An Empirical Characterization of Software Bugs in Open-Source Cyber-Physical Systems

Fiorella Zampetti[1], Ritu Kapur[a], Massimiliano Di Penta[a], Sebastiano Panichella[b]

[a]*Department of Engineering, University of Sannio, Italy*
[b]*Zurich University of Applied Science, Switzerland*

Abstract

Background: Cyber-Physical Systems (CPSs) are systems in which software and hardware components interact with each other. Understanding the specific nature and root cause of CPS bugs would help to design better verification and validation (V&V) techniques for these systems such as domain-specific mutants.

Aim: We look at CPS bugs from an open-source perspective, trying to understand what kinds of bugs occur in a set of open-source CPSs belonging to different domains.

Method: We analyze 1,151 issues from 14 projects related to drones, automotive, robotics, and Arduino. We apply a hybrid card-sorting procedure to create a taxonomy of CPS bugs, by extending a previously proposed taxonomy specific to the automotive domain.

Results: We provide a taxonomy featuring 22 root causes, grouped into eight high-level categories. Our qualitative and quantitative analyses suggest that 33.4% of the analyzed bugs occurring in CPSs are peculiar to those and, consequently, require specific care during verification and validation activities.

Conclusion: The taxonomy provides an overview of the root causes related to bugs found in open-source CPSs belonging to different domains. Such root causes are related to different components of a CPS, including hardware, interface, configuration, network, data, and application logic.

Keywords: Cyber-Physical Systems, Open-source, Bugs, Defects Taxonomy

1. Introduction

Nowadays, software development concerns more and more about software systems interacting with hardware devices such as sensors and actuators. Examples include automotive and avionic systems, as well as, e-health devices [48], and also software governing Internet of Things (IoT) infrastructures for building automation, smart cities, and manufacturing. Such systems are named Cyber-Physical Systems (CPSs). The main distinguishing element of CPSs is that they are systems that collect, analyze, and leverage sensor data from the surrounding environment to control physical actuators at run-time [1, 7].

The interaction of a CPS with hardware devices, as well as with humans and other systems, makes the nature and effect of bugs in CPS environments very specific and non-predictable [57, 61]. One of the most famous software failures in a CPS is the one related to the Ariane 5 [19, 33], caused by improper reuse from its predecessor, i.e., Ariane 4. As a consequence, there is the need to empirically define a CPS-specific bug taxonomy that helps to determine the root causes of different bugs that might occur in a CPS. Such a taxonomy would help in developing effective CPS-specific bug detection tools and techniques.

Email addresses: fzampetti@unisannio.it (Fiorella Zampetti), dev.ritu.kapur@gmail.com (Ritu Kapur), dipenta@unisannio.it (Massimiliano Di Penta), panc@zhaw.ch (Sebastiano Panichella)

Definition 1: CPS bug and CPS failure

We define *CPS bug* as “a flaw in the hardware (not properly handled by the software), or an incorrect interaction between the software and hardware components leading to a CPS misbehavior.” A CPS bug can manifest as a *CPS failure*, which makes a CPS unable to deliver its required functionality or not fulfilling some non-functional properties.

CPS-specific bugs can occur in presence of broken sensors [40] or a security attack [58], leading to (unexpected) inputs resulting in a misbehavior of the autonomous system. As an example, in PX4-AUTOPILOT the presence of noisy data, i.e., false or unrealistic, coming from airspeed sensors, leads to an unexpected behavior of the drone in-flight, i.e., *“the controller scaled the control surface signals up leading to heavy oscillations”*[1]. A different example has been experienced in the OPENPILOT project where, on a specific device (Rav4 Prime) the software does not work as expected while *“a CAN bus error occurs”*[2]. After having tried the software over different devices, developers ended up that the software is not compatible with the Rav4 Prime. The latter highlights that it is possible to see unexpected behavior in the CPS when running the software on unsupported hardware devices.

The *goal* of this paper is to empirically define a *taxonomy of bugs occurring in CPSs belonging to different domains* to design better verification and validation (V&V) techniques for CPSs. The purpose of devising such bug taxonomy is manyfold. Specifically, it can be useful to (i) better understand the root causes of failures [23], or (ii) better plan code review [39] and testing activities, as well as, (iii) to define domain-specific (testing) mutants. The latter is important since previous research has shown that test mutants do not always represent real faults [31], and therefore domain-specific mutants’ taxonomy may be required for emerging systems such as CPSs. Not only the fault distribution would change, but, also, there could be faults, e.g., related to sensor failures (or interfacing), which may not be fully captured by existing general-purpose mutants’ taxonomies [32, 35].

We stem from the Garcia et al.’s autonomous vehicle (AV) bug taxonomy [23] derived by looking at data of two AV systems, and contribute with a differentiated replication study (i.e., a replication differing both in terms of domain, methodology, and possibly focusing on specific aspects of the original study [13, 49]) as summarized in the following:

- *Different purpose*: while the goal of the Garcia et al.’s study is to relate symptoms and root causes, we only focus on the root causes with the aim of supporting future research aimed at deriving specific mutation testing strategies for CPSs.

- *Different domains*: while Garcia et al.’s study specifically targets bugs about self-driving car software from two systems, our work involves more projects and spans across different CPS domains. Specifically, we analyze a more heterogeneous set of bugs from 14 different projects including Arduino (e.g., Arduino core, as well as, Internet of Things – IoT, and Infrared Remote libraries for Arduino), drones, robotics, and automotive.

- *Number of projects analyzed*: while Garcia et al.’s work targets two open-source projects, we complement this work by analyzing 1,151 issues from 14 open-source projects belonging to four different CPS domains.

- *Explicitly distinguishing CPS-specific bugs from generic bugs*: while discussing different bug categories, we make the case for, and discuss, bugs that are specific to CPSs from bugs that may occur in any (conventional) software system.

The bug categorization has been conducted using a hybrid card-sorting approach [45], i.e., we started from a set of predefined categories used in Garcia et al. [23]’s work. While their work belongs to the automotive domain, we found its domain relatively close to our work (as we also consider automotive

[1] https://github.com/PX4/PX4-Autopilot/issues/8980

[2] https://github.com/commaai/openpilot/issues/2103

projects), and the categories they defined in terms of root causes applicable to our context as well.

As a result of the manual categorization, we obtained a taxonomy featuring 22 root causes, grouped into eight high-level categories. The taxonomy enhances and extends the one previously created by Garcia et al. [23] for AV bugs. To the best of our knowledge, this is the first work that *proposes a taxonomy aimed at identifying the root causes of the bugs introduced by developers while developing CPSs including hardware, network, interface, data, configuration, algorithm and documentation-related bugs*.

On the one hand, our results point out that ≈ 33% of the bugs are CPS-specific. Even if this percentage may appear to be relatively limited, it is not entirely surprising that the majority of bugs occurring in CPS (as in any other traditional software system) are conventional (e.g., programming logic or other) bugs. On the other hand, our set of CPS-specific bugs could be used to define mutation testing or fault-injection [51] strategies specific to CPS domains, as well as, to testing solutions specific to CPS.

Our replication package, submitted as additional material for review and available on Zenodo [56], contains (i) the scripts developed to extract the data used for this research and (ii) the manual validated dataset of bugs occurring in CPSs.

The paper is organized as follows. Section 2 details the study definition and planning. The CPS bug taxonomy is presented and discussed in Section 3. The study implications for developers and researchers are discussed in Section 4, while Section 5 discusses the the threats to the study validity. Finally, Section 6 discusses related research, while Section 7 summarizes our findings, and outlines directions for future work.

2. Study Definition and Methodology

The *goal* of our study is to analyze the root cause of bugs occurring in open-source CPSs. The *perspective* is of researchers developing suitable approaches to support the discovery, localization, and management of CPS-specific bugs. Also, the study results can be useful to developers in understanding the nature of bugs occurring in CPSs. The *context* of the study consists of 1,151 closed issues sampled from 14 open-source CPS projects hosted on GitHub.

