Investigating Controller Evolution and Divergence through Mining and Mutation*

Balaji Balasubramaniam*, Hamid Bagheri†, Sebastian Elbaum‡, Justin Bradley†
*Department of Information Technology, Central Community College, Grand Island, NE, USA
bbalasubramaniam@cccneb.edu
†Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE, USA
hbagheri@cse.unl.edu, jbradley@cse.unl.edu
‡Department of Computer Science, University of Virginia, Charlottesville, VA, USA
selbaum@virginia.edu

***Abstract*—Successful cyber-physical system controllers evolve as they are refined, extended, and adapted to new systems and contexts. This evolution occurs in the controller design and also in its software implementation. Model-based design and controller synthesis can help to synchronize this evolution of design and software, but such synchronization is rarely complete as software tends to also evolve in response to elements rarely present in a control model, leading to mismatches between the control design and the software. In this paper we perform a first-of-its-kind study on the evolution of two popular open-source safety-critical autopilot control software – ArduPilot, and Paparazzi, to better understand how controllers evolve and the space of potential mismatches between control design and their software implementation. We then use that understanding to prototype a technique that can generate mutated versions of code to mimic evolution to assess its impact on a controller's behavior.**

We find that 1) control software evolves quickly and controllers are rewritten in their entirety over their lifetime, implying that the design, synthesis, and implementation of controllers should also support incremental evolution, 2) many software changes stem from an inherent mismatch between continuous physical models and their corresponding discrete software implementation, but also from the mishandling of exceptional conditions, and limitations and distinct data representation of the underlying computing architecture, 3) small code changes can have a dramatic effect in a controller's behavior, implying that further support is needed to bridge these mismatches as carefully verified model properties may not necessarily translate to its software implementation.

I. Introduction

Controllers, most typically feedback controllers, compute inputs to a system based on observations of the system state [1]. These controllers are typically represented as mathematical models and subsequently implemented in software and executed as periodic tasks in a computer system. Successful robotic, vehicle, and other controlled cyber-physical systems (CPS) evolve, as do their controllers. Conceptually, this evolution occurs at two distinct levels as shown in Figure 1. At the control design level, that evolution may occur on the mathematical representations or higher level models in the chosen representation (e.g., Simulink, MATLAB, Octave). At this level it is common to observe model changes meant to refine the control law as the logical conditions under which a system should operate are realized, as the assumptions or levels of abstraction of the model are refined, or the model is revised to fit another system.

* This work was supported in part by NSF awards #1638099 and #1853374.

At the software level of the controller we observe at least three types of changes. First, software changes that directly map to the same changes in the control design. These changes constitute the primary target for tools supporting model-based design [2], [3] or controller synthesis [4]. Second, software changes that are meant to complete pieces of the implementation that were not defined in the design, either because of the higher level of modeling abstraction that was employed, or because it was not cost-effective to define them at the design level. Third, changes driven by the need to integrate the software controller files with a larger software ecosystem that goes beyond the controller itself, or by software maintenance needs.

The frequency of each type of software change varies across systems. For selected safety-critical software with large development resources, most changes can occur at the design level and be automatically verified and transferred to code with high fidelity (as shown by the arrow in Figure 1 going from the design model to the partial implementation of that model in software). For most projects, however, many changes occur just in the software as the controller design concentrates on the key building blocks providing a partial model of the system. Furthermore, the design necessarily abstracts many of the computing elements and context that must then be implemented in software. Sometimes these software changes make it back to the model through some mechanism like an issue tracker (dotted arrow). Most often, however, implementation changes do not

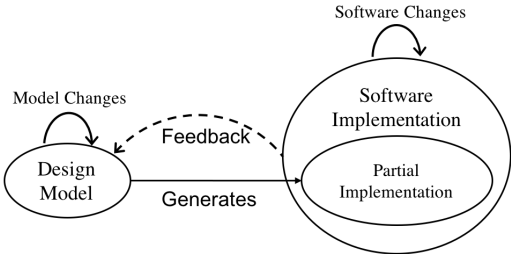

Fig. 1: Evolution of control design and software.

2642-9500/20/$31.00 ©2020 IEEE
DOI 10.1109/ICCPS48487.2020.00022

make it back or cannot be incorporated into the design model. This causes a divergence as the changes in the implementation of the control system may invalidate the properties so carefully proven at the design level.

In spite of the prevalence and impact of this software evolution, we, as a community, know very little about how the controllers that we so carefully craft change during their evolution, particularly at the software level. Our courses on control design, our textbooks, the tools we use, and the most promising research efforts *largely ignore the evolution of controllers*. We are distinctly aware of the inherent mismatches between the physical and software worlds (e.g., continuous vs. discrete, infinite vs. finite), but lack an understanding of how those mismatches manifest as the software changes [5], [6].

In this work, we shed light on this evolution by performing a first-of-its-kind case study exclusively on control software to show how and in what way it evolves. To do this, we examine 964 commits to the popular autopilot control software systems (i.e., ArduPilot [7] and Paparazzi UAV [8]) used on a wide range of Unmanned Air Systems (UAS). First, to provide a baseline for how much a controller evolves, we report metrics capturing to what extent controller-related evolution happens in these controllers. Our results show that controllers were entirely rewritten over nine times throughout their lifetime. **Implication**. This means as control software matures it may have little code in common with the original. Unless a tight correspondence between model and software is enforced over the many small changes (e.g., bug fixes, new features, etc.) made, the evolved model and control software may diverge drastically over its lifetime. This suggests that techniques such as control synthesis and model-based design techniques and tools must focus on accommodating this type of evolution.

We then identify 4 categories that capture the evolutionary changes resulting from inherent mismatches between system models and controllers, and their software implemented counterparts. Our results show that although some changes stem from an inherent mismatch between the continuous time/space physical model and its corresponding discrete software implementation, the majority of the changes were associated with handling exceptional conditions, and with the limitations and distinct data representation of the underlying computing architecture. **Implication**. This points to an unexplored opportunity for automated synthesis and software development techniques that can bridge these mismatches appearing during software evolution that may render carefully verified model properties invalid at the control software level.

Last, we explore the effects of software evolution in the performance of 3 controllers designed with Simulink. To do this, we developed a novel mutation technique that generates versions of the original code with mutated regions reflecting the categorized changes we observed in ArduPilot and Paparazzi UAV. The technique takes as input the code that is automatically generated by the Simulink toolset from carefully crafted control models, and can be configured to generate different types and numbers of mutants. The results demonstrate how *small and typical software changes can dramatically impact control performance.* **Implication**. A robustness measure, similar to traditional control theory robustness [9], can be developed to help control and software engineers improve controller designs as software complexity increases in modern CPSs.

