EasyChair Preprint

№ 9

Crashing simulated planes is cheap: Can simulation detect robotics bugs early?

Christopher Steven Timperley, Afsoon Afzal, Deborah Katz, Jam Marcos Hernandez and Claire Le Goues

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

March 2, 2018

Crashing simulated planes is cheap: Can simulation detect robotics bugs early?

Christopher Steven Timperley*, Afsoon Afzal*, Deborah S. Katz*, Jam Marcos Hernandez† and Claire Le Goues*
*Carnegie Mellon University, Pittsburgh, PA
†State University of New York at Potsdam, Potsdam, NY
Email: ctimperley@cmu.edu, afsoona@cs.cmu.edu, dskatz@cs.cmu.edu, jamarck96@gmail.com, clegoues@cs.cmu.edu

***Abstract*—Robotics and autonomy systems are becoming increasingly important, moving from specialised factory domains to increasingly general and consumer-focused applications. As such systems grow ubiquitous, there is a commensurate need to protect against potentially catastrophic harm. System-level testing in simulation is a particularly promising approach for assuring robotics systems, allowing for more extensive testing in realistic scenarios and seeking bugs that may not manifest at the unit-level. Ideally, such testing could find critical bugs well before expensive field-testing is required. However, simulations can only model coarse environmental abstractions, contributing to a common perception that robotics bugs can only be found in live deployment. To address this gap, we conduct an empirical study on bugs that have been fixed in the widely used, open-source ARDUPILOT system. We identify bug-fixing commits by exploiting commenting conventions in the version-control history. We provide a quantitative and qualitative evaluation of the bugs, focusing on characterising how the bugs are triggered and how they can be detected, with a goal of identifying how they can be best identified in simulation, well before field testing. To our surprise, we find that the majority of bugs manifest under simple conditions that can be easily reproduced in software-based simulation. Conversely, we find that system configurations and forms of input play an important role in triggering bugs. We use these results to inform a novel framework for testing for these and other bugs in simulation, consistently and reproducibly. These contributions can inform the construction of techniques for automated testing of robotics systems, with the goal of finding bugs early and cheaply, without incurring the costs of physically testing for bugs in live systems.**

***Index Terms*—automated testing, empirical study, robotics, autonomous vehicles, dataset, repository mining, ARDUPILOT**

I. INTRODUCTION

Robotics and autonomous systems have been around for many years. However, for much of that time, they have been confined to situations in which they do not interact with the general public, e.g., factory automation, industrial control systems, automotive subsystems, vehicles sent to space and under the sea, and other less-visible uses. More recently, the use of robotics and autonomous systems has increased beyond these applications. Companies are currently testing partly autonomous vehicles on public roads and starting to deploy package-delivery drones (Levin, 2017; Glaser, 2017). As the cost of robotics systems has decreased and the technology has become more advanced, the number of systems that can interact with and cause danger to the public has grown.

However, as safety-critical systems, failures in robotics systems can be expensive and, in some cases, deadly. As potentially dangerous robotics and autonomous systems increasingly come into contact with humans, it is essential to develop effective quality-assurance methods. Field testing, unit testing, and verification remain important to quality assurance in robots, but they cannot cover all situations a system may potentially encounter. Field testing is especially critical in safety-critical systems and can identify key issues, but failures at this late stage can be enormously expensive. One notable example of the need for simulation testing is the ExoMars Lander, which crashed in October 2016 at an approximate cost of $350 million in materials and time. After the crash, investigators were able to recreate the circumstances of the crash in simulation, which led to the simulated vehicle also crashing (exo, 2016).

Instead, ideally, bugs can be identified as early as possible, reducing the cost of finding and fixing them, and before they have manifested in physical systems (Williamson, 2008).

Automated full-system testing (e.g., Liu and Mei 2014) in simulation will ideally assist in addressing these deficiencies. Indeed, this is our long-term research ambition: to produce highly effective techniques for automatically detecting bugs in real-world robotic systems through the use of software-based simulation, dramatically reducing the cost of such bugs by avoiding the need of costly deployment. However, simulation, by necessity, represents a simplified abstraction of the environment and the system. Is it sufficiently expressive to trigger and support detection of bugs that manifest in reality?

To answer this question, it is necessary to first understand the factors, if any, that make reproducing and detecting bugs in simulation difficult. In this paper, we systematically produce a dataset of historical bugs in an open-source robotics ecosystem, specifically seeking insight into the difficulties that underlie robotics testing in both simulation and deployment. We then characterise those defects to produce insights into the challenges and opportunities afforded by system-level test generation for such systems in simulation. An in-depth and nuanced knowledge of existing robotics bugs can lead to the development of techniques capable of catching future bugs with similar characteristics. Moreover, if a tool can detect bugs that humans have previously identified in these systems, such a tool may also be able to detect other latent bugs in the

system, especially if the latent bugs have similar characteristics to earlier-identified bugs.

Although datasets of robotics bugs do exist (e.g., Wienke et al. 2016; Steinbauer 2013; Grottke et al. 2010; Cotroneo et al. 2013; Sotiropoulos et al. 2017), none allows faults to be reproduced and inspected in simulation, nor do their accompanying analyses investigate the difficulties of triggering and detecting bugs. Ensuring that bugs can be reliably reproduced allows datasets to be used for a rich diversity of studies, including testing, fault localisation, and automated program repair, as similar datasets for non-robotic systems (Le Goues et al., 2015; Just et al., 2014; Tan et al., 2017; Sahoo et al., 2010; Do et al., 2005; Henningsson and Wohlin, 2004) have demonstrated in broader contexts. These studies inspire our work to recreate and detect robotics and autonomous systems bugs in simulation, with a view towards detecting new bugs, which is a direction the previous work does not take. Indeed, well-defined benchmarks and datasets can be instrumental for clarifying and advancing a coherent definition of a discipline's dominant research paradigms (Sim et al., 2003).

To this end, in this work we identify 228 bug-fixing commits within the version-control history of the ARDUPILOT project through a combination of automatic and manual methods. We examine the bugs that the commits fix and analyse the bugs to discover the salient characteristics of each bug, including the circumstances that trigger it and how to detect it in simulation. Our high-level goal is to determine whether simulation is a feasible mechanism. Our intuition was that most real-world bugs would be difficult and impractical to reproduce in simulation. Specifically, we thought that complex triggering conditions (e.g., concurrent events) and environmental requirements would hamper automated testing; we were pleasantly surprised. Instead, from our analysis, we found the following key insights, which inform the development of automated testing techniques for robotics and autonomous systems:

- The majority of bugs we examine are capable of being discovered in simulation and are, in fact, highly amenable to being found in simulation. These bugs do not depend on environmental factors, the presence of any specific type of hardware or any hardware at all, or events happening concurrently.
- Many bugs occur under different configurations, or require certain input types. In order to detect these bugs in simulation, the system under test must run in different configurations, and exercise a variety of input types.
- As a proof of concept, we present a high-level framework for the automated testing of robotics systems and demonstrate that a fairly simple implementation of that framework is capable of triggering and detecting a subset of the real-world bugs in our dataset.