Specifically, we answer the following research question:

RQ: *What types of bugs occur in open-source CPSs?*

This research question *focuses* on qualitatively defining a taxonomy comprising of root causes for *bugs* occurring in CPSs. We aim at discriminating bugs specific to CPS from bugs similar to those also occurring in traditional (general-purpose) software.

In the following, we detail the methodology adopted to answer our research question.

2.1. Methodology Overview

Fig. 1 depicts the methodology we followed, which consists of four subsequent steps. First, (1) we performed an inception phase aimed at enriching our knowledge about the studied problem and determining a starting point for our taxonomy. Then, (2) we selected the pool of projects to be considered in the study and extracted issues from them. After that, (3) we performed the CPS bugs categorization, which involved four people (two annotators and two reviewers). Finally, (4) an independent annotator re-labeled the whole set of bugs, to limit subjectivity, and after having solved conflicts, the final taxonomy of CPS bugs was created.

2.2. Inception Phase

As a first step, we needed to enrich our knowledge by identifying from previous literature, studies aimed at characterizing bugs in different software application domains. More specifically, as detailed in Section 6, we looked at previous bug taxonomies [23, 24, 27]. Among them, the closest (in terms of the domain) was the one proposed by Garcia et al. [23], which looked at bugs affecting two open-source autonomous vehicles (AV) systems. They classified the root causes of AV bugs into 13 categories, summarized in Table 1.

By looking at their descriptions, we found such categories a suitable starting point for our work, mainly because they feature

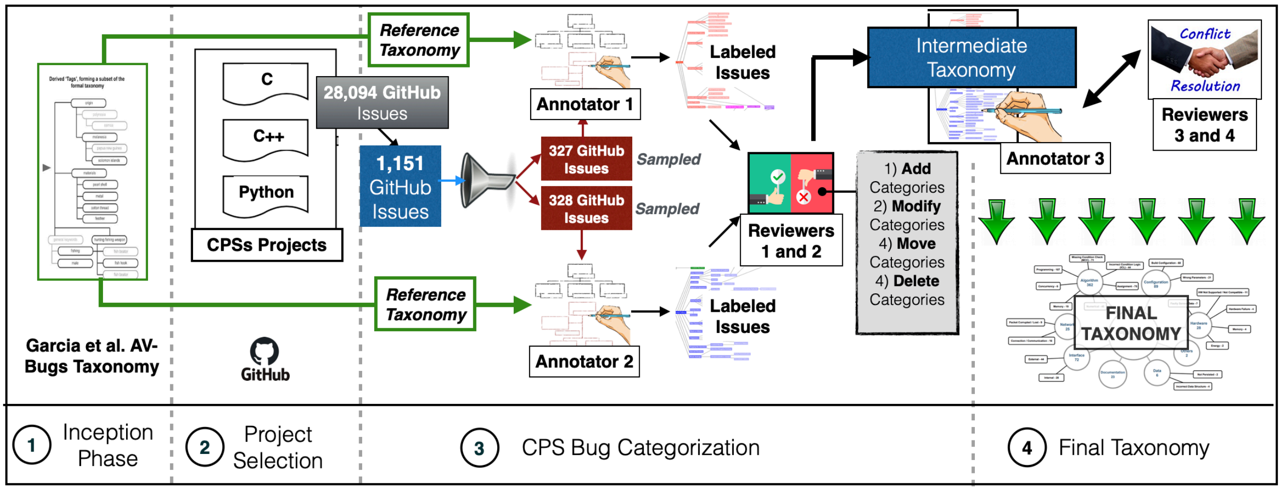

Figure 1: Research methodology

conventional bugs (e.g., missing condition checks and incorrect condition logic) but also numerical bugs, and bugs related to how the software interfaces with hardware components, which are likely to occur in CPSs. As a consequence, we refer to the categories detailed in Table 1 as a starting point for our investigation, while (i) adding further categories, and (ii) determining the extent to which those categories apply in the open-source CPSs relying on different domains.

2.3. Project Selection

To properly derive a taxonomy aimed at covering root causes for bugs occurring in CPSs, we selected 14 open-source projects hosted on GitHub. To identify CPS-related projects, we used the GitHub query search feature. Since our goal is to identify projects belonging to different CPS domains, including the previously studied automotive domain [23], we experimented with CPS-specific GitHub queries: *"Drone"*, *robot*, and *autonomous-vehicles*. In addition, we also considered *arduino* to explicitly target projects that likely design and manufacture single-board micro-controllers and micro-controllers kits for building digital devices.

Starting from the initial set of projects, we applied the following selection criteria, ending up with a sample of 14 projects whose characteristics are summarized in Table 2:

- *Project Popularity*: We sorted the results by stars to focus on popular repositories. Note that, while selecting projects solely based on stars has been criticized [11], this is not our only criteria.

- *Programming languages*: We selected projects having as main programming languages C++ or Python since, while querying GitHub for projects belonging to the CPS domain, we realized that most of them use the selected languages. This is consistent with the findings from previous literature highlighting that CPS development is mostly performed using C/C++, with many complex CPSs that *"do not allow the use of other languages"* such as Java or Swift [44].

- *Use of GitHub issue tracker*: The projects must rely on GitHub for tracking their issues, so we only considered projects having at least 100 closed issues.

- *Use of issue labels*: Since our goal is to classify bugs and not any type of issue (e.g., enhancements or new features), we only focused on projects that use labels for discriminating about different types of issues, accounting for those projects having at least 10 issues being related to bugs, and labeled as such.

- *Active Projects*: We selected projects having at least 5 closed issues in the last 3 months of the observed period.

Table 1: Garcia et al.'s taxonomy [23].

Name of the Root Cause	**Description of the Root Cause**
Incorrect algorithm implementation	A difficult to fix incorrect implementation of the algorithm's logic
Incorrect numerical computation	Incorrect numerical calculations, values, or usage
Incorrect assignment	Variable(s) incorrectly assigned or initialized
Missing condition checks	A necessary conditional statement is missing
Data	Incorrect data structure (or data type conversion) and misused data pointers
Misuse of an external interface	The misuse of interfaces of other systems or libraries
Misuse of an internal interface	Misuse of interfaces of other components and incorrect opening, reading, and writing
Incorrect condition logic	This occurs due to incorrect conditional expressions
Concurrency	Misuse of concurrency oriented structures
Memory	Misuse of memory (e.g., improper memory allocation or de-allocation).
Invalid Documentation	Incorrect manuals, tutorials, code comments, and text.
Incorrect configuration	Modifications to files for compilation, build, compatibility, and installation
Other	Root causes that do not fall into any one of the above categories.

The number of projects to consider as context for our study has been determined so that (i) we were able to sample and analyze enough issues per project (this would not be possible if analyzing a large number of projects and then sampling very few bugs from them), and (ii) we had a similar number of projects for each CPS domain.

2.4. CPS Bug Categorization

Once having identified the projects of interest, we proceeded with the extraction of their GitHub issues data (i.e., title, description, labels, status, and comments) using *Perceval* [20].

As reported in Table 2, we downloaded a total of 32,333 issues of which 28,094 are closed issues. Once having filtered out all those closed issues that did not have a bug-related label, we ended up with a total of 3,713 issues relevant to our study.

Issue Sampling. Since manually analyzing all bug-related issues in our dataset would be infeasible, we extracted a statistically significant sample to be analyzed. Specifically, we applied a stratified random sampling on each project, with a significance level of 95% and a confidence interval of ±2.4%, which led to the selection of 1,151 closed bug-related issues to be manually analyzed. The sample size (SS) is based on a formula for an unknown population [41]:

$$SS = p \cdot (1-p)\,\frac{Z_\alpha^2}{E^2}$$

and SS_{adj} for a known population pop:

$$SS_{adj} = \frac{SS}{1 + \frac{SS-1}{pop}}$$

where p is the estimated probability of the observation event to occur (assumed to be 0.5 if we do not know it a priori), Z_α is the value of the Z distribution for a given confidence level, and E is the estimated margin of error (5%).