To summarize, this paper makes the following contributions:

- We present an empirical study investigating control software to better understand its evolution
- We contribute a novel controller change categorization scheme and suite of controller-aware mutation operators derived from widely-used, safety-critical control software.
- We show how to exploit derived mutation operators that can map software changes directly to controller performance, paving the way for design of controllers robust to software changes.

II. Related Work

Most work at the intersection of software and control has examined the impacts of the disparity between the continuous mathematical models representing physical systems and controllers and the fundamentally discrete nature of computing [10]. Such research focuses on the effects of computation (e.g., quantization, delay) on the controller and seeks to find ways to incorporate them into controller design [1]. This is the substance of digital control theory [11].

However, the control community does not generally examine the role the software development process plays in impacting control design. But examining control software and its evolution could have far reaching impacts. For example, in the process of software maintenance, a year after the controller design, if a key calculation alters the precision by changing `fabs()` to `fabsf()`[1] does this impact system stability? Does a software change to limit stack size, a limitation of the computer architecture, cause a function call chain to fail, impacting controller performance? A study of control software evolution can provide insight into how these effects could be mitigated either in the control model or in the software evolution process.

This motivation has led to some work focusing on software and control systems. Feron has examined how to integrate proofs of important control system properties, such as stability, directly into software [12]. This can alert the software developer when sensitive code is being modified, and provide a mechanism for verification processes to assess correctness. But unless the annotation process is less costly, the sources of unsoundness controlled, and the tools well integrated into the developer's environment and workflow, such strategies will struggle to gain mainstream acceptance [13], [14].

In safety-critical systems, model-based design strategies ideally create a 1:1 correspondence between the model and the software [10], [15], [16]. This strategy has been included in the most recent revisions of DO-178C "Software Considerations in Airborne Systems and Equipment Certification" [17] and its supplements [18]. This is done by building models in MATLAB, Simulink, Stateflow, or other tools, verifying these models, and then autogenerating corresponding code. In this paradigm the

[1] `fabs` operates on type `double` while `fabsf` operates on type `float`.

code autogeneration tool must be certified to produce provably correct results. While this strategy links the model and software it may only exist in domain-specific applications [3], and may not link third-party software libraries, drivers, or other specialized pieces of code used in development of the system, or may be incomplete. Indeed, the vast majority of control systems onboard UASs - a safety critical system - are not developed using model-based design, but rather, use hand coded controllers such as ArduPilot.

The software engineering community has developed techniques to cope with the validation and verification of systems that includes control software (e.g., [19]–[23]), or their sound application to assist in self-adaptation [24]. Unfortunately, outside of highly regulated safety-critical systems, use of these strategies is limited due to high costs. This is particularly noticeable in the extremely active UAS industry where open source autopilots (e.g., ArduPilot [7], Paparazzi UAV [8], PX4 [25]) are used extensively on various types of hardware with contrastingly very light regulations and rigor in design and test processes.

Software evolution has been an active research topic for decades, and the realization that successful systems evolve and how it evolves has led to laws of software evolution [26], and a rich suite of techniques to understand, handle, and support changes associated with all the entities involved in the software development process [27]. The focus of the study in this paper is on analyzing the evolution of control software developed independently of model-based design [16], synthesis [20], or domain-specific annotations in code [12].

Following established practices [28], we analyzed two bodies of small, open-source, unregulated, safety-critical control software for which there are hundreds of available code changes recorded with commit level granularity. We have chosen these bodies of software for two key reasons. First, a large and increasing portion of critical software development with wide-reaching impacts is being developed in lightly controlled development and largely unregulated environments — such as the UAS industry. Second, understanding how the software evolves and reasoning about where mismatches with the model are likely to occur can pinpoint areas that future studies and techniques must target.

Finally, our mutation technique builds on a large body of work on mutation testing, which aims to evaluate the strength of a test suite in terms of the percentage of code versions with seeded code changes it can detect. Those versions are called mutants, and a test suite is said to kill a mutant when the presence of the change is detected by the test suite. There is a large number of mutation approaches available [29], [30], as well as several analyses to improve the effectiveness and efficiency of the mutation process [31], [32]. Unlike all prior work, we utilize mutation in our setting as a way to mimic the evolution we observed in widely-used safety-critical control software, and then to assess the impact of those mutations on controller operations.

III. Study

The following research questions will provide a foundation for understanding and characterizing control software evolution, and will underscore future tools that incorporate this knowledge into a framework for controller development:

RQ1: How does the software implementing a control system evolve? We seek to quantify the degree and nature of changes in control software in the absence of an explicit control model.

RQ2: To what degree can the changes in the control software be captured by a control model or constitute mismatches between the model and the software? We conservatively focus on characterizing the space of software changes that are *rarely* part of the control model.

RQ3: What are the impacts of software evolution in the performance of controllers? We are principally concerned with studying the mismatches that arise between control models and control software.

A. Analysis Artifacts

We required artifacts that included significant control software systems with many available versions reflecting their evolution. The first artifact is the popular ArduPilot [7], that provides a sophisticated control system for autopilot support that can operate on a variety of vehicles including airplanes, multirotors, helicopters, and boats[2]. It has over nine years of well maintained history, and its code base is accessible through a git repository [33] that stores the code changes committed by the developers since 2010. As of May 2019, the repository includes 448 contributors that have committed almost 38,000 changes. The latest version of ArduPilot contains approximately 250k lines of code (LOC)[3] in C/C++. We focus our analysis on the evolution of the core control files that provide coverage of functionality associated with position and attitude control. We

[2]The ArduPilot website reports that over one million vehicles use this code base, including companies like 3DR, PrecisionHawk, AgEagle, Insitu Boeing, Kespry, branches of the US military, and NASA among others.

[3]LOC - Lines of Code - is a count of lines in the text of source code excluding comment lines [34].

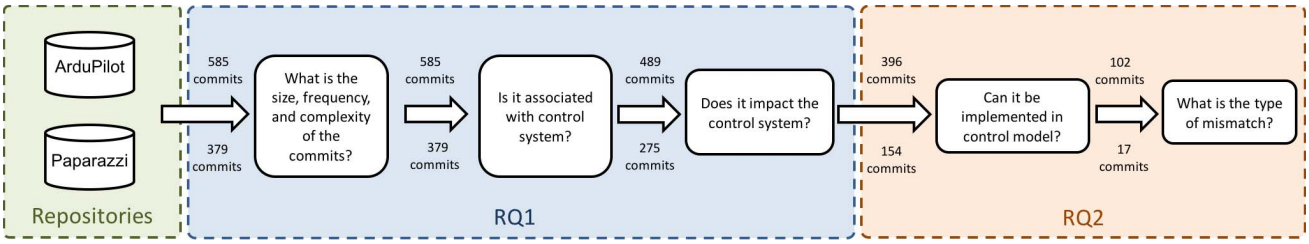

Fig. 2: Overview of the study analysis process to address the first two research questions.

analyzed 585 commits[4], where each commit included changes to at least one of the target files.