We envision that the research and engineering communities may benefit from the results of our study in several ways. For example, developers can use our results to better understand the types of faults that are common in robotics systems and take steps to prevent similar faults from occurring in their code. In addition, our results identify the kinds of faults that can affect autonomous vehicles; the community can use these resources to inform development of effective and precise quality-assurance techniques for autonomous vehicles. Our dataset is also valuable for evaluating the effectiveness of techniques for program repair and fault localisation in robotics and autonomous systems.

The main contributions of this paper are as follows:

- We use a principled methodology to mine 228 bug-fixing commits from the version-control history of the popular, open-source autopilot software, ARDUPILOT and package and release the dataset along with a Docker container for each bug, allowing for reliable and consistent reproducibility.
- We conduct an empirical study of the bugs within our dataset and produce a number of key insights, which inform the development of automated testing techniques.
- We present a general-purpose framework for the automated testing of robotics and autonomous systems, based on our systematic analysis of how to trigger and detect the bugs studied above. We also describe an open-source, Python-based implementation of this framework, as a proof of concept.

The rest of the paper is structured as follows. Section II introduces the system, ARDUPILOT, that was used as the subject of our empirical study. Section III describes our methodology for collecting, packaging, and analysing our dataset. Section IV discusses the results of our analysis. Sections V presents our proof-of-concept framework for testing robotics systems. Section VI presents threats to validity. Section VII discusses the existing work that we believe is most closely related to this work. Section VIII concludes, including directions for future work.

II. ARDUPILOT BACKGROUND

In this section, we introduce the reader to the system used to focus our study: ARDUPILOT[1]. The open-source ARDUPILOT project, written in C++, uses a common framework and collection of libraries to implement a set of general-purpose autopilot systems for use with a variety of vehicles, including, but not limited to, submarines, helicopters, multirotors, and aeroplanes. ARDUPILOT is extremely popular with hobbyists and professionals alike. It is installed in over one million vehicles worldwide and used by organisations including NASA, Intel, and Boeing, as well as many higher-education institutes around the world.[2]

Beyond being highly popular and open-source, we choose the ARDUPILOT project as our case study due to its rich version-control history, containing over 29,000 commits since May 2010, and for its consistent bug-fix commit description conventions (discussed in Section III-A).

[1]http://ardupilot.org
[2]http://ardupilot.org/about

Controller	# LOC	# Tests
APMrover2	5,099	4
ArduCopter	15,002	35
ArduPlane	11,940	13
ArduSub	7,317	0

TABLE I
THE NUMBER OF LINES OF SOURCE CODE (CALCULATED USING cloc) AND TEST CASES FOR EACH OF ARDUPILOT'S VEHICLE CONTROLLERS.

A. Usage

To deploy the ARDUPILOT platform to a given vehicle, users must first compile the ARDUPILOT source code using an appropriate set of compilation parameters (describing the kind and features of the vehicle). The resulting binaries are then written to the firmware of the physical *flight controller* device used by the vehicle. The flight controller may either be a dedicated hardware device, equipped with a range of sensors (e.g., barometer, gyroscope, accelerometer), such as the Pixhawk 2 flight controller,[3] or instead, a generic single-board computer (such as the Raspberry Pi) connected to sensors via a daughterboard.

After installing ARDUPILOT to the device, users may use the ARDUPILOT's desktop mission planning software to pre-program a mission (described as a sequence of instructions) and write it to the EEPROM of the device. The parameters of the system, such as its maximum speed, may also be changed by writing to the EEPROM.

Users may interact with the vehicle while in operation in two ways. The operator may issue radio commands to the vehicle using a set of predefined radio channels. Alternatively, users may use a ground control system (provided as part of the ARDUPILOT platform) to indirectly communicate with the vehicle programmatically via the MAVLINK[4] protocol. The MAVLINK protocol allows the operator to issue commands to the vehicle (e.g., go to a certain location, land, change operating mode), and is also used by the vehicle to constantly communicate its status (e.g., telemetry, operating mode) to the ground control system. Prior to launch, users may use the ARDUPILOT's command-line interface and mission planning software to interact with the vehicle as it is docked and connected to an external machine.

B. Project Structure

The source code of the ARDUPILOT project is split across several modules, each of which occupies its own directory at the root of its file structure. The `APMrover2`, `ArduCopter`, `ArduPlane`, and `ArduSub` modules, Table I, are used to implement controllers for specific categories of vehicle (i.e., rovers, copters, planes, submarines). The `libraries` module implements functionalities that are shared by one or more categories of vehicle. The `Tools` module provides tools for interacting with ARDUPILOT systems, such as the mission planner, and replay functionality, as well as a set of automated tests.

C. Simulation

To facilitate rapid prototyping and reduce the costs of whole-system testing, ARDUPILOT offers a number of simulators for most of its vehicles (excluding submarines). In general, those platforms simulate the dynamics of the vehicle under test, feed artificial sensor values to the controller, and relay the state of its actuators to the physics simulation. Hardware-in-the-loop (HIL) simulators are used to perform testing on a given flight controller hardware device by directly reading from and writing to it. In contrast, software-in-the-loop (SITL) simulators test a software implementation of the flight controller by running it on a general-purpose computer.

D. Continuous Integration

In an effort to detect regressions introduced during development, the ARDUPILOT project uses continuous integration to automatically test each commit against a developed-provided test suite. At the time of writing, the developed-provided test suite checks that: (1) the code can be built, both on the desktop and a sub-set of supported boards; (2) that each vehicle controller behaves as expected when passed a series of commands in a simulated environment; and (3) that flight logs are successfully produced.

To check the behaviour of the robot, the test suite spawns the robot at a fixed set of coordinates in simulation, and sends the robot a sequence of commands. After dispatching a command to the robot, the test script continually checks to see whether the command has finished executing. The sequence of commands is split into two halves: the first half tests the *manual* operation of the robot by sending specific commands to the robot; the second half puts the robot into *autonomous* mode, and instructs it to complete a preprogrammed mission.