Preliminary Bug Categorization. By following the recommendation from previous work [50], we choose to label only the issues where the discussion was not tangled to reduce the possibility of misclassifying their root causes. Moreover, since developers may assign inappropriate labels when opening and discussing issues [4, 25], we discard the issues that were not bugs, despite the label. In other words, we performed a first high-level manual filtering of the issues in our dataset, discarding issues that (i) were not bugs, despite the label; (ii) were not linked to a specific fix; and (iii) were duplicates of already analyzed issues. After this preliminary manual filtering, we ended

Table 2: Characteristics of the analyzed projects.

Project	**Domain**	**Issues**	**Closed Issues**	**Bug-related Issues**
Autoware-AI/autoware.ai	Automotive	1,030	1,027	49
commaai/openpilot	Automotive	608	562	78
ArduPilot/ardupilot	Drones	5,123	3,613	1,044
PX4/PX4-Autopilot	Drones	5,875	5,303	1,197
dronekit/DroneKit-Python	Drones	631	311	62
mavlink/qgroundcontrol	Drones	3,873	3,038	92
ros/ros	Robotics	100	99	41
carla-simulator/carla	Robotics	3,298	2,933	168
cyberbotics/webots	Robotics	1,076	912	486
bblanchon/ArduinoJSON	Arduino	1,346	1,329	136
Arduino-IRremote/Arduino-IRremote	Arduino	505	502	17
miguelbalboa/rfid	Arduino	334	312	17
espressif/arduino-esp32	Arduino	3,367	3,208	12
esp8266/Arduino	Arduino	5,167	4,945	314
TOTAL	**-**	**32,333**	**28,094**	**3,713**

up with a set of 655 bugs to be used for deriving the taxonomy for CPS bugs. With respect to the initial population, this is a statistically significant sample with 95% confidence level and ±3.78% confidence interval.

On the useful set of 655 issues, we performed a hybrid card-sorting approach [45], by considering the Garcia et al.'s taxonomy [23] as a reference starting point.

Our card sorting consisted of the following steps:

1. *Annotation Phase*: We split the 655 issues into two sets. Two annotators independently evaluated the assigned set of issues and proposed labels for them. To identify the root cause of an issue, the annotators looked at the title, description, discussion, and source code change diffs associated with each linked fix. The labeling was performed by either reusing labels provided by Garcia et al. [23], thus adding new labels when necessary.

2. *Reviewing Phase*: Each labeled issue has been subsequently reviewed by a different annotator (not involved in the initial annotation phase), which confirmed or rejected the categories assigned in the previous step. In presence of disagreement, i.e., the new annotator rejected the previously identified category, a discussion was opened involving an additional annotator (not involved in the previous labeling). A decision was taken by applying a majority vote strategy among the participants of the discussion. Among the 655 analyzed issues, a discussion was needed in 93 cases.

As an outcome of the overall bug categorization step, a preliminary version of the taxonomy was created.

2.5. Definition of the Final Taxonomy

To guarantee the integrity of the labeled dataset, i.e., reducing possible subjectivity and bias, and of the emerging categories, i.e., removing potential redundancies from the preliminary version of the taxonomy, an additional annotator, not involved in the previous bug categorization step, re-labeled the previously manually validated issues independently. Then, the two labeled datasets have been compared to identify disagreements, that have been discussed and solved in a discussion session with two different reviewers.

Besides performing a discussion to resolve disagreements, we computed the inter-rater agreement to determine to what extent annotators agreed by chance. That is, if the inter-rater agreement is too low compared to the agreement rate, this indicates that several cases of agreement could have (also mistakenly) occurred by chance. To determine the reliability of the manual labeling, we used the Cohen's k inter-rater agreement [16]. Specifically, the agreement has been computed at two different levels. First, we computed the agreement looking at whether or not a bug is CPS-specific, obtaining a percentage of agreement of $\approx 93\%$ and a Cohen $k = 0.84$, which indicates an almost perfect agreement between the annotators. Then, we looked at the percentage of agreement while assigning the high-level category of our taxonomy. In this case, the percentage of agreement is equal to 76% with a Cohen $k = 0.66$, representing a strong agreement.

3. Study Results

Table 3 reports the root causes of bugs in the CPSs we analyzed. The taxonomy comprises 22 different root causes properly grouped into 8 high-level categories:

1. *Hardware* includes bugs whose root cause is in the hardware and its related software, e.g., faulty data sent by sensors, components not supported by the current implementation, or overflow of the physical storage on the real device;
2. *Network* includes bugs whose root cause is in the communication between the hardware and the software components in terms of connections and packets being lost or corrupted;
3. *Interface* groups bugs whose root cause resides in a misuse of the interface of the hardware devices, other software libraries and/or components, as well as, bugs inherited by third-party components integrated into the system;
4. *Data* groups bugs caused by the usage of an improper data structure, as well as related to its storage;
5. *Configuration* includes bugs due to a wrong build configuration process;
6. *Algorithm* groups bugs whose root cause is related to how the application logic is implemented, e.g., incorrect conditional expressions, incorrect numerical calculations or values, as well as misuse of memory in terms of strategies used for allocating or de-allocating it;
7. *Documentation* includes bugs that do not occur in the source code but rather in the documentation associated with the application, i.e., documentation is outdated with respect to the current version of the software;
8. *Others* groups those root causes that cannot be classified into any of the previous categories.

Table 3 also reports, for each root cause, the number and percentage of bugs belonging to it discriminating between bugs that are specific to CPSs (e.g., related to hardware or its interfacing, or algorithms related to hardware controlling), as well as, generic bugs that are not CPS-specific.

In the following, we discuss each category, reporting a short description together with representative examples for each specific root cause, and then outline our main findings including, whenever possible, implications for practitioners and researchers. The discussion starts from the categories not included in the classification of AV bugs by Garcia et al. [23] (i.e., Hardware and Network), and then considers the categories that abstract or specialize those from the previous taxonomy.

3.1. Hardware

Description: We found five root causes dealing with hardware components being integrated with CPSs: (i) *HW not supported/not compatible* groups bugs that are generated by using a hardware component/device that is not supported or is not

Table 3: Taxonomy of CPS bugs: number of bugs for each root cause, number (percentage) of CPS-specific bugs, and other bugs.

Category	**Root Cause**	**# of (%)**	**# of (%)**
		CPS-specific bugs	**other bugs**
Hardware	Energy	2 (0.31)	0 (0)
	Faulty Sensor Data	7 (1.07)	0 (0)
	Hardware Failure	4 (0.61)	0 (0)
	HW Not Supported/Not Compatible	11 (1.68)	0 (0)
	Memory	4 (0.61)	0 (0)
Total Hardware		**28 (4.28)**	**0 (0)**
Network	Connection/Communication	5 (0.76)	11 (1.68)
	Packet Corrupted/Lost	8 (1.22)	1 (0.15)
Total Network		**13 (1.98)**	**12 (1.83)**
Interface	External	27 (4.12)	27 (4.12)
	Internal	6 (0.92)	24 (3.66)
Total Interface		**33 (5.04)**	**51 (7. 78)**
Data	Incorrect Data Structure	0 (0)	9 (1.37)
	Not Persisted	0 (0)	2 (0.31)
Total Data		**0 (0)**	**11 (1.68)**
Configuration	Build Configuration	2 (0.31)	68 (10.38)
	Wrong Parameters	5 (0.76)	13 (1.98)
Total Configuration		**7 (1.07)**	**81 (12.36)**
Algorithm	Assignment	19 (2.9)	58 (8.85)
	Concurrency	2 (0.31)	5 (0.76)
	Incorrect Condition Logic	17 (2.6)	35 (5.34)
	Memory	5 (0.76)	16 (2.44)
	Missing Condition Check	25 (3.82)	50 (7.63)
	Numerical	18 (2.75)	23 (3.51)
	Programming	26 (3.97)	90 (13.74)
Total Algorithm		**112 (17.11)**	**277 (42.27)**
Documentation	-	1 (0.15)	27 (4.12)
Others	-	0 (0)	2 (0.31)
OVERALL		**194 (33.40)**	**461 (66.56)**

compatible with the CPS system; (ii) *Faulty Sensor Data* includes bugs due to sensors providing faulty values; (iii) *Memory* groups bugs generated by storage on physical devices; similarly, (iv) *Energy* includes bugs dealing with the power on physical devices; and (v) *Hardware Failure* groups bugs where the root cause of the failure is directly on the hardware component/device. Based on the above description, and as reported in Table 3, this category includes only CPS-specific bugs.