The second artifact is Paparazzi UAV [8], which has over 13 years of development history. Paparazzi UAV provides autopilot capabilities for fixed-wing, and rotorcraft vehicles. The code base is accessible through a git repository [35]. As of May 2019 there are 99 contributors and ~15,400 changes. The latest version of Paparazzi UAV contains approximately 212k LOC in C/C++. We again selected control files central to position and attitude control, and analyzed 379 commits, where each commit included changes to at least one of the target files.

Neither ArduPilot nor Paparazzi UAV have formal models of the controller, and do not practice complete model-based design. **These controllers are maintained and modified primarily at the code-level. This is common practice among small companies, researchers, and hobbyists in areas not subject to strict regulation and certification requirements.** Because ArduPilot and Paparazzi UAV provide safety-critical software to unmanned systems without a rigorous certification/verification process, they provide an excellent example of control software development that may be (at best) weakly linked to a mathematical model with provable guarantees.

B. Analysis Process

Figure 2 depicts a high-level overview of the study analysis process to address the first two research questions, which consists of a set of filtering and analysis steps. The process to answer **RQ1** starts by systematically querying the git repositories to quantify the degree of changes in terms of size, frequency, and people involved. To do this, we downloaded the latest repositories, and developed a set of scripts, in combination with the git client management tool Giteye [36], to collect the data.

To better understand the changes, we devised a procedure to identify commits that are most likely associated with changes to the control system. This procedure focused on the developers' comments and code changes, and was partially automated through a syntactic file search using common control keywords (e.g., control, derivative, error, feedforward, filter, frame, frequency, gain, integral, kalman, proportional) and also keywords specific to the target autopilot controller (e.g., acceleration, altitude, distance, pitch, roll, yaw, waypoint, speed, velocity). This process also took into consideration the online documentation explaining the roles of key configurable parameters and variable naming practices. Although we were biased in our selection toward conservatively including potentially impactful changes, our process is incomplete and a more rigorous strategy of what, precisely, constitutes a "control software" change is needed.

The resulting commits (489 for ArduPilot and 275 for Paparazzi UAV) were further analyzed to discriminate between changes deemed semantically equivalent such as those caused by documentation, refactoring, or abstraction to ease the maintenance of the software without directly impacting the functionality. For example, code found to be repeated may be extracted into a function call. This, theoretically, has no impact on the controller as it is purely a software maintenance change. This filtering left 396 ArduPilot and 154 Paparazzi UAV commits/changes impacting the controller directly.

The process to answer **RQ2** (cf. Figure 2) filtered the remaining commits by making a qualitative analysis to determine whether the change could have been handled in a control model. Since neither ArduPilot nor Paparazzi UAV have formal control models, our assessment consists of a conservative judgement of whether, if a mathematical model of the control system would be available, such a model could accommodate a given change. It is conservative in that, when in doubt, we assume that a control model could handle such a change. Our judgment of what can be modeled is based on traditional control theory [11], modern model-based design strategies [3], [16], and emerging research [37]. More specifically, unless the changes are tightly associated with: 1) the computing architecture, 2) the representations of data in that architecture, 3) the discretization of time and space to function in that architecture, or 4) the handling of anomalies due to that software functions, we assume it could be represented in a control model. When we determine that a control model would not typically include such a change, we assume it constitutes a mismatch between model and code that could have an impact on the system behavior. We then proceed to classify each change into one of four categories that emerged as we analyzed these mismatches and grouped them according to their characteristics, defined in Table I. This classification procedure was costly, with some changes requiring minutes and others requiring hours and the participation of multiple authors. Furthermore, this classification process was iterative as new mismatches emerged that either did not fit existing categories or fit multiple ones.

To address **RQ3**, we introduce a technique and corresponding tool we have developed to help mimic and analyze the software controller evolution and its impacts. Mutation analysis is a practice used to assess the robustness of software test suites apropos small, isolated changes in the software [29]. The tool, an overview of which is shown in Figure 5, generates code from Simulink models, mutates the code, compiles it, executes a test suite, and compares the output to the output of the original design. Our test suite for each mutant is a step response characterized by key control design quantities. The tool mutates the generated code according to our categories shown in Table I. We are primarily concerned with studying the mismatches that occur between control models and control software. As a result, the tool is focused on mutating code generated from Simulink because: i) model-based design is an increasingly important methodology but the evolution of model-based designs and corresponding generated code needs more study, and ii) model-based design should maintain a 1:1 correspondence between the design and generated code, but many changes that *could* be made in the code may not be represented in the design.

Our mutation tool is capable of handling C and C++ programming languages. This is done by constructing the

[4]In git, a commit consists of one or more changed files identified as a single change unit by the developer and assigned a single identification.

TABLE I: Definitions and Examples of Categories for Mismatches Between Models and Software

