions play an important role in preserving the physical behavior of UAVs. To the best of our knowledge, this is the first systematic empirical in-field study on open-source UAV software frameworks.

Beneficiaries We envision our empirical study will be beneficial in the following ways. (1) For *UAV software developers*, our empirical study may serve as a reference point for systematically addressing safety concerns in future UAV development. UAVs are well known for their diverse hardware platforms, but the key safety-critical datatypes identified by this paper are likely to transcend the specifics of diverse platforms of UAVs. We show that despite the large code base, the BF instances revolve around a small set of physical variables, which future developers should pay particular attention to. (2) For *framework and language designers*, our datatype-based taxonomy may inspire new abstractions to generalize, modularize, and reason about UAV software systems, with the identified datatypes and their associated use scenarios serving as motivations for new language-based designs such as automated BF placement and enforcement. (3) For *researchers* interested in *automated analysis* for UAV software (e.g., through testing, debugging, and verification), our identified BFs and their differential simulation serve as a source for identifying new invariants, and as a litmus test on validating the coverage of their approaches. In addition, `PBF-Detector` can serve as a base system to facilitate Clang/LLVM-based development.

Artifacts In the repository [3], we provide the following artifacts: (a) the source code of `PBF-Detector` together with modified Paparazzi source (Makefiles); (b) a detailed documentation on each BF use; (c) all data of the simulation results, including log data, simulation screenshots, along with aircraft and flight plan files as test cases; (d) scripts for statistical analysis and for reproducing the results; (e) a report of the complete BF taxonomy.

Acknowledgments We thank Brian Grant for his participation in the early stage of this project. We thank Gautier Hattenberger for his help on the Paparazzi Forum. This project is sponsored by NSF Awards CNS-1823260, CNS-1823230, and SHF-1749539.

[3]https://github.com/SUNY-BU-Software-Systems-Research-Group/PaparazziBF

REFERENCES

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," *SIGPLAN Not.*, vol. 38, no. 5, p. 196–207, May 2003. [Online]. Available: https://doi.org/10.1145/780822.781153

[2] S. Miller, E. Anderson, L. Wagner, M. Whalen, and M. Heimdahl, "Formal verification of flight critical software," in *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2005, p. 6431.

[3] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: A new symbolic model verifier," in *International conference on computer aided verification*. Springer, 1999, pp. 495–499.

[4] M. Kloetzer and C. Belta, "A fully automated framework for control of linear systems from temporal logic specifications," *IEEE Transactions on Automatic Control*, vol. 53, no. 1, pp. 287–297, 2008.

[5] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE transactions on robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.

[6] J. Yoo, E. Jee, and S. Cha, "Formal modeling and verification of safety-critical software," *IEEE Software*, vol. 26, no. 3, pp. 42–49, 2009.

[7] P. Moosbrugger, K. Y. Rozier, and J. Schumann, "R2u2: Monitoring and diagnosis of security threats for unmanned aerial systems," *Form. Methods Syst. Des.*, vol. 51, no. 1, p. 31–61, Aug. 2017.

[8] G. Hattenberger, M. Bronz, and M. Gorraz, "Using the paparazzi uav system for scientific research," in *Proceedings of the International Micro Air Vehicle Conference and Competition 2014*, August 2014.

[9] B. Liskov and S. Zilles, "Programming with abstract data types," *ACM Sigplan Notices*, vol. 9, no. 4, pp. 50–59, 1974.

[10] D. Anderson and S. Eberhardt. (2015) How airplanes fly: A physical description of lift. [Online; accessed 06-March-2020]. [Online]. Available: http://www.aviation-history.com/theory/lift.htm

[11] S. May. (2017) What is a helicopter? [Online; accessed 06-March-2020]. [Online]. Available: https://www.nasa.gov/audience/forstudents/5-8/features/nasa-knows/what-is-a-helicopter-58.html

[12] Pir Arkam. (2020) How does a plane fly? [Online; accessed 9-May-2020]. [Online]. Available: https://rookieelectronics.com/the-aerodynamics-of-flight-how-does-a-plane-fly/

[13] M. Araki, "Pid control," in *CONTROL SYSTEMS, ROBOTICS AND AUTOMATION - Volume II: System Analysis and Control: Classical Approaches-II*, H. Unbehauen, Ed. EOLSS Publications, 2009, pp. 58–79. [Online]. Available: https://books.google.com/books?id=RF1xDAAAQBAJ

[14] E. J. J. Smeur, Q. Chu, and G. C. H. E. de Croon, "Adaptive incremental nonlinear dynamic inversion for attitude control of micro air vehicles," *Journal of Guidance, Control, and Dynamics*, vol. 39, no. 3, pp. 450–461, 2016.

[15] K. J. Åström and T. Hägglund, *PID controllers: theory, design, and tuning*. Instrument society of America Research Triangle Park, NC, 1995, vol. 2.