Discussion and Examples: 11 out of 28 bugs belong to *HW not supported/not compatible*. Bug #2103[3] in OPENPILOT points out the presence of a CAN bus error on a specific device (i.e., Rav4 Prime). After a detailed discussion, in which other users still experienced the same problem, the developer

[3] To access the bug description use https://github.com/$owner/$repo/issues/$issue_number. For this specific example it is https://github.com/commaai/openpilot/issues/2103.

ended up stating: “This isn’t an issue - the RAV4 Prime isn’t listed as a supported car”. From a different perspective, still in OPENPILOT we found a different bug (#1813) where the problem is due to the usage of the wrong simulator, indeed, since that “some older HKG vehicles do not have FCA11msg” it is required to “use SCC12 for stock ADAS signals on cars that don’t have FCA11”.

Faulty Sensor Data includes seven CPS-specific bugs. For instance, we found a case in which there is a GPS glitch (bug #14253 in ARDUPILOT), or, false and unrealistic airspeed measurements (bug #8980 in PX4-AUTOPILOT) resulting in an unsuccessful flight.

We found four CPS-specific bugs belonging to *Memory*: consider bug #1662 in AUTOWARE.AI where the functionality crashes since the disk capacity has been filled up, or bug #2352 in ARDUPILOT where the misbehavior (i.e., empty logs list over MAVLink) occurs as a consequence of the SD card being full.

We found two CPS-specific bugs belonging to *Energy*. As an example, in ARDUPILOT, the user experienced a crash together with a partial data freeze as a consequence of “a power failure. Power overload, not a software problem” (bug #6300).

Finally, we found four bugs for which the root cause is directly related to the hardware component/device. For instance, bug #9738 in PX4-AUTOPILOT, after a very long discussion, has been closed stating that: “... it turns out that my board had hardfaulted ... and then it got “stuck” waiting for keyboard input on the console to clear the hardfault. The fault wouldn’t/didn’t clear itself over multiple reboots, leading to a “bricked” board.”

Main Findings: Hardware-specific bugs are peculiar to our taxonomy, and, unsurprisingly, all of them are CPS-specific. Recognizing (and simulating) hardware failures has paramount importance in V&V. Also, developers should take particular care of hardware compatibility, especially for CPSs targeting multiple devices. Last, but not least, the interaction with the hardware makes particularly crucial the analysis of non-functional properties such as performance, memory, and energy consumption.

3.2. Network

Description: Differently from the Garcia et al.’s work [23], we have identified a new root cause accounting for bugs occurring in the networking between software and hardware components. Indeed, the network plays a paramount role in many CPSs [42], e.g., in the Internet-of-Things (IoT) domain, or domains such as drones, satellite, automotive, etc. [3, 21, 55]. We discriminated between bugs dealing with (i) packets being lost or corrupted, and (ii) merely connection problems. Out of 25 bugs in this category, 13 are CPS-specific with most of them belonging to *Packet Corrupted/Lost*.

Discussion and Examples: Bug #1696 in OPENPILOT is an example of *Packet Corrupted/Lost*, in which the fault is due to an improper parity bit and command being received in the message order (i.e., packet is corrupted). With the same root cause, we also found bug #4302 in ARDUINO, where there is a memory leak while doing repeated connections to a server, causing the loss of around 8KB for each connection.

For what concerns bugs with *Connection/Communication* root cause, we found five out of 16 being CPS-specific. In ARDUPILOT we found a bug (#11398) where the problem experienced is related to “gimbal’s tilt control is overshooting badly” while using the ChibiOS environment. After a long discussion, a developer found that the root cause of the misbehavior is the latency while communicating through an i2c bus. A different problem has been reported in ARDUINO (bug #4060): once having properly completed a sequence of requests through the network, the user started to receive a "Connection Refused" error and while looking at the connection status it was lost. After those events, it was neither possible for the user to reconnect to the network.

Main Findings: In several CPSs, networking plays a paramount role, and therefore can be the origin of bugs. The CPS monitoring infrastructure should therefore include network monitors. Moreover, V&V techniques may contemplate CPS misbehavior caused by network-specific aspects.

3.3. Interface

Description: This category groups bugs dealing with a misuse of the interfaces among software and hardware components (*External*), as well as among different software components that may be both external software libraries being used in the CPS system (*External*), or modules and/or classes in the same CPS system (*Internal*). It is important to remark that when the reported issue affects a third-party library used by the system (i.e., inherited bugs), we did not exclude it, but labeled it as an *external interface bug*. 84 out of 655 ($\approx$ 13%) bugs in our dataset belong to this category, with 33 of them being CPS-specific.

Discussion and Examples: As regards the inheritance of bugs in third-party components being integrated into the software application (*External*), in PX4-AUTOPILOT we found a bug (#6546) dealing with GPS "jamming" that has already been reported as an issue in the library aimed at supporting the Intel Aero Platform. In other words, the bug has been inherited from the library being used while interfacing with GPS. However, in the *External* root cause, there are CPS-specific bugs dealing with the interface towards the hardware components. For instance, bug #8822 in PX4-AUTOPILOT reports a problem with the name of the rotors to use while connecting to the drone. In this case, it is required to use "rear" instead of "rear-right".

We also found cases where, as a consequence of updates in the firmware, the CPSs start to not work as expected. For instance, bug #226 in ARDUINO reports: "ESP8266HTTPUpdateServer fails to check sketch size versus available space" and, while looking at the fix we realized that the bug has been raised as a consequence of the new firmware being released (i.e., make updater fail if there is not enough space to fit the new firmware[4]).

Finally, we found functional bugs in the drivers' implementation (i.e., interface component). As an example, the issue report #11854 in PX4-AUTOPILOT points out that the driver connected to the temperature sensor is not able to provide the right results even if the sensor values are not corrupted. By looking at the issue report, we found that the problem is in the logic implemented for filling the report buffer with the temperature values coming from the sensor. To fix the problem, the developer changed the code while considering the value reported in the datasheet of the sensor measuring the temperature values (i.e., BMI055).

Concerning the interface with *Internal* software components, we found only six CPS-specific bugs out of 30 bugs. Very often developers tend to rely on the wrong API (or misuse certain APIs) while accomplishing a specific task. For instance, bug #66 in DRONEKIT-PYTHON states that there is a wrong usage of the API used for giving a command: the developer relies on the "VehicleMode" class instead of using the "Command" class.

Main Findings: Interfacing bugs are challenging for developers coping with CPSs. Surely, testing efforts should focus on this aspect. Moreover, when hardware or firmware changes, there may be a lack of documentation or code examples for developers.