To expedite the testing process, the test harness makes use of the simulator's time multiplier to speed-up the simulation clock by a factor of ten.

III. METHODOLOGY

In this section, we first discuss our process for identifying bug-fixing commits within the version-control history of the ARDUPILOT project. We then discuss how we transform each bug-fixing commit into an executable Docker image, capable of reproducing the bug in simulation. Finally, we describe how we analysed each bug.

A. Bug Collection

To identify which of the $> 29,000$ commits within the version-control history of the ARDUPILOT represent bug fixes, we used GITPYTHON[5] to mine potential bug-fixing commits from the project's GITHUB repository[6]. To do so, we implemented a script that uses a multi-stage process to identify commits.

[3] https://pixhawk.org/
[4] http://qgroundcontrol.org/mavlink/
[5] https://github.com/gitpython-developers/GitPython
[6] https://github.com/ArduPilot/ardupilot

1) To ensure reproducibility, we restrict our attention to all 29,081 commits within the repository that occurred before October 1st, 2017.
2) Next, we remove all commits that do not modify at least one `.cpp`, `.hpp`, or `.pde` file. We end up with 24,897 commits after this step.
3) We then filter the set of commits to those whose descriptions contain either of the following terms: "bug" or "fix". There are 2,213 commits with these keywords in their description. From manually trawling the commit history, we find that the majority of bug-fixing commits use at least one of these terms.
4) We then focus our attention on commits related to the ARDUPILOT's vehicle controller modules, with the exception of the ARDUSUB modules, since at present time, there exists no simulator. To identify commits related to these modules, we exploit ARDUPILOT's conventions for writing commit descriptions[7] to determine the module that was modified by the commit. After this stage, 414 commits remained.
5) Finally, we perform another round of keyword filtering, to drop all commits containing a taboo term, suggesting that the bug is not relevant to our dataset. In this stage, we remove commits we believe to be related to the build system, compilation, documentation, or cosmetic changes. A complete set of keywords, can be found in the script (included as part of our dataset). At the end, we found 333 commits that satisfied all the filters.

After automatically identifying the likely bug-fixing commits using our script, we manually inspected each commit, and discarded those that we deemed to be irrelevant to our dataset. We deemed refactorings, compilation bugs, cosmetic tweaks, and documentation changes to be irrelevant. We also excluded commits that we deemed to be improvements; this included both non-functional improvements (e.g., use of a more memory-efficient algorithm, `6da68c53`), and functional improvements (e.g., "nose of copter now points at next guided point when it is more than 10m away", `0460147a`).

To reduce the likelihood of falsely including or excluding a bug from our dataset, two of the authors, both of whom are familiar with the implementation of the ARDUPILOT platform, independently marked each commit as relevant or irrelevant; in the case that the reviewer determined the commit to be irrelevant, a reason was provided. Following this process, the reviewers unanimously agreed to remove 63 commits, and disagreed over the relevancy of 57 commits.

To settle the disputed commits, an independent party served as an arbiter, and was given the responsibility of determining the relevancy of the commit. The arbiter deemed 42 of the 57 disputed commits to be irrelevant. In total, we identified 228 commits as bug fixes. Our approach to independent bug classification is similar to that used in previous work (Martinez and Monperrus, 2015).

B. Packaging

After identifying the set of suitable bug-fixing commits in the version control history of the ARDUPILOT project, we set about packaging them into minimal Docker containers[8], capable of consistently reproducing the bug within the confines of simulation. To reproduce the bug, we follow an approach similar to that used by MANYBUGS (Le Goues et al., 2015), a widely used dataset (Le Goues et al., 2012; Mechtaev et al., 2016; Long and Rinard, 2015) of historical bugs in large-scale C programs, by using the version (i.e., commit) of the source code immediately before the bug-fixing commit.

We excluded ten of the commits from the dataset, but included them in our analysis, since they only manifest when executed on specific physical hardware (e.g., "fix LED notify during auto esc calibration"; `a3450a95`).

We have freely released the Dockerfiles used to construct the images for each bug, together with the results of our analysis, and scripts for reproducing our bug collection process.[9] Prebuilt container images may also be downloaded from DockerHub, as described in the released artifacts.

C. Characterisation

After reaching a consensus on the list of bug-fixing commits, we manually inspected each commit to determine whether the bug can be reproduced in simulation, and if so, what are the requirements for *triggering* and *detecting* it. To obtain this information, we answered the following questions:

1) **Does triggering or observing the bug rely on physical hardware?** We ask this question to determine whether software-in-the-loop simulation approaches are sufficiently capable of detecting most bugs, or whether the majority of bugs require physical hardware.
2) **Is the bug only triggered when handling concurrent events?** Parallelism and concurrent events are inherent features of most robotics systems due to their physical nature. Bugs of this nature cannot be triggered by subjecting the system to a sequential stream of commands. Automatic detection of such bugs may prove especially challenging; specification languages, such as process calculi (Baeten, 2005) and timed automata (Bengtsson and Yi, 2004), are required to describe how the system should behave under such circumstances.
   We ask this question to determine how many bugs can be triggered and detecting using simpler modelling approaches, restricted to describing the behaviour of the system in response to a sequential stream of commands.
3) **Which kinds of input are required to trigger the bug?** In most cases, inputs are essential for triggering the bug. ARDUPILOT allows inputs to be provided to the system in a number of different ways. Discrete inputs, such as

[7]Since April, 2013, almost all commits to the ARDUPILOT repository observe the following form: "*submodule*: *description*", where *submodule* describes that submodule that is modified by the commit, and *description* provides a description of the changes.

[8]https://www.docker.com/

[9]https://github.com/squaresLab/ArduBugs

preprogrammed missions and ground control commands, are more amenable to automated testing than continuous inputs, such as radio-control inputs. Bugs that require more than one kind of input place an even greater strain on testing techniques. We ask this question to determine how many bugs can be triggered using only a single discrete input type.