<table>
<tr><th>Category</th><th>ArduPilot</th><th>Paparazzi UAV</th></tr>
<tr><td>Resource Attributes</td><td colspan="2">Definition: A software change resulting from features or limitations of the computing architecture, including software and hardware. Such changes are often intended to better fit or utilize existing resources such as memory, energy, or bandwidth.</td></tr>
<tr><td></td><td>This change stores variables in flash memory instead of static random access memory.
Commit id: 452749149fd4d3e910e6ed22a6f861d5862a4b0
Committers comment: convert AC_PID library to AP_Param
...
+const AP_Param::GroupInfo AC_PID::var_info[] PROGMEM={
+ AP_GROUPINFO("P", AC_PID, _kp),
+ AP_GROUPINFO("I", AC_PID, _ki),
+ AP_GROUPINFO("D", AC_PID, _kd),</td><td>The horizontal feedforward gain is defined as 0. This is later used for multiplication bit operation to determine the control command for horizontal guidance navigation. Bit representations of control variables cannot be represented in the control model.
Commit id: 5de51d35588fa0080db7b8416924a900b405b4e9
Committers comment: [guidance] fix IGAIN precision and add VGAIN based on #682 this may introduce too large horizontal guidance IGAIN in rotorcraft airframe files
...
+#ifndef GUIDANCE_H_VGAIN
+#define GUIDANCE_H_VGAIN 0
+#endif
guidance_h_cmd_earth.x = pd_x +
((guidance_h_vgain * guidance_h_speed_ref.x) >> 17) +
((guidance_h_again * guidance_h_accel_ref.x) >> 8);</td></tr>
<tr><td>Precision and Accuracy</td><td colspan="2">Definition: A software change that modifies a measured value or a numerical calculation in order to more closely mimic continuous mathematics. Such changes often consist of utilizing improved functions in advanced math libraries or newer sensor devices, and simply using types with more bits for representation.</td></tr>
<tr><td></td><td>Replaces fast_atan with atanf to improve accuracy and precision for calculating the target pitch angle.
Commit id: 872583f4412ade16a31e8b7bd0363c294a20d301
Committers comment:AC_AttitudeControl removed
↪ fast_atan
...
-_pitch_target = constrain_float(fast_atan(
↪ -accel_forward/(GRAVITY_MSS * 100))*
↪ (18000/M_PI_F),-lean_angle_max,lean_angle_max);
+_pitch_target = constrain_float(atanf(
↪ -accel_ forward/(GRAVITY_MSS * 100))*
↪ (18000/M_PI_F),-lean_angle_max,lean_angle_max);</td><td>Replaces int32 with float to improve accuracy and precision for calculating the angular rate set point.
Commit id: 0c95b9e26edaba085f210b41d0a8325b607d9ada
Committers comment: [rotorcraft] converted PI rate controller to floating point closes #1624
...
- struct Int32Rates stabilization_rate_sp;
+ struct FloatRates stabilization_rate_sp;</td></tr>
<tr><td>Time and Space Model</td><td colspan="2">Definition: A software change resulting from the intrinsic discrete nature of the computing system in representing time and space. Such changes often consist of handling the inherent mismatch between continuous and discrete paradigms in representing and manipulating time in the calculations of derivatives and integrals, in the manipulation of variables associated with the vehicle location or motion, or in governing the periodic execution of certain pieces of code (e.g., tasks).</td></tr>
<tr><td></td><td>This change alters the time representation from seconds to milliseconds to more frequently check the position controller activity.
Commit id: 88ec13b10d913d72cdb0b24ba2e1244e6ed37734
Committers comment: fix build
...
- if (dt > POSCONTROL_ACTIVE_TIMEOUT_SEC) {
+ if (dt > POSCONTROL_ACTIVE_TIMEOUT_MS*1.0e-3f) {</td><td>This change alters the execution frequency of the navigation task from 10 Hz to 16 Hz.
Commit id: 624ce9eea923bff55e3c913363e9b42fe9cd6aab
Committers comment: navigation function in guidance; frequency set at 16 Hz
...
- RunOnceEvery(50, nav_periodic_task_10Hz());
+ RunOnceEvery(32, nav_periodic_task());</td></tr>
<tr><td>Exception Handling</td><td colspan="2">Definition: A software change resulting from the handling of anomalous conditions that would otherwise result in computational failures. Such changes often consist in additional support for conditions to adhere to either mathematical laws (e.g., dividing by zero), or computational laws (e.g., unexpected input, seg fault, etc.).</td></tr>
<tr><td></td><td>This change checks whether the input variable to the PID controller is infinite or undefined before using it to calculate the PID terms of the controller.
Commit id: ae77c18a1933dcb00eb9fc838872119b2250915c
Committers comment: Input to the PID controller is protect against NaN and INF.
...
+ // don't pass in inf or NaN
+ if (isfinite(input)){</td><td>This change prevents a divide by zero error by ensuring the variable is greater than zero before being used to calculate the navigation ratio for the vehicle controller.
Commit id: 7f91efa2854fee702a6601256dea5ff195e58f80
Committers comment: Fixed Error preventing AGR climb from working. Navigation would not blend.
...
+if (AGR_BLEND_START > AGR_BLEND_END &&
↪ AGR_BLEND_END > 0){
...
+nav_ratio = AGR_CLIMB_NAV_RATIO + (1 -
↪ AGR_CLIMB_NAV_RATIO)*(1 -
↪ (fabs (altitude_error) - GR_BLEND_END) /
↪ (AGR_BLEND_START - AGR_BLEND_END));</td></tr>
</table>

abstract syntax tree (AST) [38] of each program. We use Clang 3.8.2 and LLVM 4.0 for this and use the `MatchFinder` class of Clang to process the AST. To support repeatability, we have built this software infrastructure inside an operating-system-level virtualization, called docker (version 1.13.1).

To study the effects of the mutated software code, we used three different, complex system design models developed in Simulink: an automotive cruise control, a helicopter, and a

Boeing 747. The automotive cruise controller contains 14 Matlab blocks, and is available at [39]. The helicopter system contains 40 blocks, and is provided by MATLAB [40]. The third system is an airspeed controller for a Boeing 747 containing 465 blocks [41].

Mutation Tool. Figure 5 shows the overall architecture of the mutation tool, which contains three main phases. In the first phase, it generates the C code from the Simulink models and automatically derives AST for the generated code. It then parses the AST to identify the locations where the code could be mutated. Mutation templates, obtained from software abstractions of our mismatch categories in Table I, are used to identify where code can be mutated. The templates are constructed such that many locations in the code can be mutated by a single template in a variety of ways. The tool randomly chooses a location and applies the mutation.

In total, we have implemented 21 unique template mutators for the three categories of Precision and Accuracy, Exception Handling, and Time and Space Model as detailed in Table III. The Resource Attributes category in Table I is excluded due to the tight dependence on specific hardware configurations.

In the second phase, the tool compiles and executes the mutated code to obtain a step input response from the mutated code to compare against the original model. We quantify the control performance via 8 traditional control step response metrics: rise time, settling time, settling min, settling max, overshoot, undershoot, peak, and peak time [11].

Finally, in the third phase, we verify whether the mutated code has altered the performance of the controller. This was done by comparing the step response characteristics values with their counterparts obtained by running the original system.

C. Threats to validity

This study has shortcomings that may impact the validity of the findings. First, the scope of the study is limited to the software side of controller evolution. This choice was intentional and allowed us to quickly leverage readily available data while decoupling the evolution occuring in software from that which would occur in a model. Future studies of controller design evolution, controller design coupled with control software, and impact on system performance will provide a broader understanding of the topic.

Second, our study is focused on two control software systems. This choice was opportunistic in that Ardupilot and Paparazzi have been widely deployed, so findings in these code bases can still be valid for similar systems (e.g., LibrePilot [42], PX4 [25]). Likewise, even though the cost of analyzing hundreds of commits limited the scope of files studied, those files perform different controller tasks and were designed by different groups of developers. As a result, we anticipate these findings will also apply to other files designed by other developers. We also acknowledge that the granularity of changes we studied (i.e., commits) may not expose all code changes made by developers.

Third, our analysis had a quantitative aspect that is partially automated and highly reproducible, and a qualitative aspect that in many instances required us to make judgment calls. Such judgment calls are subject to many biases, which we tried to reduce by defining clear criteria for filtering and classification, by having multiple authors check different parts of the results, and by iterating and revisiting the results as anomalies emerged. We have prepared a website with the mutation docker and detailed data[5].