3.4. Data

Description: This category groups bugs whose root cause is in the way data is stored (*Not Persisted*) and handled (*Incorrect Data Structure*) by the application logic of the system. Quite surprisingly, we did not find any bug that is specific to the CPS domain, and, as reported in Table 3, only a few bugs (1.68%) belong to this category.

Discussion and Examples: Two bugs are related to the data persistence, and nine deal with incorrect data structures being

[4] https://github.com/esp8266/Arduino/pull/2405

used for modeling purposes. An interesting example is bug #60 in ArduPilot stating that some settings are not stored permanently while they should (i.e., "video device input setting not stored permanently"). The latter comes with unexpected behavior summarized by the user as: "the video is replaced with a white box and the desired input device needs to be selected again".

Main Findings: We did not find many data-related bugs, and those found are not CPS-specific. Although data exchange and storage have paramount importance for CPSs, we found that problems tend to occur at the interface level rather than being related to data structure management. Therefore, developers (and testers in particular) should focus their effort on that.

3.5. Configuration

Description: This category groups bugs related to (i) how the build process is configured, and in particular how the commands or the environments have been configured; or (ii) how the application run-time parameters are configured. Even if ≈ 92% of them are not CPS-specific, we found seven bugs, i.e., two in *Build Configuration* and five in *Wrong parameters* that are strictly related to the CPS domain.

Discussion and Examples: In PX4-AutoPilot we have a bug (#2229) in which a user found that a driver for a specific hardware component (i.e., PCA8574) is included in the startup script even if it is never compiled. As a solution, the developer, instead of completely removing it from the script, only commented it out since "it might be useful down the road". Moving to the *Wrong parameters* root cause, we refer to bug #1017 in ArduinoJSON where, after a crash of the board, the watchdog timer resets correctly but the board will not restart automatically. After a long discussion in which the developer provides additional details on how to properly connect the hardware components to the software application, a hard reset was forced. The latter translates into modifying the parameters being used while running a specific command during configuration[5].

Main Findings: Besides conventional build scripts, CPS developers should pay specific attention to properly configuring builds aimed at integrating (and possibly testing) different combinations of drivers/hardware versions.

3.6. Algorithm

Description: This category includes 389 bugs (of which 112 are CPS-specific) mainly due to how the application logic is implemented. In this category we have seven different root causes: (i) *Assignment* deals with variables that are wrongly assigned and/or initialized; (ii) *Missing Condition Check (MCC)* includes bugs related to a logic being only partially implemented; (iii) *Incorrect Condition Logic* groups those bugs in which the condition logic has been improperly implemented; (iv) *Numerical* includes bugs due to incorrect numerical calculations and values/ranges; (v) *Memory* accounts for bugs in the logic used by the application while allocating or de-allocating the memory; (vi) *Concurrency* groups bugs that are related to a misuse of the concurrency-oriented structures; and (vii) *Programming* deals with bugs that cannot be assigned to only one of the other root causes in the same category.

Discussion and Examples: 77 out of 389 algorithm bugs are related to *Assignment*. As an example, in ArduPilot we found a bug (#801) related to how the vertical acceleration was being set (i.e., assigned the first time being used) and used. This has been confirmed by the fix commit stating: "Vehicle was not reaching target climb or descent rate because of incorrectly defaulted acceleration". In PX4-Autopilot, instead, we found a bug (#1098) manifesting during compilation dues to a parameter not being initialized (i.e., "warning: 'alt_sp' may be used uninitialized in this function").

As reported in Table 3, 75 bugs are generated from conditions that are not considered and handled (i.e., Missing Condition Check). As expected, 66.7% of them are not CPS-specific and can occur in systems independently on whether or not they

[5] https://github.com/esp8266/Arduino/pull/5433/files

interact with hardware devices and simulators. Among the 25 CPS-specific bugs, in ARDUPILOT a bug (#2620) discussing that the value outputted by the barometer sensor in a specific condition is not handled by the application. Specifically, the user stated: "the barometer altitude became NaN [...] but the EKF probably continued to use the barometer altitude because the EKF's readHgtData method doesn't check the health of the baro". Fixing this problem requires checking the status of the sensor before consuming its value.

As regards the *Incorrect Condition Logic*, we identified 52 bugs belonging to this root cause of which 17 are CPS-specific. For instance, in ARDUPILOT we found a bug (#5660) in which the user discovered that while stopping the propeller movement, the Revolutions per Minute (RPM) sensing does not provide any new measurement, so no updates are processed by the Mission Planner. The latter implies that "if your engine dies midair, you will never get a 0 RPM". By carefully analyzing the issue, the developer confirmed that "if you stop getting signals, or get them slower than 1 Hz, then it sets the "quality" to zero and the healthy goes false and it will no longer log it." The bug was fixed by changing the condition being checked to always log RPM when enabled and not only when healthy.

For *Numerical* bugs we found that 44% of them (18 out of 41) are CPS-specific. Compared to the work by Di Franco et al. [18], we only accounted for incorrect numerical calculations and values/ranges. Specifically, we considered all those cases where (i) there is a division by zero, (ii) the value may not have a precise representation in memory because of rounding errors, and (iii) the value is wrongly evaluated, i.e., there is an error in the formula being used to determine the value. For instance, in DRONEKIT-PYTHON we have a bug (#298) manifesting in a race condition dues to the usage of a value wrongly defaulted as "None", before correctly instantiating it. As a result, the application raises a divide by None error. Dealing with this problem means properly assigning the default value instead of relying on "None".

21 out of 389 algorithmic bugs deal with how the *Memory* has been used by the application logic. Note that this root cause is different from the one we have in the *Hardware* category. For instance, bug #5670 in ARDUPILOT where the user reports: "When changing baud rates buffers should be cleared" otherwise having data in the buffer that has not yet been parsed results in falsely detecting valid GPS instances.

In terms of misuse of concurrency-oriented structures, we only found seven cases in our dataset, with only two being CPS-specific. One likely reason for such a smaller proportion is that most of the systems we consider do not use concurrency at all (e.g., Arduino does not have native support for threads). An example of a concurrency bug was found in DRONEKIT-PYTHON (bug #12), in which the root cause has been highlighted by the developer as "... this is caused by race conditions caused by threading. Spin-waiting on two separate threads for parameter confirmation causes a lot of "BAD DATA" messages to pop out [in a non-deterministic manner]".

Finally, most of the bugs in our manually analyzed sample belong to *Programming* (116 out of 389) with 26 being specific to the CPS domain. As an example, in PX4-AUTOPILOT we found bug #5446 dealing with the flakiness of a test command for Arduino, i.e., "The 9250 test command is flaky, interfering with the normal operation of the sensor.".

Main Findings: Algorithmic bugs in CPSs tend to be similar to those occurring in other types of software systems. Therefore, existing mutants taxonomies can be used to seed some representative faults. However, the way failures manifest (e.g., flaky effects on the hardware or actuators) can make these bugs more subtle to detect and potentially dangerous. This should encourage developers to make heavy use (while caring about overhead) of logging and assertion. Some root causes, such as concurrency, are generally avoided "by construction", i.e., by not supporting concurrency at all.

3.7. Documentation

Description: This category includes bugs dealing with the system's documentation. As previous research has pointed out, documentation issues are as important as program-related issues [2, 14]. These problems mostly occur independently from the application domain. Indeed, as reported in Table 3, we found 28 bugs belonging to this category (≈ 4%), with only one being CPS-specific.

Discussion and Examples: In this context a perfect alignment between the system (which includes not only software but also hardware) and the related documentation is crucial. In general, this has been the subject of various studies and approaches [47, 5, 59, 53, 60]. In the context of CPSs, we need to pay attention to properly connecting hardware and software components focusing on how and whether the functionality may change based on the characteristics of the hardware. An example is bug #6522 in ArduPilot where developers, once having struggled to find the root cause of a bug dealing with parameters configuration being lost, realized that: "if you downgrade from Plane 3.8 to an earlier version of the plane then any changes that were made to the RC_* parameters will be lost in the earlier version and that upgrading back to 3.8 will not copy over any param changes that happened in the earlier version. This is only an issue if the user decides to downgrade". As a consequence, it was required to add into the release note a warning that "if users downgrade back to 3.7, their RC settings will likely be wrong, up to inverted servos and crashes".