4) **At which stage in the execution does the bug manifest?** Bugs can occur at different points during execution; they may manifest during initialisation or normal operation, or they may occur during failure recovery and system reboot. Handling failure recovery and system reboots places additional requirements on testing approaches, and requires that failure conditions can be triggered. We ask this question to determine how many bugs can be triggered without the need to induce failure or a system reboot.
5) **Is the bug only triggered under certain configurations?** Bugs that depend on the *static* configuration of the system only manifest when the system is compiled with certain options. Similarly, bugs may only trigger under certain *dynamic* configurations of the system (i.e., parameters supplied to the system at run-time). We ask this question to determine how many bugs are only triggered under certain static and/or dynamic configurations.
6) **Is the bug only triggered in the presence of certain environmental factors?** Environmental factors include the presence of obstacles and geographical features (e.g., hills and valleys), wind and weather conditions, unreliable sensor behaviour, and the need for human interaction. Testing bugs that require such triggers places an additional burden on the simulation environment, and vastly increases the search space. We ask this question to determine how many bugs can be triggered without requiring specific environmental conditions.
7) **How does the bug affect the behaviour of the system?** Bugs can manifest in a diversity of ways, and have varying consequences on the reliable operation of the system. Characterising the exact effects of each bug is difficult, error prone, and hard to interpret. To that end, we broadly classified the effects of each bug as either *logging-related*, *behavioural*, or *(program) crashing*.
   As their name suggests, crashing bugs are known to cause the program to crash. Logging-related bugs corrupt the log files or cause incorrect status messages to be produced, but do not otherwise affect the run-time behaviour of the robot. Behavioural bugs manifest in observable changes to the behaviour of the robot; for these bugs, we also provided a qualitative description of how the behaviour of the robot is affected.
   Since the detection of certain kinds of bug require different oracles and techniques, we ask this question to assess the potential effectiveness of those techniques (e.g., fuzz testing (Csallner and Smaragdakis, 2004, 2005)).

The inspection process consisted of reading the commit description, understanding the effects of the changes made by the fix, and in some cases, executing the buggy version. To reduce the likelihood of a false label, two of the authors independently went through the list of bugs and assigned labels. The authors disagreed on more than 150 out of approximately 1,600 labels; in those cases, an independent arbiter made the final decision.

IV. Results

In this section, we present the results of our analysis on the set of bugs found in ArduPilot code repository. In total, we collected 228 bugs in three ArduPilot subsystems: 157 for ArduCopter, 50 for ArduPlane, and 21 for ArduRover.

A. Fix Characteristics

The median number of files changed by the identified bug-fixing commits is one, and the median number of line insertions and deletions, according to `git diff`, is five. 183 of the bugs were fixed by modifying a single file; this finding is encouraging for the prospects of performing fault localisation and program repair in robotics systems. The true number of line insertions and deletions related to the bug fix is likely to be lower than the observed median; a number of the commits perform extensive refactoring, unrelated to the bug.

B. Bug Characteristics

Studying the characteristics of bugs can provide us with the insight of the nature of bugs in the system and help us to improve bug detection and test-generation techniques. Below, we discuss the findings of our analysis in terms of the questions proposed in Section III-C.

(1) Does triggering or observing the bug rely on physical hardware?

In total, only 10 of 228 bugs relied on the presence of physical hardware for their detection or observation. 5 of the 10 bugs concerned platform-specific code for the robot, and thus cannot be tested using software-in-the-loop simulation. 4 of the 10 bugs affected the robot's lights and sounds; in theory, these bugs may be detected by using a higher-fidelity simulator. The remaining bug (`52c4715c`) only manifested on hardware with low memory capacity.

This finding suggests that software-in-the-loop simulation approaches are capable of detecting the majority of bugs within robotics systems. By alleviating the need for specific physical hardware, the time and cost of testing robotics systems can be reduced by using cloud-computing resources.

(2) Is the bug only triggered when handling concurrent events?

To our surprise, we determined that only 13 out of 228 bugs (5%) require concurrent events in order to be triggered. This particularly interesting result demonstrates that automated testing techniques with simple oracles, capable only of capturing the expected behaviour of sequential streams of events, may be sufficient to detect the majority of bugs in robotics systems. We believe that the lack of a need to describe parallel

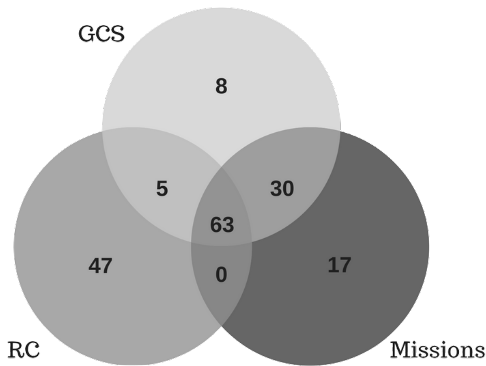

Fig. 1. The number of bugs that could be triggered by exclusively using each input type. The intersection of input types shows the number of bugs that can be triggered by any of them. Less than a quarter of the bugs rely on continuous radio-controller inputs, which are normally provided by a human-operator.

behaviours of the system reduces the specification burden on designers, and thus increases the likelihood of the acceptance of automated testing techniques.

(3) Which kinds of inputs are required to trigger the bug?

We found that 9 bugs can only be triggered by the system's command-line interface, which is used to interact with the robot when it is docked and tethered. The other 219 bugs are triggered by ground control system (GCS) commands, preprogrammed missions, and/or RC inputs. Only 7 of those 219 bugs relied on more than input type in order to be triggered (e.g., GCS commands and RC inputs). Figure 1 illustrates how many of the remaining 212 bugs could be triggered by each non-CLI input type.

Mimicking continuous radio-controller (RC) inputs, usually provided by a human-operator, is significantly more challenging than supplying the system with discrete, well-formed inputs (i.e., GCS commands, and preprogrammed missions). Encouragingly, our findings show that 165 of the 212 bugs do not rely on continuous input. Just under half of the bugs (106) can be triggered by exclusively using GCS commands, which place the least requirements on the oracle.

(4) At which stage in the execution does the bug manifest?

We discovered that 20 bugs occur when the system is in its preflight phase (i.e., initialisation), 10 manifest during its failsafe and recovery behaviours, 3 happen following a soft reboot, and 11 are encountered during the tuning phase. The remaining 184 bugs are triggered during the normal operation of the robot. Some of these execution stages are more difficult to simulate, model, and encounter. For example, testing the failsafe behaviour of the robot requires that failures can be induced during the simulation.

Importantly, the observation that almost 80% of bugs occur during the normal operation of the robot shows that automated testing techniques do not necessarily need to cover all modes of operations in order to detect the majority of bugs.

(5) Is the bug only triggered under certain configurations?

We determined that 81 of 228 bugs depend on either a particular static (53) or dynamic (20) configuration, or a combination of both (8). Knowing that more than one-third of bugs depend on a particular configuration to be triggered demonstrates the importance of testing the system under a wide range of configurations. It may be fruitful for automated testing techniques to explore the configuration space.