IV. RESULTS

In this section, we report and interpret the results of our study that we have conducted in the context of this work.

Answers to RQ1 – How Much do Controllers Evolve?

We quantify the evolution of the selected ArduPilot and Paparazzi UAV control files, and the results thereof are captured in Table II, summarizing the control software evolution across various files for each subject system. Specifically, it reports on initial and final LOC, # of commits, LOC changed, and people involved for each of the files of interest. Changes were made by 28 developers who changed 15,066 LOC over 964 commits throughout the lifetime of the files. The guiding principle in this analysis is to examine the evolution of control software, and as a result, throughout the presented results we focus on *changes* to the software which excludes the first commit representing the initial implementation.

The Growth metric, shown in row 8 of Table II, assesses how much the software grows over its lifetime. Growth is computed as $\frac{(X-Y)}{Y}\%$ where X is the number of lines of code, excluding comments, in the latest commit (row 4 in Table II) and Y is the number of lines of code (excluding comments) in the earliest commit (row 3 in Table II). Growth captures the net lines of code changed including changes stemming from model clarifications, new features, bug fixes, and software maintenance. As an example for this metric, `AC_PID.cpp` had 54 lines of code initially, and in the latest commit has 141 lines of code, a growth of 161%. The ArduPilot files have an average growth rate of 131% while the Paparazzi UAV files average growth rate is 198%, implying that the initial implementations for both systems required significant changes to complete them and refine them, and more generally that these control files, like any successful software, grow in complexity as they evolve. In some cases, like for `AP_Baro.cpp`, we notice a dramatic growth of almost ~600% to abstract common features, support more devices, and improve calibration. Other files like `AC_PosControl.cpp` exhibit a more stable development from the start with only ~10% growth.

Growth does not, however, capture the amount of change occurring in a file. To measure this, code churn is defined as the total number of lines of code changed (row 6 in Table II) [43]. For example, the code churn for `AC_AttitudeControl.cpp` is 3350 lines of code with an average of over 26 lines changed per each of its 127 commits. To further emphasize the seriousness of code churn for control software, we use a metric we call "Rewrite Rate", that captures how many times the original controller has been essentially rewritten from a software perspective. Rewrite Rate is computed as $\frac{Z}{Y}$, where

[5]https://nimbus.unl.edu/CE/controllerevolution.html

TABLE II: Overview of Control Software Evolution

	ArduPilot						Paparazzi UAV					
Filename(→)	AC_PID.cpp	AC_PosControl.cpp	AC_AttitudeControl.cpp	AC_WPNav.cpp	AP_Baro.cpp	Total	stabilization_attitude.c	stabilization_attitude_euler_int.c	stabilization_rate.c	guidance_h.c	baro_board.c	Total
Initial commit	1/28/12	2/14/14	2/14/14	4/13/13	6/27/12		10/19/06	07/26/09	02/10/09	02/10/09	08/21/10	
Latest commit	2/18/17	4/27/17	6/22/17	7/9/17	7/7/17		02/19/17	03/22/16	04/27/16	12/23/17	12/27/17	
Initial LOC	54	601	152	166	55	1028	135	89	36	126	77	463
Latest LOC	141	661	440	754	382	2378	323	195	150	546	168	1382
Commits	37	134	127	185	102	585	72	52	60	159	36	379
Code churn	463	1672	3350	3252	1043	9780	707	636	757	2859	327	5286
Developers	6	3	3	4	6	8	10	5	5	12	5	20
Growth (%)	161.1	10.0	189.5	354.2	594.6	131.3	139.3	119.1	316.7	333.3	118.2	198.5
Rewrite Rate	8.57	2.78	22.04	19.59	18.96	9.51	5.24	7.15	21.03	22.69	4.25	11.42

Z is the code churn and Y is LOC in the earliest commit. Note that high growth does not necessarily mean high churn. For instance, `AP_Baro.cpp` in the ArduPilot project exhibits the highest growth of all files, but `AC_AttitudeControl.cpp` shows the highest code churn. About half of the analyzed files represent Rewrite Rates of around or above 20, indicating those control files have *almost nothing in common* with the original versions. To give perspective, even the file with the lowest growth rate, `AC_PosControl.cpp`, has been rewritten almost three times.

For control designers this implies that a **controller implemented in software may significantly diverge from the original design without a correspondence to the model unless those ties are continuously enforced**. It also means that if a tight correspondence between the model and software is not enforced, a large amount of time must be spent updating the controller to correspond with the software (dashed arrow in Figure 1) or most likely the model will become obsolete along with its proven guarantees.

Answers to RQ2 – What Evolution Results from Model and Software Mismatches?

If control models and software evolve independently it is critical to understand what kind of changes prevent a 1:1 correspondence between them. We classified the 102 ArduPilot commits and the 17 Paparazzi UAV commits from the last stage of Figure 2 into the four categories defined in Table I. These categories represent the primary mismatches resulting from the incongruences between control models of the physical system and the computational paradigm of software implementation. In the right hand column are examples to clarify the types of changes in these categories. The mismatched commits and classifications distribution are shown in Figure 3. Each commit could have an arbitrary number of LOC changed, and hence a single commit may have multiple mismatches and be classified into more than one category.

Overall, from Figure 3, the distribution of mismatches is similar across ArduPilot and Paparazzi UAV. However, the quantity in ArduPilot is five times larger than Paparazzi UAV despite having smaller growth, rewrite rate, and fewer developers involved (cf. Table II). This is due, in part, to the larger number of commits that affect the control model in ArduPilot, and that, in Paparazzi UAV, some control elements were externalized into a separate configuration file (e.g., sampling periods) to isolate potential changes to the system.

Observing the categories, "*Precision & Accuracy*" was the biggest source of mismatches between model and software (cf. Figure 3), accounting for 41% of the ArduPilot and 40% of the Paparazzi UAV mismatches. **This implies developers prioritized improvements to the precision and accuracy of calculations to either 1) more closely mimic continuous mathematical assumptions of infinite precision, or, 2) prioritize improvement in computational system performance while sacrifing precision and accuracy.** We observe that some of these changes were not particulary complex (changing an `int` to `float`), while others involved utilizing special functions from a math software library. Still others, like switching `fabs` to `fabsf`, seem to sacrifice precision presumably to be consistent in the use of `float` to represent decimals and avoid unnecessary conversions potentially saving unnecessary computations at runtime. These mismatches were pervasive throughout the evolution of all files.

"*Time and Space Model*" mismatches are concerned with accounting for and tracking discrete time in control software. While we considered discretized space in the same category, which would be more prominent in control software incorporating, for example, a computer vision component, we did not observe any discretized space mismatches in this set of files. Ensuring consistency between periodic execution of a controller and associated computation of discrete derivative and integral equivalents is critical for correct control performance.