Main Findings: While documentation inconsistencies in CPSs are similar to other systems, CPS documentation must align with the characteristics and changes of software and hardware components and CPS specific APIs. That is, changes to CPS hardware devices, components, or sensors can trigger changes in the software documentation too.

3.8. Comparison with domain-specific taxonomies

Our work has similarities, but also differences, with respect to previous work on bug taxonomies for specific CPS domains, and in particular the AV taxonomy by Garcia et al. [23] and the Unmanned Aerial Vehicles (UAV) taxonomy by Wang et al. [52].

Differently from previous literature, we made a distinction between bugs specific to CPS domains, and bugs that may also occur in conventional software applications. This is relevant especially when defining domain-specific mutation testing approaches, aimed at capturing bugs that may not be captured using conventional mutation operators. Our results highlight that ≈ 33% of the studied bugs are CPS-specific, indicating that peculiar V&V approaches are required.

Specifically, the root causes of bugs identified by Garcia et al. [23] in the automotive domain can also manifest in different CPS domains (e.g., Arduino or drones). However, our taxonomy highlights the presence of two new bug categories not mentioned by Garcia et al.'s taxonomy [23]. Specifically, we found bugs originating directly from the hardware devices (e.g., faulty sensors, hardware failures, or energy drain), but also from the network infrastructure and protocol. Furthermore, we specialized the *Data* category to account for bugs dealing with data persistence (i.e., *Not Persisted*), and we included the bugs inherited from third-party components in the *External Interface* category.

For what concerns the Wang et al. work [52], identifying eight root causes of UAV specific bugs together with challenges in detecting and fixing them, the main commonalities with our taxonomy are:

- The "Hardware support" category in their taxonomy, is, in our taxonomy, a sub-category of the *Hardware* category. However, while Wang et al. found that the "hardware support" bugs in UAV systems are no different from those in traditional systems, all the bugs in our category are CPS-specific.

- The "Correction" category of Wang et al., dealing with correction of sensor data, can be potentially mapped onto our *Programming* sub-category of the *Algorithm* category, where developers could misuse data coming from sensors without properly cleaning them.

- The "Math" category in their case, can be mapped onto our *Numerical* sub-category of the *Algorithm* category.

- The “Parameter” category of Wang et al. can be potentially mapped onto our (more general) *Interface* category.

Besides that, the rest of Wang et al. taxonomy accounts for categories that are specific to the UAV domain (e.g., "Limit", "Priority", or "Consistency"). These categories of bugs have a lower level of abstraction with respect to the root causes in our taxonomy, which makes them not generalizable/reusable to generic CPS domains. For instance, Bugs having as root causes inconsistencies between hardware and software in a UAV system, could not be generalized in other CPS domains.

In summary, our taxonomy is more generic than Wang et al. taxonomy and its categories do not refer to any specific CPS domain. In other words, we focus on designing a CPS bugs taxonomy from a different perspective, which does make it broader and more re-usable in different CPS domains. Indeed, our taxonomy is less specific to previously investigated domains (e.g., UAVs and automotive), but also more comprehensive, since bugs in our taxonomy also cover bug types not observed in previously studied CPS domains.

4. Implications

This work can have relevant implications for developers and researchers.

For what concerns **developers**, the elicited CPS bug taxonomy highlights specific problems that need to be carefully monitored in CPS development. These include, for instance, the need for coping with multiple hardware versions, which could cause incompatibilities. Also, it is of paramount importance to identify symptoms of hardware failures (e.g., broken sensors) so that they can be properly handled by the software.

The presence of bugs originating from hardware-specific problems highlights the need for complementary software and hardware (which may or may not be available) knowledge in a project.

Finally, to enable the detection and fixing of CPS bugs during the evolution of CPSs, developers should focus on properly configuring CI/CD pipelines aimed at integrating and testing different combinations of drivers/hardware devices in diversified testing scenarios. Of course, we expect that solutions for monitoring and detecting CPS bugs can vary between CPS domains.

For what concerns **researchers**, this work triggers activities towards better testing and analysis of CPSs. First and foremost, given the identified bug taxonomy, it can be used to derive higher order [29] CPS-specific mutation operators. For example, bugs related to faulty-sensor data could lead to mutants that artificially change the sensor inputs towards a faulty value, or else, change the source code by omitting the input correctness check. More complicated would be dealing with memory-related problems, which may require to be simulated by perturbing the configuration of simulators during the testing process. Interface-related mutants could be produced as higher-order mutants from already existing mutants—e.g., those from object-oriented language mutants [32]—by modifying the communication between CPS and devices, for example altering the ordering of method calls and/or passed parameters. Finally, network communication mutants may include, among others, mutants aimed at perturbing the exchanged packets, similarly to what was previously proposed by Xu et al. for web service testing [54]. As explained in previous literature [29], higher order mutants may subsume trivial operators, yet have been shown to be harder to kill than their constituents.

Also, the work could foster the development of specific static analysis tools, looking for CPS-specific recurring problems. Finally, complementary empirical research could be directed to investigate the difficulty (e.g., duration) to fix CPS-specific bugs, and to develop tools guiding developers in allocating the appropriate development effort to various types of CPS bugs.

In the context of CPSs, achieving a deep knowledge of CPS bugs and their root causes would facilitate the development of better approaches and tools to facilitate their reproduction. Specifically, being able to reproduce a bug is crucial during bug triaging and debugging tasks [8, 26, 62]. Researchers proposed several automated solutions to generate test cases reproducing the crashes of software-only systems [6, 15, 30, 38, 43], focusing on the problem of generating the program execution state that triggered a crash in the field.

Fixing or addressing CPS-specific bugs and automatically assessing the correctness of the CPS behavior represent a critical challenge. Our investigation has highlighted types of CPS bugs related to the uncertainty of CPSs behavior. Hence, future studies should look more on safety-related bugs due to the uncertainty of CPS behavior, concerning for instance the CPS initialization, or concerning potential CPS misbehavior. This topic has been recently studied in the automotive domain [9, 10, 46], yet it requires further investigations in other CPS domains.

5. Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. Those are mainly due to the imprecisions in issue classification, e.g., as it is reported in the issue tracker [4, 25], and to the subjectivity/error-proneness of the manual classification. We mitigated both threats with a multi-stage manual classification detailed in Section 2.

Another threat could be related to how our taxonomy has been obtained, especially because we started from the taxonomy proposed by Garcia et al. [23] rather than creating a new taxonomy from scratch. On the one hand, this allows us to build from past experience, especially because of the related domain. On the other hand, there could have been the risk of repeating previous mistakes. This risk has been partially mitigated since every time it was necessary to classify a bug, we determined whether it fitted previous categories, or whether it would have been useful to create new (sub) categories.

Threats to *conclusion validity* concern the relationship between treatment and outcome. As described in Section 2, to achieve a reliable classification of the analyzed bugs, we performed multiple annotation rounds, and then computed the Cohen's k inter-rater agreement.

Threats to *internal validity* may concern the cause-effect relationships between the investigated bugs (effects) and their root causes inferred from the fixes and discussions. We looked at discussions as well as fix change diffs, which could be helpful to infer bugs' root causes. Furthermore, while as explained in Section 2.4 we have excluded from the analysis bugs with multiple root causes, these are likely part of the bugs occurring in a software projects. In other words, we have focused our analysis on the occurrence of single causes, while in practice there may be cases in which multiple root causes co-occur to determine a bug. Further analyses towards the co-occurrence of root causes may therefore be desirable.