(6) Is the bug only triggered in the presence of certain environmental factors?

To our surprise, once again, we discovered that only 22 of 228 bugs depend on environmental factors. For example, `36634265` is only encountered in windy conditions. Other bugs, `1e2e24ee` and `436062ef`, require a human to be physically present in order to throw the plane. These bugs can be especially challenging to detect due to the demands they place on the fidelity of the simulation, and the need to test the system against a large and variegated range of environments.

Crucially, this finding shows that simpler (and implicitly more efficient) automated testing techniques that do not attempt to account for environmental factors are capable of triggering the majority of bugs.

(7) How does the bug affect the behaviour of the system?

We found that 17 bugs only cause corruption of the log files, or report incorrect status messages to the user. From inspecting the commit message, the modifications to the source code, and associated bug report, where one was provided, we determined that 6 bugs were reported to or are likely to crash. The remaining 205 bugs resulted in observable, behavioural changes to the program. Descriptions of the effects of these bugs are provided as part of our dataset.

These findings suggest that techniques such as fuzz testing are unlikely to detect many bugs. The vast majority of bugs can be detected by solely observing the run-time behaviour of the robot. Although log-checking-based techniques exclusively detect a relatively small number of bugs, incorporating log information into the oracle could allow certain behavioural bugs to be detected more easily.

V. Automated Systems Testing for Robotics

Motivated by our finding that most bugs can be reproduced using software-in-the-loop simulation without relying on complex triggering conditions (e.g., concurrent events, failure handling, environmental factors), we designed a high-level framework for performing robotics systems testing. In this section, we first provide a high-level description of this framework. We then discuss our ongoing work to realise that framework, and provide proof-of-concept examples that demonstrate how it can be used to trigger and detect bugs within the ArduBugs dataset.

A. High-Level Approach

In our high-level framework, robotics systems are modelled as a *black box*. To test a system using this paradigm, a new

blackbox is constructed for each test using a given set of *configurable parameter* values, describing the configuration of the system. The set of configurable parameters for the system is split into its *static parameters* and *dynamic parameters*. Static parameters are used to provide an abstract representation of compilation flags and launch options for the system under test (SUT). The values of static parameters are immutable and may only be specified upon the creation of the blackbox. Unlike static parameters, the values of dynamic parameters are mutable and may change during the execution of the SUT.

Once the blackbox has been constructed, a portion of the system's state, represented by its *observable variables* and the values of its parameters, may be inspected by an outside observer to produce a partial snapshot of the system's state at a given moment in time. Observable variables are used to abstractly represent information about the system that may be obtained by a (trusted) third-party over a network. For example, systems built with the popular Robot Operating System, a.k.a. ROS (Quigley et al., 2009), use a publish-subscribe model to broadcast their state to a shared topic over a network. Real-valued observable variables may have an associated *measurement noise*, which is to specify and account for the expected variance in its observed values.

Inputs are provided to the black box in the form of *events*. Events are used to describe both intended interactions with the system, in the form of *commands* (e.g., a request to move to a given position), and unintended perturbations, such as the failure of a given sensor or actuator. Each event is associated with an *event schema*. Event schemas are responsible for specifying the parameters (e.g., target location, sensor ID), if any, for all events belonging to that schema, along with their type and range of possible values. Event schemas are also responsible for describing the set of possible behaviours that are produced by the system in response to events belonging to that schema. Since the system may naturally react differently to an event under different circumstances, the event schema is responsible for providing a separate specification for each unique behavioural response (referred to as a *behaviour*).

We use our framework to test systems by subjecting them to *scenarios*. Each scenario tests how a robot with a specified initial state and configuration responds to a sequence of events that unfold in a given environment. Scenarios check the correctness of the system by comparing the observed outcomes against the expected outcomes that are given by the model.

B. Implementation

In this section, we describe an initial implementation of this framework. Our implementation, HOUSTON, named after NASA's mission control centre, is written as an open-source Python library. Within our early implementation, we attempt to produce the simplest possible realisation of our proposed high-level framework, motivated by the findings of our study. To that end, we realise scenarios as a sequential stream of discrete events, where the set of possible events is represented by a subset of the possible GCS commands. Other types of input, such as preprogrammed missions and RC inputs, are not covered; nor do we cover unintended events, such as the failure of a sensor or the interruption of a GPS signal. Dynamic and static configurations are not explicitly modelled by our implementation, but can be manually provided by an end-user.

The set of possible behaviours for each kind of event is described as set of precondition-postcondition pairs. Preconditions are used to specify the *circumstances* under which a given behaviour is applicable; at any given time, only one precondition for a particular event may be satisfied. Postconditions describe the *consequences* of executing a particular behaviour by describing the state of the system immediately after an event has finished executing. Both preconditions and postconditions are expressed in terms of the state of the system's observable variables before and immediately after the execution of the event. Convenience functions are used to account for measurement noise when specifying relationships over approximate values.

In addition to providing a precondition-postcondition pair, each behaviour specification is required to provide an accompanying *timeout function*. This function accepts the (observable) state of the robot, and the values of any parameters for that event as its input, and produces a suitable timeout for the execution of the event as its output.

Algorithm 1 describes how scenarios are executed within our framework, and how their outcomes are checked for correctness. Scenarios are deemed to be successful if each of their events is successfully completed within its computed timeout. In the event that the program crashes, the scenario is also considered to have failed. To maximise portability, and to ensure reproducibility and idempotency, our framework executes each scenario within an isolated, ephemeral Docker container. To facilitate automated testing, we provide a number of guided and unguided test generation algorithms in our framework.

HOUSTON is open-source and available to download at:

https://github.com/squaresLab/Houston

C. Preliminary Results

Here we present three examples of real-world bugs that can be triggered and detected using our framework.

