Quality Assurance Automation in Autonomous Systems*

Afsoon Afzal
Carnegie Mellon University
Pittsburgh, PA, United States
afsoona@cs.cmu.edu

ABSTRACT

Robots and autonomous systems are finding their way to interact with the public and failures in these systems could be extremely expensive, even deadly. However, low-cost software-based simulation could be a promising approach to systematically test robotics systems and prevent failures as early as possible. In our early work, we showed that the majority of bugs could actually be reproduced and discovered using low-fidelity simulation environment. We created a high-level framework for automated testing of popular ArduPilot systems. In this work, I propose novel approaches to automatically infer powerful representation of system models, and generate test suites with the purpose of enhancing automated fault localization performance and describing the root cause of failures. Finally, I propose to use those novel approaches to inform the construction of automated program repair techniques for autonomous systems.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Software safety**;

KEYWORDS

automated quality assurance, autonomous systems

ACM Reference Format:
Afsoon Afzal. 2018. Quality Assurance Automation in Autonomous Systems. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3236024.3275429

1 INTRODUCTION

For many years, the application of robots and autonomous systems were limited to industrial settings. But there are no doubts that robotics and autonomous systems are increasingly involved in peoples' everyday lives in recent years. For example, companies have started testing autonomous vehicles on public roads. Besides transportation, robots are commonly used in health care and medical operations, autonomous delivery, education or even serving coffee.

One thing is now evident: failures in these systems can be very expensive, and even deadly. In March 2018, a fatal crash by an autonomous car took the life of a woman in Tempe, Arizona. As these autonomous systems increasingly influence safety of the public, it becomes critical to investigate and develop effective quality assurance methods to prevent failures as early as possible.

As distributed systems, quality assurance in these systems face particular challenges. First, validation of robotics systems commonly takes place using unit testing, component testing and field testing. Although unit tests can detect and presumably prevent failures at the method or module level, they are not particularly effective in distributed context where multiple components need to interact with each other. Field testing plays a vital role in robot validation and is the main method used to find critical issues. However, it is extremely expensive. The fatal car crash is only one example of such an event. Accordingly, testing and validation using simulation environment would be a promising approach to minimize the cost of failures. ExoMars Lander crash in October 2016 is a remarkable example where the incident could have been prevented with proper simulation testing.[1]

However, testing and validation of distributed systems in simulation is non-trivial. First, specifying and determining the expected behavior of the system under different configurations and setup is challenging especially when the notion of correctness becomes fuzzy and unclear. For example, when a robot is instructed to go to location L, is it only considered a correct behavior if it arrives at *exact* coordinates of L? How close to L would be close enough to be accepted as correct behavior?

Secondly, triggering and detecting bugs in robotics systems could depend on factors that are not inherently captured in simulation. For instance, an extreme windy environment may be required for a defect to be manifested. However, most existing simulators are not advanced enough to simulate all aspects of environment and the physical hardware. In addition, concurrent events and parallel processes in distributed systems make it difficult to trigger defects that depend on thread interleaving and timing constraints.

Thirdly, even if a bug is successfully triggered, it may not manifest until some later point in the execution. Even if a misbehavior is correctly detected, finding the root cause of the failure requires tremendous manual effort. All aforementioned reasons explain why it is burdensome to systematically test these systems in simulation.

My long-term goal is to discover and develop powerful methods to automatically detect, localize and fix defects in real-world autonomous systems using low-fidelity, software-based simulation. In this paper, I propose ideas that will implement automated quality assurance methods which are not easily applicable to autonomous and distributed systems. The results of this research will encourage

*This work is funded by NSF (#CCF-1563797) and CMU Presidential Fellowship; the author is grateful for the support.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3275429

[1] http://www.esa.int/Our_Activities/Space_Science/ExoMars/Schiaparelli_landing_investigation_makes_progress

and support roboticists to systematically validate robotic systems in simulation, long before field testing takes place, reducing the cost and danger of any failures.

2 RELATED WORK

Automated testing and test generation for traditional software has been around for decades. However, few studies have been done in the area of autonomous systems. One of the earliest works on this subject uses model-based testing approach and specifies a meta-model of both the entities of the system under test (SUT) and interaction scenarios [4]. It applies a meta-heuristic search to automatically generate test data representing complex situations. In a drone collision avoidance system, an evolutionary test generation finds challenging situations and corner cases [9]. Similarly a recent study generates test scenarios for autonomous systems which expose situations where transitions in performance mode of the system takes place [6]. All aforementioned research require manual specification of the system's model and focus on generating a generic test suite that could be used for regression, targeting corner cases and complex circumstances. Building upon them, I propose to automate inferring system models and specifically target test generation for automated fault localization and program repair purposes.