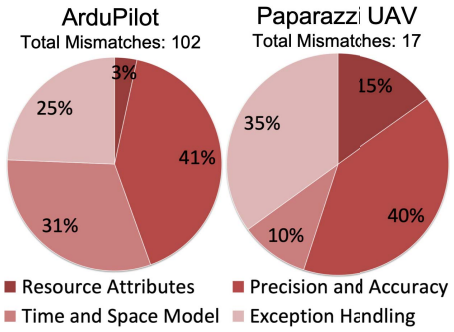

Fig. 3: Mismatches across ArduPilot and Paparazzi UAV.

We observed many changes that focused on improving this consistency in a programming language (C/C++) that does not natively provide semantic support for timing [44]. Most of these mismatches occur in the navigation/guidance (`AC_WPNav.cpp` and `guidance_h.c`), and position controller (`AC_PosControl.cpp`) portions of the controller software. Our results reporting on the number of changes involving timing provide further support for Lee's claims that timing in computation is a major obstacle to the development of combined cyber-physical models in which determinism is preserved [44]. Although many of these mismatches could be incorporated into the model by using MATLAB toolboxes such as "TrueTime" [45] or checked using other timing verification strategies like UPPAAL [46], these are often costly and continue to be underutilized in many development environments like the one we have studied.

Often overlooked by control designers are the undefined mathematical operations in engineered systems such as dividing by zero, or multiplying by ∞. In mathematical models these exceptions are built into the assumptions of continuous mathematics and are *implicitly* avoided. In software they must be *explicitly* avoided with lines of code protecting potentially undefined operations from causing the program to end prematurely or perform incorrectly. This exception handling also extends to software and computing architectural rules that must be obeyed (e.g., handling `NULL` pointers). The combined 36 total mismatches in this category (row 4 of Table IV in the Appendix) suggest that even software developers may take implicit assumptions about exception handling for granted. As the code evolves these exceptions are dealt with possibly in response to failed test cases or bug reports.

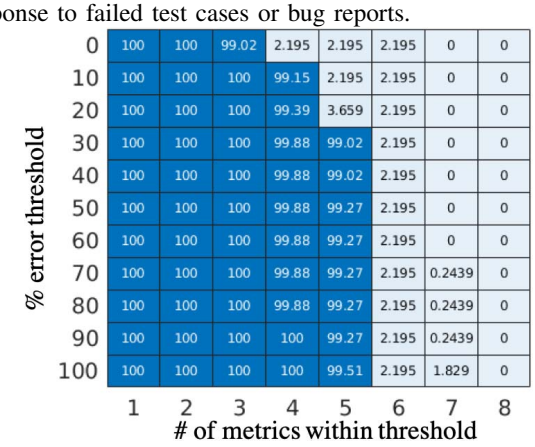

Fig. 4: Percentage of live mutants across the performance metrics for Boeing 747 airspeed controller.

Finally, computing architectural issues result in mismatches we classified as "*Resource Attribute*." Modern programming language abstractions have helped reduce these mismatches as compilers and libraries allow flexibility and optimizations without special programmer knowledge, and operating systems provide virtual memory and thread handling for executing processes. The small number of mismatches in this category is likely a result of the non-specialized hardware platform for ArduPilot and Paparazzi UAV. Had the control software required a specialized Digital Signal Processing (DSP) chip, or Graphical Processing Unit (GPU) we would have expected to see more mismatches in this category to accommodate those special-purpose computing architectures. Nevertheless, this category represents an important side-effect of software implementations of controllers - unless the control model explicitly captures the details of each target hardware architecture, programming language, 3^{rd}-party library or hardware driver, and operating system there will likely be mismatches between the model and implementation.

We further observe the number of mismatches per file is correlated with code churn in the file, with `AC_AttitudeControl.cpp` and `guidance_h.c` being the most affected. Still, `AC_AttitudeControl.cpp` seems to be the exception, suggesting that other factors (e.g., abstractions, refactoring) likely contributed to the evolution changes for `AC_AttitudeControl.cpp`. Generally, however, mismatch changes track proportionally with the total number of changes.

Comparing `AC_AttitudeControl.cpp` and `AC_WPNav.cpp` reveals that despite having roughly similar starting code size and total LOC changed in their lifetime, `AC_AttitudeControl.cpp` has only 30% as many mismatch changes. `AC_PosControl.cpp` and `AC_WPNav.cpp` have similar mismatch changes even though `AC_WPNav.cpp` was initially much smaller but grew to be twice as large and have much higher code churn. This is not surprising as `AC_WPNav.cpp` is the navigation code library that calculates the desired velocity, and acceleration to reach the destination. When the user provides the destination, `AC_WPNav.cpp` creates a flight path using spline waypoints and ensures the vehicle operates within the set range of acceleration, velocity, and speed, and determines whether the vehicle has reached its target. Such a software module is critical and difficult to develop correctly due in part to the many calculations requiring many vehicle and environmental parameters. Supporting this conclusion is a similar observation for `guidance_h.c` in Paparazzi UAV given its high relative mismatches, churn, and growth compared with other Paparazzi UAV files. This is perhaps the apex of joint model and software integration.

Answers to RQ3 – What are the Impacts of Software Changes on Control Performance?

Using our mutation tool (cf. Figure 5), we generated a total of 1539 mutants from the control models of the three subject systems: an automotive cruise control, a helicopter, and a Boeing 747. Table III provides details on the number of mutants, the number compiled, and the number executed for each system. The tool covered a considerable percentage of the code, altering more than one-third of the code in each of the systems. Compilation errors were the result of rare syntax mismatches. Occasionally, a mutant would fail to execute due to a runtime error. Overall, more than 97.6% of the mutants (1503 mutants out of the 1539 mutants) were successfully compiled and executed. This demonstrates the strength and robustness of our mutation tool that relies on various categories of mutation operators mined from the repositories of widely-used, safety-critical control software.