1) Arming while armed: Commit `742cdf6` resolves an issue where according to the description written by the developer, sending an arm command while the copter is armed would result in the inability to arm the copter in future. For triggering the bug we must first arm the copter, before issuing a second arm command whilst it is armed. To detect the problem, we simply need to disarm the copter and arm it again. The bug will manifest when the copter is unable to arm in response to the third arm instruction. The scenario that we used in HOUSTON is as follow:

$$\texttt{ARM} \rightarrow \texttt{ARM} \rightarrow \texttt{DISARM} \rightarrow \texttt{ARM}$$

Executing the last command causes the scenario to fail since the command's postcondition is left unsatisfied (the robot is not armed); thus, the bug is detected.

```
Algorithm 1: Scenario Execution
Input: system under test : system
Input: scenario : m = SCENARIO(events, s_0, env)
Input: time between state checks : interval

executed ← [];
state ← s_0;
state' ← s_0;
PREPARE(system, s_0, env);

for a : events do
    e = EVENT(schema, params);
    state ← OBSERVESTATE(system);
    for b : BRANCHES(system) do
        if ISAPPLICABLE(b, params, state, env) then
            branch ← b
        end
    end
    DISPATCH(system, e);
    executed ← executed @ e;
    timeout ← COMPUTETIMEOUT(b, params, state, env);
    while true do
        state' ← OBSERVESTATE(system);
        if ISCOMPLETED(b, params, env, state, state')
         then
            break
        else if run out of time then
            return FAILED;
        end
        SLEEP(interval);
    end
end
return PASSED;
```

2) Unwanted motor test after parachute command: A forgotten `break` statement at the end of one of the `cases` of the `switch` statement, fixed by commit `7613964`, introduces a bug where asking the robot to use its parachute initiates an unwanted motor test (where a single motor is spun). Triggering this bug requires that the static configuration enables the system's parachute feature, and that a specification for parachute command is provided. This bug can be observed when a parachute command is sent to the copter while it is disarmed. As a result of executing the motor test, the copter arms itself (without permission from the user). Using HOUSTON we can detect this bug with this simple scenario:

`DISARM` → `PARACHUTE`

The parachute command in this scenario violates its postcondition when it accidentally arms the robot.

3) Arming in Guided mode: Commit `99ca779` fixes a bug that is both critical and simple. This bug prevents the copter from arming whilst in its "guided" mode when an arm command is issued to the copter by the ground control station. The bug is fixed by flipping the value of a boolean variable passed to the function responsible for arming the copter. Triggering this bug is as simple as switching the operating mode of the copter to "guided", before issuing a request to arm, as shown below.

`SETMODE(GUIDED)` → `ARM`

The bug is observed by noticing that the copter fails to arm after issued with an arming request.

VI. THREATS TO VALIDITY

There is a risk that our results may not generalise beyond the single system that we used as our case study. Our study goals motivated us to focus on a single system in depth, rather than performing a (necessarily) less-detailed study of multiple systems. This risk is mitigated by the rising popularity of ROS-based and Ardu* systems in the consumer market, increasing the potential utility of our results even if they do not fully generalise. While we do not have exact numbers for the number of Ardu* systems' users, we note its active GitHub development with 350 contributors and over 30,000 commits. It is also possible that the predominance of simple bugs in our data set was because it was easier for users to report those bugs and for developers to fix them. However, the fact that the software enjoys continued use and popularity suggests that the developers have fixed the the bugs that most interfered with use. Our findings are also corroborated by similar studies (Grottke et al., 2010; Cotroneo et al., 2013), discussed in Section VII, which found, for example, that relatively simple bugs predominate even in complex software. The bugs we studied were drawn from all commits; it was unclear for many whether they were discovered in the field or in simulation. Additionally, the system we studied operates on a relatively simple control loop design; systems with more complex architectures may have bugs that are less amenable to being replicated in simulation. Addressing this risk motivates future work characterising defects in more complex systems.

Our approach to identifying bugs is neither sound nor complete – a known risk in developing datasets from source control histories (Bird et al., 2009). Some bug-fixing commits do not satisfy our search criteria, and so they are excluded from the dataset. Additionally, it may be the case that certain kinds of bugs are more likely to satisfy our criteria, leading to an unrepresentative dataset. This risk is likely *more* applicable to non-cyber-physical systems, where critical live bugs can be more easily found and fixed over the course of normal development.

Similarly, there is a risk that commits were incorrectly labelled as bug fixes and non-bug fixes during the manual phase of the bug identification process. To mitigate this, we used a voting system to perform identification. Indeed, although we made our best efforts to usefully label the dataset, we may have mislabelled portions of the dataset or not chosen the most useful labels. Our approach to using consensus to classify commits as bug fixes is similar to that used by Martinez and Monperrus (2015) in their work on learning the shapes of bug-fixing patches. However, note that the previous work had all three examiners inspect each commit; we only require that a third examiner inspect a commit if the other

two examiners disagree or are unsure. We were unable to characterise 10 commits, which we subsequently dropped from the dataset. We mitigate this threat by performing three passes through the dataset and having several evaluators adjudicate disagreements, and by releasing our dataset and analysis results publicly for replication and review by other researchers.[10]

VII. Related Work

Steinbauer (2013) conducted a survey of faults exhibited by the robots that competed in the RoboCup, an international robot football championship, by asking teams from previous years' championships to complete a questionnaire. Participants were asked to report the faults within their system, and to characterise those faults according to their frequency, causes, and symptoms. Additionally, participants were asked to divide faults into those that related to *sensors*, *manipulators*, *hardware*, *software*, *algorithmic*. Steinbauer found that the most common software faults were caused by misconfiguration, a lack of timeliness, and memory leaks. Although our study shares similarities with Steinbauer's study, our focus and motivation are different; our study is exclusively concerned with determining which bugs can be discovered in simulation, and what is required to trigger and detect them.

Sotiropoulos et al. (2017) performed a study of 33 bugs in academic code for outdoor robot navigation. The study found that for many navigation bugs, only a low-fidelity simulation of the environment is necessary to reproduce the bug. Our work differs in that it has a larger reach and dataset with broader applicability, and we analyse different aspects of reproducibility.

Wienke et al. (2016) also used historical faults in the RoboCup to produce a dataset of "performance bugs" (i.e., bugs that do not prevent the robot from completing its mission, but do degrade its quality of service) for the purposes of run-time fault identification. Each fault within the dataset is provided the form of a prerecorded execution; in contrast, our dataset provides a version of the software that can be run in simulation, allowing it to be used for a greater variety of purposes. Additionally, our dataset differs to Wienke et al.'s in its composition; we focus on software-related bugs, but do not discriminate between fatal and non-fatal bugs.

Grottke et al. (2010) examined 520 faults in the onboard software systems used in 14 NASA/JPL space missions. The authors investigated and categorised the conditions under which each bug is triggered. Crucially, the authors discovered that the majority of bugs (61.4%) did not depend upon "complex" triggering conditions such as timing constraints, the sequence (or parallelism) of operations, and environmental interactions.