Finally, threats to *external validity* concern the generalization of our findings. Although the number of analyzed issues (1,151) is relatively large for a multiple-person manual analysis, and although they have been sampled from 14 projects belonging to different domains, by no means they can be generalized to the universe of open-source CPSs. Therefore, further replications are desirable, in both open-source but also in industrial contexts, to assess the generalizability of the proposed taxonomy.

6. Related Work

This section discusses the literature concerning similar studies and taxonomies about bugs related to different domains.

Dealing with software bugs involves significant costs for software organizations [37]. For this reason, researchers have investigated the nature, root causes, and symptoms of bugs affecting different types of application domains.

Gunawi et al. [24] presented an extensive exploration of issues and bugs associated with cloud applications. According to them, the dominant aspects of cloud-related bugs are associated with performance, security, Quality of Service (QoS), reliability, and consistency. This means that cloud application bugs have different characteristics and are typically more difficult to detect (e.g., bugs involving distributed systems) than bugs occurring in more traditional systems [24, 34, 36].

Linares et al. [50] proposed a framework for improving the mutation testing of Android applications. To achieve this goal, the authors systematically devised a taxonomy of 262 types of

Android faults grouped into 14 categories by manually analyzing 2,023 software artifacts from different sources (e.g., bug reports, commits).

Humbatova et al. [27] introduced a large taxonomy of faults in Deep Learning (DL) systems by manually analyzing 1,059 artifacts gathered from GitHub commits and issues of projects that use the most popular DL frameworks (i.e., TensorFlow, Keras, and PyTorch) and from related Stack Overflow posts. In a follow-up work, Jahangirova et al. [28] developed a mutant taxonomy for DL.

Fischer et al. [22] studied dependency bugs in the Robot Operating System (ROS), which makes their work more specific to the robotic domain and focused on a specific bug type, compared to our study.

Wang et al. [52] studied Unmanned aerial vehicles (UAVs) bugs by manually analyzing 569 bugs from two open-source GitHub repositories (i.e., PX4 and Ardupilot). In this sample, they found 168 UAV-specific bugs that were related to eight different types of root-causes. Wang et al. also summarized challenges for detecting and fixing the UAV-specific bugs. With respect to our work, which is more general, Wang et al. perform an in-depth analysis of problems specifically occurring in UAVs.

Garcia et al.'s work [23] investigated the bugs affecting two autonomous vehicles (AV) simulation tools. Specifically, they investigated the frequency, root causes, symptoms, and location (e.g., components) of bugs affecting such systems. In this paper, we considered the Garcia et al.'s taxonomy as a reference starting point to design our taxonomy.

To the best of our knowledge, our study represents the first work providing a taxonomy that aims at identifying the root causes of the mistakes made by developers while developing a wide set of CPSs.

Since requirement specifications of CPSs are typically expressed using signal-based temporal properties, Boufaied et al. [12] presented a taxonomy of the various types of signal-based properties and provide, for each type, a comprehensive and detailed description as well as a formalization in temporal logic. Hence, they also reported on the application of the taxonomy to classify the requirements specifications of an industrial case study in the aerospace domain.

Finally, Delgado-Perez et al. [17] applied mutation testing in the nuclear power domain, showing how such a technique can be promising for evaluating test suite effectiveness and to achieve good fault detection. This shows the applicability of mutation-testing in CPS-related domains. At the same time, following the results of previous studies on mutation testing effectiveness [31], as well as previous studies on domain-specific mutants [50], it may be necessary to customize the taxonomies of mutant operators. Therefore, creating a taxonomy of CPS-specific bugs represents a starting point towards that direction.

7. Conclusions and Future Work

This paper studied the root causes of bugs occurring in open-source Cyber-Physical Systems (CPSs). By using a hybrid card-sorting strategy [45], we have manually analyzed a statistically significant sample of 1,151 issues (of which 655 were classified as bugs) from 14 open-source projects hosted on GitHub, developed using C++ and Python, and belonging to different domains, i.e., Arduino, automotive, robotics, and drones.

Our analysis reveals that $\approx 34\%$ of the classified bugs are CPS-specific, most of which are related to hardware, networking, or interface. The inspection of root causes for samples of such bugs suggests different ways in which developers could improve Verification & Validation, but also support the comprehension and evolution of CPSs.

Future work aims at replicating this work in industrial contexts (belonging to different domains), at investigating whether there are recurring (and, possibly, reusable) patterns for certain bug categories, and at proposing specific mutation operators and tools for CPSs.

Acknowledgements

We thank Remy Egloff, Jan Willi, Davide Fontanella, Stefano Anzolut for their help in the labeling of GitHub issues.

We gratefully acknowledges the Horizon 2020 (EU Commission) support for the project *COSMOS* (DevOps for Complex Cyber-physical Systems), Project No. 957254-COSMOS.

References

[1] Academies of Sciences. 2017. *A 21st Century Cyber-Physical Systems Education*. National Academies Press.

[2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 1199–1210. https://doi.org/10.1109/ICSE.2019.00122

[3] Fawaz Alsolami, Fahad A. Alqurashi, Mohammad Kamrul Hasan, Rashid A. Saeed, Sayed Abdel-Khalek, and Anis Ben Ishak. 2021. Development of Self-Synchronized Drones' Network Using Cluster-Based Swarm Intelligence Approach. *IEEE Access* 9 (2021), 48010–48022. https://doi.org/10.1109/ACCESS.2021.3064905

[4] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*. 23.

[5] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic antipatterns: what they are and how developers perceive them. *Empirical Software Engineering* 21, 1 (2016), 104–158.

[6] Shay Artzi, Sunghun Kim, and Michael D. Ernst. 2008. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5142)*, Jan Vitek (Ed.). Springer, 542–565. https://doi.org/10.1007/978-3-540-70592-5_23

[7] Radhakisan Baheti and Helen Gill. 2011. Cyber-physical systems. *The impact of control technology* 12, 1 (2011), 161–166.

[8] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann. 2007. Quality of bug reports in Eclipse. In *Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange, ETX 2007, Montreal, Quebec, Canada, October 21, 2007*, Li-Te Cheng, Alessandro Orso, and Martin P. Robillard (Eds.). ACM, 21–25. https://doi.org/10.1145/1328279.1328284

[9] Christian Birchler, Nicolas Ganz, Sajad Khatiri, Alessio Gambi, and Sebastiano Panichella. 2022. Cost-effective Simulation-based Test Selection in Self-driving Cars Software with SDC-Scissor. In *the 29th IEEE International Conference on Software Analysis, Evolution, and Reengineering - To Appear*.

[10] Christian Birchler, Sajad Khatiri, Pouria Derakhshanfar, Sebastiano Panichella, and Annibale Panichella. 2022. Single and Multi-objective Test Cases Prioritization for Self-driving Cars in Virtual Environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2022). https://doi.org/toappear

[11] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *J. Syst. Softw.* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016

[12] Chaima Boufaied, Maris Jukss, Domenico Bianculli, Lionel Claude Briand, and Yago Isasi Parache. 2021. Signal-Based Properties of Cyber-Physical Systems: Taxonomy and Logic-based Characterization. In *International Conference on Software Engineering, 2021*.

[13] David Brinberg and Joseph E McGrath. 1985. *Validity and the research process*. SAGE Publications, Incorporated.

[14] Bernd Brugge and Allen Dutoit. 2013. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Pearson International.

[15] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 363–374. https://doi.org/10.1145/1542476.1542517

[16] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[17] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John A. Clark, and Inmaculada Medina-Bulo. 2018. Evaluation of Mutation Testing in a Nuclear Industry Case Study. *IEEE Trans. Reliab.* 67, 4 (2018), 1406–1419.