Spectrum-based fault localization (SBFL) is among the first automated fault localization techniques to be proposed [2]. The suspiciousness of each line in program is calculated based on the number of failing and passing test cases that execute the line. SBFL is still widely used in automated program repair community for debugging traditional software.

3 PRELIMINARY WORK

Even though autonomous systems' simulation may not be 100% compatible with reality, it still provides valuable information about the behavior of the SUT. However, most roboticists believe that real-world field testing is the only way to find major flaws in the system. To assess this common belief, I, in collaboration with others, collected a dataset of historical bugs in highly popular open-source ArduPilot[2] project [7].

ArduPilot provides a framework for autonomous control of a variety of vehicles including drones, planes and even submarines. It accepts a variety of inputs from the user such as radio channel (RC) input, ground control system (GCS), command line interface (CLI) and predefined missions. Using GCS, the user can instruct the ArduPilot system to autonomously undertake a command such as going to a location or taking off the ground. In addition to its wide range of inputs and subsystems, the rich version-control history and popularity of ArduPilot made it a suitable system to be considered as our case study.

ArduBugs: After applying automated and manual analysis on thousands of commits, we identified 228 historical bugs in ArduPilot system to create the ArduBugs dataset.[3] Plus, we made these bugs easily reproducible by including Docker containers for each bug. We manually investigated every individual bug in this dataset to respond to 7 questions showed in Table 1. These questions best reflect the characteristics of bugs that impact their manifestation in simulation environment. Reasons behind selecting these questions and the methodology of conducting the analysis are demonstrated in details in [7]. Overall, we found that the absence of high-fidelity simulation environment with support of complex mechanisms only stops 45% of bugs from reproduction. For example, we could only identify 10 bugs in the dataset requiring physical hardware for manifestation. Similarly, only 9.6% of bugs are dependent on environmental factors such as wind or obstacles. The empirical study supports our hypothesis that the majority of these bugs can actually be detected using low-fidelity simulation.

The findings of our initial study and availability of a comprehensive dataset of robotics bugs encouraged us to take the next step: a framework for test generation and execution. To automatically assess the quality of SUT, we first need to design a framework in which we can instruct the system to perform a number of commands and observe its behavior for failure detection. We created Houston: a high-level framework for testing ArduPilot systems [7].[4]

Houston: Overall, given the specifications of the SUT over each possible action in terms of pre- and post-conditions, Houston executes sequences of commands and asserts satisfiability of specifications. As an example, specifications for a single action of `TAKEOFF` command for a drone are as follows:

$$(\texttt{SUT.IS\_ARMED} \wedge \texttt{SUT.ALTITUDE} = 0) \Rightarrow \texttt{SUT.ALTITUDE\_AFTER} = \texttt{TAKEOFF\_PARAMETER}$$

This action is taken only if SUT is armed and resides on the ground at the moment of receiving the command. If the preconditions are satisfied, then the action is expected to take place. As a result, in the final state of the system, after finishing the command, the drone should have taken off the ground and the altitude should be approximately the same as the parameter given to the command as specified by the post-condition.

In our initial attempt, we were able to reproduce a few bugs in ArduBugs dataset using Houston by manually determining the sequence of commands that will trigger the buggy code [7]. Thus we confirmed given the SUT's specifications, Houston can successfully perform different scenarios in simulation and assert correctness of the SUT's behavior according to the provided specifications.

4 PROPOSED FUTURE WORK

The preliminary work suggests opportunity in application and advantages of automated quality assurance in the field of robotics. Figure 1 depicts an overview of the overall structure of Houston that I envision to achieve my goals. In Section 4.1, I propose to elevate the reasoning power of Houston and automatically infer SUT's specifications. In Section 4.2, I propose to generate targeted test suites with the purpose of increasing the precision of automated fault localization techniques. Finally, in Section 4.3, I propose to apply state-of-the-art automated program repair techniques on robotics systems and advocate for improvements in that area.

4.1 Empowering Houston

Even though Houston has shown promises in test execution and fault detection in ArduPilot systems, it demands additional improvements and advancements:

[2] http://ardupilot.org
[3] https://github.com/squaresLab/ArduBugs
[4] https://github.com/squaresLab/Houston

Table 1: Questions and possible options that specify bug characteristics impacting their manifestation in simulation.