Cotroneo et al. (2013) conduct a similar investigation into the triggers of faults, but focus on non-robotics systems instead. Like our study and Wienke et al.'s study, Cotroneo et al. find that most bugs are "Bohrbugs" (i.e., they do not rely on complex conditions). Contrary to popular belief, Cotroneo et al. observe that the frequency of Bohrbugs does not decrease over time – their proportion remains constant. Additionally, the authors find no link between the complexity of triggering conditions and the severity of the bug as reported by the developers; i.e., bugs that are simple to trigger are just as severe as those with complex triggers. These findings suggest that the kinds of bugs that are detected by automated techniques may be just as important as those that are found during field testing.

Sahoo et al. (2010) performed a study of reported bugs in a variety of server software systems to determine the feasibility of perform automated bug diagnosis. Sahoo et al. are interested in determining whether known bugs, reported by end-users, can be reproduced using simple record-and-replay techniques, which would enable the possibility of automatically diagnosing the bug. In contrast, our study is motivated by the prospect of automatically detecting previously unknown and potentially dangerous bugs in simulation without the need for field testing.

Outside of the context of robotics, there exists a number of widely used datasets of bugs, both artificial and historical. The Defects4J (Just et al., 2014) and ManyBugs (Le Goues et al., 2015) datasets consist of historical bugs in large-scale Java and C programs, respectively. At the opposite end of the scale, the CodeFlaws (Tan et al., 2017) and IntroClass (Le Goues et al., 2015) datasets are composed of bugs in small, single-file programming assignments (or challenges) completed by novices, using C. The Software Infrastructure Repository (Do et al., 2005) represents the first concerted effort to provide a dataset of reproducible faults. Unlike the aforementioned datasets, the SIR is predominantly composed of artificial bugs, and covers programs written in a variety of different languages. These datasets have been used to conduct studies on testing (Li et al., 2007), program repair (Le Goues et al., 2012; Long and Rinard, 2015; Mechtaev et al., 2016) and fault localisation (Yoo, 2012; Moon et al., 2014; Papadakis and Le Traon, 2015; Timperley et al., 2017).

VIII. Conclusion

Robotics systems are increasingly touching the lives of the everyday consumer, both through systemic innovations like self-driving cars and via consumer-accessible systems based on ROS and other accessible, open-source technologies. However, these systems remain importantly safety-critical, even as they become more ubiquitous. Field testing is an important but enormously expensive assurance stage at which to identify key bugs. However, simulation is often perceived as providing an inadequately rich environment for identifying critical bugs before system deployment in the real world.

Our results, based on an empirical study of historical bugs in real-world robotics system, dispute this belief. We discovered that, contrary to our expectations, the majority of bugs can indeed be reproduced using software-in-the-loop simulation approaches without the need for complex triggering mechanisms (e.g., environmental conditions, concurrent events, component failures). We found that only a small minority—approximately 10%—of bugs are dependent on particular environmental

[10] https://github.com/squaresLab/ArduBugs

factors. However, we also found that continuous events, in the form of radio-controller inputs, and specific configurations are required to trigger a large number of bugs. We believe that both of these challenges, whilst difficult, can be overcome by developing specialised testing methods and leveraging and building upon existing knowledge in, e.g., testing of highly configurable systems (Kästner et al., 2012).

Motivated by these findings, we propose a high-level framework for automatically testing robotics systems, and implemented the simplest possible realisation of that framework as a proof of concept. We outline directions for developing this framework, and discuss how it can be used to detect a larger number of bugs without significantly increasing user burden.

Beyond serving as a source of historical bugs for our study, our dataset can be used to develop and compare techniques for testing, localising faults, and performing automated program repair on robotics systems. We encourage the community to use it, and publicly release our results and detailed instructions for their reproduction to support this goal.[11]

Note that our original goal was to automate the generation of bug-identifying test suites in simulation for robotics systems. However, we quickly found that constructing and evaluating such a (putative) system required an indicative set of known bugs (or a mechanism for seeding such indicative bugs) and a modeling approach that can reproduce and detect those bugs; these needs motivated this work. Fortunately, our results and framework provide promising support for automating and improving the quality assurance process for robotics systems, and present a strong economic case for the development of automated testing techniques. In addition to research in testing systems reliant on continuous events and highly configurable systems, our results motivate several additional future directions in automated testing and quality assurance for robotics:

a) Test Suite Generation: The ARDUBUGS dataset and the framework it motivates provide a mechanism to assess the effectiveness of different test generation approaches with respect to a well-understood dataset of real-world bugs and a fixed system specification. In its current form, HOUSTON implements guided (designed to maximise model coverage) and unguided methods of test generation. However, whole-test-suite generation (Fraser and Arcuri, 2011) remains a promising and underexplored avenue for testing these types of systems. We anticipate that bespoke algorithms built on such techniques will be more effective, particularly those that target the detection of bugs with particular characteristics. Techniques that target subsets of bug types, rather than targeting all possible defects, may produce optimised algorithms that are highly efficient and deliver greater confidence.

b) Modelling Unexpected Events: Robustness testing consists of checking that a system responds safely and effectively to unexpected inputs (Kropp et al., 1998). In the current realisation of our automated testing framework, the set of possible events consists solely of commands that are sent by the user to the drone (via the ground control system). To allow a greater number of bugs to be detected, we plan to extend the set of possible events to include a subset of those that are outside of the user's control. These events could be used to represent the failure of a sensor, an unresponsive servo, or a (simulated) loss of communications with the GCS. Including such *unexpected events* allows our framework to be used to ensure that the robot fails in a safe and predictable manner.

c) Specification Languages and Inference: Our results motivate more powerful specification languages that, e.g., express constraints on the order and timing of events and model concurrency. Temporal logics (Pnueli, 1977) or timed automata (Alur and Dill, 1994) may provide a mechanism to accomplish the former; event calculi (Shanahan, 1999), reactive automata (Crochemore and Gabbay, 2011), and process calculi (Baeten, 2005) may apply to the latter. Striking an appropriate balance between the expressive power of the specification language, and the burden of using it remains an open and significant challenge. One way to reduce the specification burden is to focus our attention on ROS-based systems, an increasingly popular framework for robotics systems[12], and to develop a background theory that does most of the heavy lifting; this may enable a simple DSL for describing behaviour.

Additionally, rather than requiring manual system specifications, specification mining techniques (Ernst et al., 2001; Nguyen et al., 2017; Aliabadi et al., 2017) could be used to automatically infer likely specifications based on observational data. Given information about the types and (meaningful) parameters of different events, and the set of observable system variables, we plan to use HOUSTON to generate and execute a set of maximally diverse scenarios. Both static and dynamic inference techniques may be applicable, with the latter working over, e.g., the traces of timestamped system observations produced by HOUSTON (Ernst et al., 2001; Beschastnikh et al., 2016). Outlier traces that invalidate an otherwise consistent specification could be flagged for evaluation by a human-operator for assessment, with additional annotations fed back into mining. Ideally, such systems can produce high-quality specifications to aid in the debugging of robotics systems, with minimal human input required.