[18] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 509–519.

[19] Mark Dowson. 1997. The Ariane 5 Software Failure. *SIGSOFT Softw. Eng. Notes* 22, 2 (March 1997), 84.

[20] Santiago Dueñas, Valerio Cosentino, Gregorio Robles, and Jesus M Gonzalez-Barahona. 2018. Perceval: software project data at your will. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 1–4.

[21] Xinran Fang, Wei Feng, Te Wei, Yunfei Chen, Ning Ge, and Cheng-Xiang Wang. 2021. 5G Embraces Satellites for 6G Ubiquitous IoT: Basic Models for Integrated Satellite Terrestrial Networks. *IEEE Internet Things J.* 8, 18 (2021), 14399–14417. https://doi.org/10.1109/JIOT.2021.3068596

[22] Anders Fischer-Nielsen, Zhoulai Fu, Ting Su, and Andrzej Wasowski. 2020. The forgotten case of the dependency bugs: on the example of the robot operating system. In *ICSE-SEIP 2020: 42nd International*

Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 21–30. https://doi.org/10.1145/3377813.3381364

[23] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 385–396.

[24] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Article 7, 14 pages.

[25] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 392–401.

[26] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 141–152. https://doi.org/10.1145/2491956.2462167

[27] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1110–1121. https://doi.org/10.1145/3377811.3380395

[28] Gunel Jahangirova and Paolo Tonella. 2020. An Empirical Evaluation of Mutation Operators for Deep Learning Systems. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. 74–84.

[29] Yue Jia and Mark Harman. 2009. Higher Order Mutation Testing. *Inf. Softw. Technol.* 51, 10 (2009), 1379–1393. https://doi.org/10.1016/j.infsof.2009.04.016

[30] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing field failures for in-house debugging. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 474–484. https://doi.org/10.1109/ICSE.2012.6227168

[31] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. 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, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 654–665.

[32] Sun-Woo Kim, John A. Clark, and John Alexander McDermid. 2001. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Softw. Test. Verification Reliab.* 11, 3 (2001), 207–225. https://doi.org/10.1002/stvr.238

[33] Jacques-Louis Lions, Lennart Lbeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle Thomson, and Colin O'Halloran. 1996. *ARIANE 5 Flight 501 Failure.Report by the Inquiry Board*. Technical Report. http://sunnyday.mit.edu/nasa-class/Ariane5-report.html

[34] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. *SIGPLAN Not.* 53, 2 (March 2018), 419–431.

[35] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: an automated class mutation system. *Softw. Test. Verification Reliab.* 15, 2 (2005), 97–133.

[36] Diego Martin and Sebastiano Panichella. 2019. The cloudification perspectives of search-based software testing. In *Proceedings of the 12th International Workshop on Search-Based Software Testing, SBST@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE / ACM, 5–6.

[37] Nachiappan Nagappan and Thomas Ball. 2005. Static Analysis Tools As Early Indicators of Pre-release Defect Density. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 580–586.

[38] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *32st International Symposium on Computer Architecture (ISCA 2005), 4-8 June 2005, Madison, Wisconsin, USA*. IEEE Computer Society, 284–295. https://doi.org/10.1109/ISCA.2005.16

[39] Sebastiano Panichella and Nik Zaugg. 2020. An Empirical Investigation of Relevant Changes and Automation Needs in Modern Code Review. *Empir. Softw. Eng.* 25, 6 (2020), 4833–4872.

[40] Junkil Park, Radoslav Ivanov, James Weimer, Miroslav Pajic, Sang Hyuk Son, and Insup Lee. [n.d.]. Security of Cyber-Physical Systems in the Presence of Transient Sensor Faults. *ACM Trans. Cyber Phys. Syst.* 1, 3 ([n. d.]), 15:1–15:23.

[41] B. Rosner. 2011. *Fundamentals of Biostatistics* (7th edition ed.). Brooks/Cole, Boston, MA.

[42] Bharadwaj Satchidanandan and Panganamala Ramana Kumar. 2016. Secure control of networked cyber-physical systems. In *55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016*. IEEE, 283–289. https://doi.org/10.1109/CDC.2016.7798283

[43] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2020. Search-Based Crash Reproduction and Its Impact on Debugging. *IEEE Trans. Software Eng.* 46, 12 (2020), 1294–1317. https://doi.org/10.1109/TSE.2018.2877664

[44] P. Soulier, Depeng Li, and J. R. Williams. 2015. A survey of language-based approaches to Cyber-Physical and embedded system development. *Tsinghua Science and Technology* 20, 2 (2015), 130–141. https://

doi.org/10.1109/TST.2015.7085626

[45] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[46] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. 2020. Misbehaviour prediction for autonomous driving systems. In *International Conference on Software Engineering*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 359–371. https://doi.org/10.1145/3377811.3380353

[47] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: bugs or bad comments?*/. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 145–158.

[48] Martin Törngren and Ulf Sellgren. 2018. *Complexity Challenges in Development of Cyber-Physical Systems*. Springer International Publishing, Cham, 478–503.

[49] Mark D. Uncles and Simon Kwok. 2013. Designing research with in-built differentiated replication. *Journal of Business Research* 66, 9 (2013), 1398–1405. https://doi.org/10.1016/j.jbusres.2012.05.005 Advancing Research Methods in Marketing.

[50] Mario Linares Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 233–244.

[51] Jeffrey M. Voas and Gary McGraw. 1997. *Software Fault Injection: Inoculating Programs against Errors*. John Wiley & Sons, Inc., USA.

[52] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. 2021. An exploratory study of autopilot software bugs in unmanned aerial vehicles. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 20–31.

[53] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*. 53–64.

[54] Wuzhi Xu, Jeff Offutt, and Juan Luo. 2005. Testing Web Services by XML Perturbation. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005), 8-11 November 2005, Chicago, IL, USA*. 257–266. https://doi.org/10.1109/ISSRE.2005.44

[55] Siddika Berna Örs Yalçin, ömer Demirci, and M. Murat Enes Soltekin. 2021. Designing and Implementing Secure Automotive Network for Autonomous Cars. In *29th Signal Processing and Communications Applications Conference, SIU 2021, Istanbul, Turkey, June 9-11, 2021*. IEEE, 1–4. https://doi.org/10.1109/SIU53274.2021.9477958

[56] Fiorella Zampetti, Ritu Kapur, Massimiliano Di Penta, and Sebastiano Panichella. 2022. *Dataset of the paper "An Empirical Characterization of Software Bugs in Open-Source Cyber-Physical Systems"*. https://doi.org/10.5281/zenodo.4478387

[57] D. Zhang, G. Feng, Y. Shi, and D. Srinivasan. 2021. Physical Safety and Cyber Security Analysis of Multi-Agent Systems: A Survey of Recent Advances. *IEEE/CAA Journal of Automatica Sinica* 8, 2 (2021), 319–333. https://doi.org/10.1109/JAS.2021.1003820

[58] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2020. DeepBillboard: systematic physical-world testing of autonomous driving systems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 347–358.

[59] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald C. Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 27–37. https://doi.org/10.1109/ICSE.2017.11

[60] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald C. Gall. 2020. Automatic Detection and Repair Recommendation of Directive Defects in Java API Documentation. *IEEE Trans. Software Eng.* 46, 9 (2020), 1004–1023. https://doi.org/10.1109/TSE.2018.2872971

[61] Y. Zhou, F. R. Yu, J. Chen, and Y. Kuo. 2020. Cyber-Physical-Social Systems: A State-of-the-Art Survey, Challenges and Opportunities. *IEEE Communications Surveys Tutorials* 22, 1 (2020), 389–425. https://doi.org/10.1109/COMST.2019.2959013

[62] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Trans. Software Eng.* 36, 5 (2010), 618–643. https://doi.org/10.1109/TSE.2010.63