Question	Options
Does triggering or observing the bug rely on physical hardware?	Yes, No
Is the bug only triggered when handling concurrent events?	Yes, No
Which kinds of input are required to trigger the bug?	CLI, GCS command, RC input, Missions
At which stage in the execution does the bug manifest?	Initialization, Normal Operation, Failsafe
Is the bug only triggered under certain configurations?	None, Static, Dynamic, Both
Is the bug only triggered in the presence of certain environmental factors?	Yes, No
How does the bug affect the behavior of the system?	Logging-related, Behavioral, Software crash

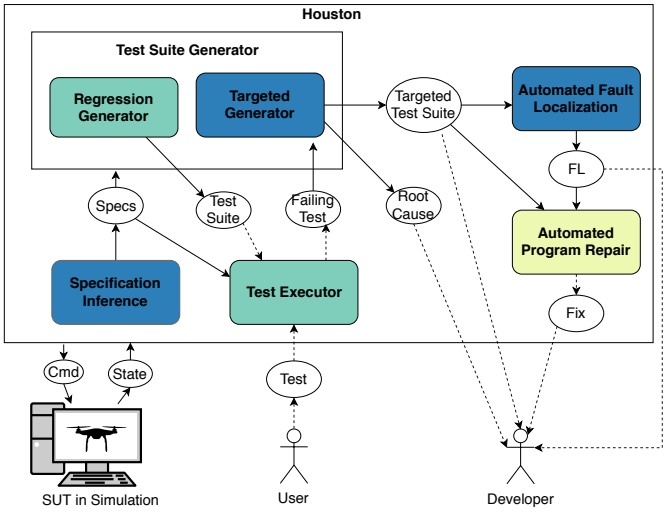

Figure 1: The overall structure and flow of information in Houston. Modules are represented by rectangles, information by ovals, mandatory input/output by solid arrows and optional input/output by dashed arrows. Green modules are already implemented, blue ones are included in this proposal and the yellow one is left for future studies.

Specification language: the current specification language of Houston is very limited. It is only capable of specifying system's behavior in terms of observable variables and parameters and it monitors system state as snapshots. I propose to express more powerful specifications on the time and order of events, possibly in continuous manner. As an example, when the robot receives the `GOTO` command to a destination, it should not only travel to the destination safely, but also it should never exceed the maximum speed allowed. Looking at the snapshots of the system and environment, before and after the command takes place, does not allow for specification of such requirement. Logics with higher expressive power such as temporal logic can accomplish the intended. The current specification language also suffers an inability to model concurrency, which can possibly benefit from event and process calculi or reactive automata. I will carefully assess the possible options, first, based on their added value to the power and flexibility of specification language. Second, the portion of bugs in ArduBugs dataset that could be captured with application of the option and finally, the burden on the users in terms of manual effort.

Automated specification inference: The significant burden of writing specifications for the system on developers is undeniable. Even with the presence of a powerful, expressive specification language, it is still highly difficult to gather an accurate and complete set of specifications for the system. Since my intention is to automate QA as much as possible, I propose to take advantage of specification mining techniques to automatically infer the likely specifications of the system. Based on observations of many correct executions and normal behavior, it is achievable to collect likely invariants of the system [1]. Inconsistent performance of the robot according to the likely invariants could be reported for manual inspection. Returning the feedback of manual review of the behavior to Houston can gradually improve its set of true specifications.

4.2 Automated Test Suite Generation

Generation of high-quality test suites for regressions is a well-studied area, and as mentioned in Section 2 there are tools and techniques currently available which could be applied to the domain of robotics and be tailored to best fit the system's requirements. Similar to other techniques, Houston currently implements guided test suite generation methods to maximize model coverage as well as unguided methods such as random test generation. Although extending Houston to use more intelligent methods such as evolutionary algorithms for test generation is an option for future direction, it will not be the focus of my proposal.

Instead, I propose to focus on targeted test suite generation with the ambition to investigate root causes of a failure in the system. Most of the users of robotics systems are not professional programmers and when they observe a failure, they report the issue, rarely accompanied by the log files, screenshots or videos to the developers. The developers then have to manually reproduce the failure or inspect the log files to find the possible root causes. Note that in these systems the failure can actually happen long after the faulty code is executed making it difficult to diagnose the problem.

It becomes more convenient to debug an issue if more instances of the failure are available. In addition, having passing tests that are somehow similar to the failure can further elaborate the circumstances impacting the failure. I propose to extend Houston that given a test case, it first trims events to exclude the ones unrelated to the failure. Secondly, Houston should generate a test suite that will find more instances of the failure as well as similar tests that successfully pass. For example, a user may observe her drone crashing onto the ground while instructing it to land automatically. She can reproduce this failure using Houston and report the issue attaching the test for investigation by developers.

Meanwhile, HOUSTON can take the failing test case, exclude all aspects of it unrelated to the failure and generate more examples of both failures and successes in the similar conditions and come up with an explanation of why such failure arises. For example, it can discover that the failure only occurs when the altitude of the drone at the time of landing is more than 8 meters.