In conclusion, the findings of our study strongly support the idea of applying cheap, simulated-based testing approaches to the problem of detecting bugs in robotics systems. We call on the research community to join us in exploiting our findings to develop a new generation of highly effective testing techniques.

IX. ACKNOWLEDGEMENTS

This research was partially funded by AFRL (#FA8750-15-2-0075) and DARPA (#FA8750-16-2-0042); the authors are grateful for their support. Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government.

[11] https://github.com/squaresLab/ArduBugs

[12] https://spectrum.ieee.org/automaton/robotics/robotics-software/ros-robot-operating-system-celebrates-8-years

REFERENCES

2016. Schiaparelli Landing Investigation Makes Progress. (2016). http://www.esa.int/Our_Activities/Space_Science/ExoMars/Schiaparelli_landing_investigation_makes_progress Accessed Mar. 1, 2018.

Maryam Raiyat Aliabadi, Amita Ajith Kamath, Julien Gascon-Samson, and Karthik Pattabiraman. 2017. ARTINALI: Dynamic Invariant Detection for Cyber-physical System Security. In *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering (ESEC/FSE '17)*. 349–361.

Rajeev Alur and David L Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126, 2 (1994), 183–235.

J. C. M. Baeten. 2005. A Brief History of Process Algebra. *Theoretical. Computer Science* 335, 2-3 (May 2005), 131–146.

Johan Bengtsson and Wang Yi. 2004. *Timed Automata: Semantics, Algorithms and Tools*. Berlin, Heidelberg, 87–124.

Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. 2016. Debugging distributed systems. *Queue* 14, 2 (2016), 91–110.

Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. 2009. Fair and Balanced?: Bias in Bug-fix Datasets. In *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. 121–130.

D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi. 2013. Fault triggers in open-source software: An experience report. In *International Symposium on Software Reliability Engineering (ISSRE '13)*. 178–187.

Maxime Crochemore and Dov M Gabbay. 2011. Reactive automata. *Information and Computation* 209, 4 (2011), 692–704.

Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience* 34, 11 (Sept. 2004), 1025–1050.

Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' Crash: Combining static checking and testing. In *International Conference on Software Engineering (ICSE '05)*. 422–431.

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering* 10, 4 (1 Oct. 2005), 405–435.

M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (Feb 2001), 99–123.

Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In *International Conference on Quality Software (QSIC '11)*. 31–40.

April Glaser. 2017. Watch Amazon's Prime Air make its first public U.S. drone delivery. (2017). https://www.recode.net/2017/3/24/15054884/amazon-prime-air-public-us-drone-delivery Accessed Mar. 1, 2018.

M. Grottke, A. P. Nikora, and K. S. Trivedi. 2010. An empirical investigation of fault types in space mission system software. In *Dependable Systems Networks (DSN '10)*. 447–456.

K. Henningsson and C. Wohlin. 2004. Assuring fault classification agreement - an empirical evaluation. In *International Symposium on Empirical Software Engineering (ISESE '04)*. 95–104.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *International Symposium on Software Testing and Analysis (ISSTA '14)*. 437–440.

Christian Kästner, Alexander Von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. 2012. Toward variability-aware testing. In *International Workshop on Feature-Oriented Software Development*. ACM, 1–8.

Nathan P. Kropp, Philip J. Koopman, and Daniel P. Siewiorek. 1998. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing (FTCS '98)*. 230–239.

Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (December 2015), 1236–1256.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

Alan Levin. 2017. Alphabet Drones Will Deliver Burritos in Australia Test. (2017). https://www.bloomberg.com/news/articles/2017-10-16/drones-to-deliver-burritos-in-australia-as-alphabet-begins-tests Accessed Mar. 1, 2018.

Z. Li, M. Harman, and R. M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering* 33, 4 (April 2007), 225–237.

Z. Liu and P. Mei. 2014. Automated testing for large-scale critical software systems. In *International Conference on Software Engineering and Service Science (ICSESS '14)*. 200–203.

Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE '15)*. 166–178.

Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of

automated program fixing. *Empirical Software Engineering* 20, 1 (01 Feb 2015), 176–205.

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE '16)*. 691–701.

S. Moon, Y. Kim, M. Kim, and S. Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *International Conference on Software Testing, Verification and Validation (ICST '14)*. 153–162.

ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-guided Approach to Finding Numerical Invariants. In *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering (ESEC/FSE '17)*. 605–615.

Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

Amir Pnueli. 1977. The temporal logic of programs. In *Foundations of Computer Science (FOCS '77)*. IEEE, 46–57.

Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*. 5.

S. K. Sahoo, J. Criswell, and V. Adve. 2010. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *International Conference on Software Engineering (ICSE '10)*. 485–494.

Murray Shanahan. 1999. The event calculus explained. In *Artificial intelligence today*. 409–430.

Susan Elliot Sim, Steve Easterbrook, and Richard C. Holt. 2003. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *International Conference on Software Engineering (ICSE '03)*. 74–83.

Thierry Sotiropoulos, Hélène Waeselynck, and Jérémie Guiochet. 2017. Can Robot Navigation Bugs Be Found In Simulation? An Exploratory Study. In *Software Quality, Reliability and Security (QRS '17)*. 150–159.

Gerald Steinbauer. 2013. *A Survey about Faults of Robots Used in RoboCup*. Berlin, Heidelberg, 344–355.

Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In *International Conference on Software Engineering (ICSE '17 Poster)*. 180–182.

Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. 2017. An Investigation into the Use of Mutation Analysis for Automated Program Repair. In *Search Based Software Engineering (SSBSE '17)*. 99–114.

Johannes Wienke, Sebastian Meyer zu Borgsen, and Sebastian Wrede. 2016. A Data Set for Fault Detection Research on Component-Based Robotic Systems. In *Towards Autonomous Robotic Systems (TAROS '16)*. 339–350.

Leigh Williamson. 2008. IBM Rational Software Analyzer: Beyond Source Code. In *Rational Software Developer Conference (RSDC '08)*.

Shin Yoo. 2012. Evolving Human Competitive Spectra-Based Fault Localisation Techniques. In *Symposium on Search Based Software Engineering (SSBSE '12)*. 244–258.