TABLE III: Mutation output details for all three systems

Mutation Operator	Mismatches	Cruise control			Helicopter			Boeing 747		
		# mutants	# compiled	# executed	# mutants	# compiled	# executed	# mutants	# compiled	# executed
int_T → uint32_T	Precision and Accuracy	2	2	2	6	6	6	3	3	3
int_T → real_T		2	1	1	6	1	1	3	0	0
uint32_T → int_T		0	0	0	0	0	0	3	3	3
real_T → int_T		1	1	1	3	3	3	46	46	46
d* → d*.0f		75	74	74	109	108	108	263	258	258
d* → d*.0		75	74	74	109	108	108	263	258	258
d*.0f or d*.0 → d*		85	85	84	125	125	124	144	144	142
double F() → (float) F()		0	0	0	0	0	0	37	37	37
if(rtIsNaN(X)) → if(!rtIsNaN(X))	Exception handling	0	0	0	3	3	2	3	3	3
if(rtIsInf(X)) → if(!rtIsInf(X))		0	0	0	0	0	0	2	2	2
insert if statment - check divide by 0		0	0	0	0	0	0	18	16	16
multiply denominator by zero		24	24	24	24	24	24	31	31	31
datatype of time is multiplied by 1000	Time and Space Model	6	5	5	18	17	17	20	17	17
datatype of time in ifstmt() is negated		4	4	3	4	4	3	0	0	0
variable of time is multiplied by 1000		2	2	2	2	2	2	3	3	3
variable of time in ifstmt() is negated		7	7	7	10	10	10	1	1	1
Total		283	279	277	416	408	406	840	822	820
Number of lines in file				279			449			1290
Number of unique mutated locations*				84			193			435
Total mutation coverage				30.10%			42.98%			33.72%

*Unique mutated location is the number of lines that got changed by the mutation tool.

We designed an oracle to classify the results as either "live" or "dead." Using the 8 step response quantities from the system (cf. Section III-B), we classify a system as "live" if k out of 8 step response metrics have an output value within a certain threshold percent of the original design, where k is varied from 1 to 8. If not, the mutant is considered "dead." In this paradigm "live" mutants represent a controller that does not exhibit performance variation within the threshold percent – indicating a robustness to software changes. We varied this threshold between 0% to 100% to capture the amount of variation in a step response that might be considered acceptable. Thresholds above 10% resulted in an inability to discriminate performance as all mutants would either be live or dead.

Figure 4 shows details for the Boeing 747 airspeed controller (other artifacts produced similar results). Each cell represents the percentage of live mutants with the number of metrics within the corresponding threshold percent. For example, consider Figure 4 (row 6, col 6), where 2.195% of mutants were live with six performance metrics within 50% error.

Recalling that "live" mutants in our tool represent controller mutants whose output does not differ from the model within the specified parameters. Figure 4 indicates the robustness of the specific controllers to software mutations—a software parallel to the robustness characterization of controller gains in traditional control theory [47]. More concretely, the airspeed controller in Figure 4, is one of the least robust to software changes given the sharp decline in live mutants for the number of metrics above 5 within a certain threshold.

More generally, our observations suggest that at a particular threshold all the mutants are either all live or all dead under the same inputs. **This shows the fragility of the control designs which generate significantly different responses with just a single change to the software**. Interestingly, only a few quantities in the system response were responsible for this dramatic change. For example, in the cruise control system, only the PeakTime quantity was not within the threshold limit. In that system, all mutants were dead, but only PeakTime was significantly impacted. For the helicopter system, only the SettlingMin quantity was highly impacted by our mutations but caused all the mutants to die. Our investigation further suggests that these two controllers are not robust to software changes and their inevitably accompanying evolution. For the Boeing 747, on the other hand, altitude was almost not affected by the mutations. Airspeed was only mildly affected. This suggests a controller that is robust to software changes and maintenance that are part of a healthy controller evolution.

The key impact of our tool is that much like a change in control gain can be directly mapped to a change in system response [47], this tool moves us toward directly mapping control software changes to a change in system response. This opens the door for studying how to design controllers that lead to robust software implementations.

V. Conclusions

A deeper understanding of the *types* and *quantity* of evolution that occur in controllers can help the control and software communities develop new models and development strategies to maintain the integrity of key properties verified in the model and/or software. We have directly studied this evolution in two dominant open-source control software suites, ArduPilot and Paparazzi UAV, used extensively in safety-critical UAS. Results show that control software evolves quickly, with controllers being entirely rewritten through their lifetime. We introduced categories capturing some of the inherent mismatches between typical control models and control software not previously identified. To facilitate more rapid study of this evolution we built a mutation tool that can rapidly change control code and compare its performance against the original designs. The

impact of this tool is the ability to map software changes directly to controller performance, thereby paving the way for studying the design of controllers robust to software changes.

REFERENCES

[1] B. Wittenmark, K. J. Åström, and K.-E. Årzén, "Computer control: An overview," *IFAC Professional Brief*, vol. 1, 2002.

[2] D. C. Schmidt, "Model-driven engineering," *COMPUTER-IEEE COMPUTER SOCIETY-*, vol. 39, no. 2, p. 25, 2006.

[3] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145–164, 2003.

[4] J. Daafouz, P. Riedinger, and C. Iung, "Stability analysis and control synthesis for switched systems: A switched Lyapunov function approach," *IEEE transactions on automatic control*, vol. 47, no. 11, pp. 1883–1887, 2002.

[5] L. V. Nguyen, K. A. Hoque, S. Bak, S. Drager, and T. T. Johnson, "Cyber-physical specification mismatches," *ACM Transactions on Cyber-Physical Systems*, vol. 2, no. 4, pp. 1–26, 2018.

[6] T. T. Johnson, S. Bak, and S. Drager, "Cyber-physical specification mismatch identification with dynamic analysis," in *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, 2015, pp. 208–217.

[7] ArduPilot, "ArduPilot Open Source Autopilot," 2018. [Online]. Available: http://ardupilot.org/

[8] PaparazziUAV, "PaparazziUAV," 2018. [Online]. Available: https://wiki.paparazziuav.org/wiki/Main_Page

[9] M. Green and D. J. Limebeer, *Linear robust control*. Courier Corporation, 2012.

[10] M. Zimmer, J. K. Hedrick, and E. A. Lee, "Ramifications of software implementation and deployment: A case study on yaw moment controller design," in *2015 American Control Conference (ACC)*, July 2015, pp. 2014–2019.

[11] G. F. Franklin, M. L. Workman, and D. Powell, *Digital Control of Dynamic Systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998.

[12] E. Feron, "From control systems to control software," *IEEE Control Systems Magazine*, vol. 30, no. 6, pp. 50–71, Dec. 2010.

[13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681.

[14] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 332–343.

[15] J. Porter, G. Karsai, P. Völgyesi, H. Nine, P. Humke, G. Hemingway, R. Thibodeaux, and J. Sztipanovits, "Towards Model-Based Integration of Tools and Techniques for Embedded Control System Design, Verification, and Implementation." in *MoDELS Workshops*. Springer, 2008, pp. 20–34.

[16] T. Erkkinen and B. Potter, "Model-based design for DO-178B with qualified tools," in *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 2009.

[17] L. Rierson, *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

[18] D. Cofer and S. Miller, "DO-333 certification case studies," in *NASA Formal Methods Symposium*. Springer, 2014, pp. 1–15.

[19] R. Majumdar, I. Saha, K. Ueda, and H. Yazarel, "Compositional equivalence checking for models and code of control systems," in *52nd IEEE Conference on Decision and Control*. IEEE, 2013, pp. 1564–1571.