[16] "Function nav_move_waypoint: url at https://github.com/paparazzi/paparazzi/blob/master/sw/airborne/subsystems/navigation/common_nav.c."

[17] "Function gv_update_ref_from_zd_sp: url at https://github.com/paparazzi/paparazzi/blob/master/sw/airborne/firmwares/rotorcraft/guidance/guidance_v_ref.c."

[18] G. Conte, S. Duranti, and T. Merz, "Dynamic 3d path following for an autonomous helicopter," in *Proceedings of the 5th IFAC Symposium on Intelligent Autonomous Vehicles*, Oxford, UK, 2004, pp. 473–478.

[19] "Function nav_circle: url at https://github.com/paparazzi/paparazzi/blob/master/sw/airborne/firmwares/rotorcraft/navigation.c."

[20] "Function nav_route: url at https://github.com/paparazzi/paparazzi/blob/master/sw/airborne/firmwares/rotorcraft/navigation.c."

[21] "Function electrical_periodic: url at https://github.com/paparazzi/paparazzi/blob/363dec86938cd1090221ccd772fc6fae58ed89a2/sw/airborne/subsystems/electrical.c."

[22] R. E. Kalman, "A new approach to linear filtering and prediction problems," 1960.

[23] "Function ins_alt_float_update_gps: url at https://github.com/paparazzi/paparazzi/blob/master/sw/airborne/subsystems/ins/ins_alt_float.c."

[24] W. T. Higgins, "A comparison of complementary and kalman filtering," *IEEE Transactions on Aerospace and Electronic Systems*, vol. AES-11, no. 3, pp. 321–325, May 1975.

[25] Paparazzi Wiki. (2015) Vibration. [Online; accessed 10-February-2020]. [Online]. Available: https://wiki.paparazziuav.org/wiki/Vibration#Complementary_AHRS

[26] D. Shortell and J. Shelley, "Lion air crash investigators looking at two american companies associated with boeing 737 max sensor," *CNN*, Apr 2019. [Online]. Available: https://www.cnn.com/2019/04/04/us/boeing-sensor-investigation/index.html

[27] "Function ahrs_fc_update_accel: url at https://github.com/paparazzi/paparazzi/blob/master/sw/airborne/subsystems/ahrs/ahrs_float_cmpl.c."

[28] "Function v_ctl_altitude_loop: url at https://github.com/paparazzi/paparazzi/blob/master/sw/airborne/firmwares/fixedwing/guidance/energy_ctrl.c."

[29] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–26. [Online]. Available: https://doi.org/10.1007/978-3-642-02161-9_1

[30] R. Lemos, H. Giese, H. Müller, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, and J. Wuttke, *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*, 01 2013, pp. 1–32.

[31] J. Deshmukh, M. Horvat, X. Jin, R. Majumdar, and V. S. Prabhu, "Testing cyber-physical systems through bayesian optimization," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3126521

[32] S. A. Chowdhury, "Automatically finding bugs in commercial cyber-physical system development tool chains," ser. ICSE '18, 2018, p. 506–508.

[33] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, "Slemi: Equivalence modulo input (emi) based mutation of cps models for finding compiler bugs in simulink," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 335–346.

[34] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen, "A comprehensive study of autonomous vehicle bugs," in *ICSE'20*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 385–396.

[35] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 1016–1026.

[36] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu, "Deepbillboard: systematic physical-world testing of autonomous driving systems," in *ICSE'20*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 347–358.

[37] Y. D. Liu and L. Ziarek, "Toward energy-aware programming for unmanned aerial vehicles," in *3rd IEEE/ACM International Workshop on Software Engineering for Smart Cyber-Physical Systems, SEsCPS@ICSE 2017, Buenos Aires, Argentina, May 21, 2017*. IEEE, 2017, pp. 30–33.

[38] J. H. Burns, X. Liang, and Y. D. Liu, "Adaptive variables for declarative uav planning," in *The 12th International Workshop on Context-Oriented Programming and Advanced Modularity (COP)*, 2020.

[39] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: a hard real-time runtime monitor," in *International Conference on Runtime Verification*. Springer, 2010, pp. 345–359.

[40] J. Cleland-Huang and M. Vierhauser, "Discovering, analyzing, and managing safety stories in agile projects," in *2018 IEEE 26th International Requirements Engineering Conference (RE)*, 2018, pp. 262–273.

[41] A. Kennedy, "Dimension types," in *In 5th European Symp. on Programming, LNCS 788*. Springer-Verlag, 1994, pp. 348–362.

[42] L. Jiang and Z. Su, "Osprey: A practical type system for validating dimensional unit correctness of c programs," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 262–271. [Online]. Available: https://doi.org/10.1145/1134285.1134323

[43] PixHawk Team. (2020) Pixhawk flight controller. [Online; accessed 15-May-2020]. [Online]. Available: https://pixhawk.org/