For achieving this goal, I propose to use delta debugging [8] and symbolic execution [3] techniques. Given a sequence of commands resulting to failure, delta debugging can be used to determine essential ones for the failure. In our earlier example, let's assume the following test case is provided by the user (for simplicity, only the altitude parameter of the commands are shown):

$$\texttt{DISARM} \rightarrow \texttt{ARM} \rightarrow \texttt{ARM} \rightarrow \texttt{TAKEOFF}(alt : 5) \rightarrow \texttt{GOTO}(alt : 3) \rightarrow \texttt{GOTO}(alt : 9) \rightarrow \texttt{GOTO}(alt : 8) \rightarrow \texttt{LAND}$$

Even though this example reproduces the failure, it includes a few commands irrelevant to the problem. Applying delta debugging to this sequence will result in the following:

$$\texttt{ARM} \rightarrow \texttt{TAKEOFF}(alt : 5) \rightarrow \texttt{GOTO}(alt : 9) \rightarrow \texttt{LAND}$$

This test still fails and reproduces the issue but now it is more clear what the possible reason could be. Furthermore, I would like to make conclusions about the order of commands essential for the failure. If A followed by B leads to a failure, it is constructive to understand whether B followed by A has the same effect or not.

Next, I will use symbolic execution to generate tests with the same sequence of commands but different parameters, to explore all possible actions and behaviors of the system (which is modeled by HOUSTON) under the provided commands. Considering all parameters and environment settings as symbolic variables, we can explore the search space and narrow down the range of parameters and settings which result in the failure. For example, consider the parameter to TAKEOFF as symbolic variable α and parameter to GOTO as β. Using symbolic execution, HOUSTON can automatically generate the following specifications for the failure:

$$(\alpha \geq 8) \bigvee (\alpha < 8 \bigwedge \beta \geq 8)$$

These logical phrases basically inform the developer that the fault only occurs when the drone is higher than 8 meters. I anticipate that the generated constraints will be significantly valuable to the developers and assist them to resolve the issue more quickly with less effort. As evaluation, I need to conduct a user-study to measure effectiveness of the proposed methods.

4.3 Fault Localization and Automated Repair

Although spectrum-based fault localization techniques are highly popular (Section 2), we suspect they perform poorly on robots and distributed systems. The assumption underlying SBFL is that the faulty code is more likely to be exclusively executed by failing tests. However in robotics systems, multiple components and modules concurrently and continuously run together in the system to perform the tasks, and the faulty code can affect system state in a way that the fault is captured long after being triggered. Therefore, fault-localization could report tens or hundreds lines of code in the project as equally the most suspicious. Inspecting a long list of possible fault locations, even if it includes the real faulty code, is not feasible for or even helpful to the developers.

I anticipate that the test suite with more instances of the failing and passing tests evolved around the original failure (Section 4.2), can actually escalate performance of fault localization in these systems. In addition, mutation-based fault localization [5] could be suitable as a technique to narrow down the most suspicious parts of code to a smaller set. The accurate fault localization not only benefits the developers to inspect the issue, but also opens up the opportunity to use automated program repair. Automated program repair techniques are directly affected by the performance of fault localization and without an accurate one, they will not be able to find a fix to the problem or will propose a low-quality fix, over-fitting to the provided test suite. As evaluation, I will compare the performance of fault localization, using test suite generated by HOUSTON with developer-provided tests and test suites generated by other techniques.

If automated fault localization on robotics systems reaches an acceptable performance, I will apply popular automated program repair tools to this domain to assess their performance in generating fixes for the defects. I propose to analyze the results, distinguish major deficiencies and possibly make suggestions for improvements.

5 CONCLUSION

In conclusion, I target applying automated quality assurance on autonomous systems using low-fidelity software simulation to reduce the cost of failures. I first propose to extend our high-level robot's testing and automated test generation framework, HOUSTON, to automatically infer systems specification using a more powerful and flexible specification language. Secondly, I propose a novel technique to generate test suites with sole purpose of enhancing performance of automated fault localization methods and conceiving a description for the possible root causes of the defect. Finally, I propose to evaluate popular automated program repair techniques on these systems and examine their shortcomings in this domain.

REFERENCES

[1] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE)* 27, 2 (2001), 99–123.

[2] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *International Conference on Software Engineering (ICSE)*. 467–477.

[3] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.

[4] Zoltán Micskei, Zoltán Szatmári, János Oláh, and István Majzik. 2012. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*. Springer, 504–513.

[5] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 153–162.

[6] Galen E Mullins, Paul G Stankiewicz, and Satyandra K Gupta. 2017. Automated generation of diverse and challenging scenarios for test and evaluation of autonomous vehicles. In *International Conference on Robotics and Automation (ICRA)*. IEEE, 1443–1450.

[7] Christopher Steven Timperley, Afsoon Afzal, Deborah S Katz, Jam Marcos Hernandez, and Claire Le Goues. 2018. Crashing simulated planes is cheap: Can simulation detect robotics bugs early?. In *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 331–342.

[8] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 253–267.

[9] Xueyi Zou, Rob Alexander, and John McDermid. 2014. Safety validation of sense and avoid algorithms using simulation and evolutionary search. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 33–48.