[20] V. Braberman, N. D'Ippolito, N. Piterman, D. Sykes, and S. Uchitel, "Controller synthesis: From modelling to enactment," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1347–1350.

[21] R. Matinnejad, S. Nejati, L. Briand, T. Bruckmann, and C. Poull, "Search-based automated testing of continuous controllers: Framework, tool support, and case studies," *Information and Software Technology*, vol. 57, pp. 705–722, Jan. 2015.

[22] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, *HyTech: A model checker for hybrid systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 460–463.

[23] M. Kwiatkowska, G. Norman, and D. Parker, "Controller dependability analysis by probabilistic model checking," *Control Engineering Practice*, vol. 15, no. 11, pp. 1427 – 1434, 2007, special Issue on Manufacturing Plant Control: Challenges and Issues.

[24] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 299–310.

[25] pixhawk, "Pixhawk Flight Controller Hardware Project," 2018. [Online]. Available: https://pixhawk.org/start

[26] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sept 1980.

[27] T. Mens and S. Demeyer, *Software Evolution*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[28] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger, *Analysing Software Repositories to Understand Software Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 37–67.

[29] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[30] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 654–665.

[31] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 275 – 378.

[32] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque, "Assessing and improving the mutation testing practice of pit," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 430–435.

[33] Ardupilot development team, "Ardupilot git repository," 2018, accessed: 2018-1-11. [Online]. Available: https://github.com/ArduPilot/ardupilot

[34] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A SLOC counting standard," in *Cocomo Ii Forum*, vol. 2007, 2007, pp. 1–16.

[35] Paparazzi development team, "Paparazzi git repository," 2018, accessed: 2018-1-11. [Online]. Available: https://github.com/paparazzi/paparazzi

[36] Giteye team, "Giteye," 2019. [Online]. Available: https://www.collab.net/products/giteye

[37] S. A. Chowdhury, S. Mohian, S. Mehra, S. Gawsane, T. T. Johnson, and C. Csallner, "Automatically finding bugs in a commercial cyber-physical system development tool chain with slforge," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 981–992.

[38] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[39] U. of Michigan, "Cruise Control System Documentation," 2018, accessed: 2018-10-11. [Online]. Available: http://ctms.engin.umich.edu/CTMS/index.php?example=CruiseControl\&section=SimulinkModeling

[40] MathWorks, "Helicopter System Documentation," 2018, accessed: 2018-10-11. [Online]. Available: https://www.mathworks.com/help/control/examples/multi-loop-control-of-a-helicopter.html

[41] G. Campa, "Airlib," 2018, accessed: 2018-10-11. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/3019-airlib

[42] L. Pilot, "LibrePilot – Open – Collaborative – Free," 2018. [Online]. Available: https://www.librepilot.org/site/index.html

[43] G. A. Hall and J. C. Munson, "Software evolution: Code delta and code churn," *Journal of Systems and Software*, vol. 54, no. 2, pp. 111–118, 2000.

[44] E. A. Lee, "The past, present and future of cyber-physical systems: A focus on models," *Sensors*, vol. 15, no. 3, pp. 4837–4869, 2015.

[45] D. Henriksson, A. Cervin, and K.-E. Årzén, "TrueTime: Simulation of control loops under shared computer resources," *IFAC Proceedings Volumes*, vol. 35, no. 1, pp. 417–422, 2002.

[46] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.

[47] M. M. Seron, J. H. Braslavsky, and G. C. Goodwin, *Fundamental limitations in filtering and control*. Springer Science & Business Media, 2012.

APPENDIX

This appendix provides additional details of our analysis. Figure 5 shows a block diagram of our mutation tool discussed in Section III-B. Expanding on Figure 3, Table IV shows a more detailed view of the distribution of mismatches in each category for each file we analyzed.

Figure 6 captures the details of our mutation analysis for the Boeing 747 altitude controller. Each cell represents the percentage of live mutants with the number of metrics within the corresponding threshold percent. In contrast to the airspeed controller in Figure 4, the Boeing 747 altitude controller in Figure 6 is the most robust controller we tested as evidenced by the large percentage of live mutants for almost any threshold and number of metrics.

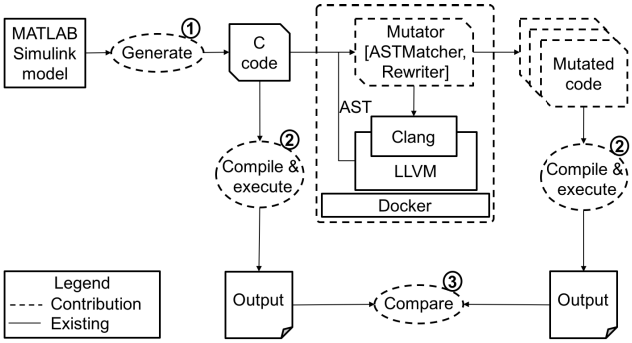

Fig. 5: Mutation Tool Architecture.

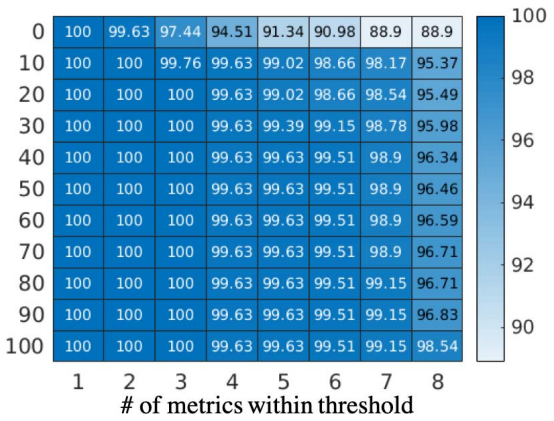

Fig. 6: Percentage of live mutants across the performance metrics for Boeing 747 altitude controller.

TABLE IV: Classification Results of Mismatches Between Models and Software

	ArduPilot						Paparazzi UAV					
Category(↓) / Filename (→)	AC_PID.cpp	AC_PosControl.cpp	AC_AttitudeControl.cpp	AC_WPNav.cpp	AP_Baro.cpp	Total	stabilization_attitude.c	stabilization_attitude_euler_int.c	stabilization_rate.c	guidance_h.c	baro_board.c	Total
Resource Attributes	1	2	0	1	0	4	0	0	0	3	0	3
Precision and Accuracy	7	15	7	13	9	49	0	1	2	5	0	8
Time and Space Model	2	15	3	11	8	37	0	0	0	1	1	2
Exception Handling	4	6	2	12	5	29	1	1	0	5	0	7
Total Commits With Mismatches	12	29	11	32	18	102	1	2	2	11	1	17