2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S) | 978-1-6654-3566-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/DSN-S52858.2021.00028

The Importance of Safety Invariants in Robustness Testing Autonomy Systems

Milda Zizyte[1] Casidhe Hutchison[1] Raewyn Duvall[2] Claire Le Goues[3] Philip Koopman[4]

***Abstract*—Common testing approaches can involve unit tests that test for functionality, or robustness tests that check for software crashes. In the case of autonomy systems, we posit that robustness testing must involve invariant checking of safety properties to be effective, because passing unit tests and the absence of software crashes do not make a sufficient safety case. In this work, we present empirical data that shows a large proportion of bugs found in mature systems would not have been discovered without invariant checking; and, in most cases, represent physical safety violations of the systems. We also discuss common invariant violations we have seen while testing systems, to serve as a starting point for future testing efforts. We conclude that invariant checking is indeed important for autonomy systems and recommend that autonomy system test teams write safety specifications and invariant checkers in order to help assure that their projects are tested for safety.**

***Index Terms*—robustness, autonomous systems, software quality, software testing, safety invariants**

I. Introduction

Robotics and autonomy software makes up a core component of cyber-physical systems. Due to the harm that these systems may potentially cause to persons and their environments, these software components must be considered safety-critical. To help ensure that such systems will not exhibit unsafe behavior, projects typically include validation, in the form of testing. Functional testing (e.g. unit and integration testing) verifies that the system is behaving according to behavioral requirements, while robustness testing is used to check that the system does not violate safety requirements, even in the presence of unexpected inputs. Our testing team at Carnegie Mellon University's National Robotics Engineering Center developed novel testing techniques for robustness testing autonomy systems and tested more than 25 autonomy systems over the last 10 years [1].

Typical software robustness testing checks whether or not certain inputs can cause software crashes [2]. However, merely checking for crashes is an insufficient safety case for autonomy systems. Because robotics systems are cyber-physical, any robustness testing effort of systems must check for violations of safety properties, such as speed limits. Such requirements are typically defined in terms of invariants, or computable properties of a system that must always hold true in order for a system to be considered safe.

Invariants are used in model checking [3], [4] and runtime verification [5] to verify and validate system safety. Runtime verification techniques exist to check invariants when performing software robustness testing [1], [6], but are only useful if system developers actually provide meaningful invariant descriptions that correspond to the safety requirements of a system. In the majority of systems we tested, a testable safety specification was not part of the system design documents.

Safety standards for autonomy systems, such as UL 4600 [7], require fault injection testing, which software robustness represents a portion of; specifically representing some sensors faults, network faults, or behavioral faults from untrusted software components. UL 4600 also requires the use of run-time monitoring for checking the correctness of responses to faults and the validity of the safety case assumptions. We argue that the two should be used in conjunction to provide safety assurances about system behavior.

In this paper, we advocate that writing and checking for invariants must be a part of the testing effort in developing autonomy system software. We first introduce the testing framework and the mature systems from commercial and defense industries that we tested (Section II). We then present a brief statistical analysis of the faults found during the testing of these systems (Section III), and follow this with a discussion of common types of invariants, paired with several illustrative examples of violations found in open source systems (Section IV). With these quantitative and qualitative examples, we argue that checking of safety invariants is both necessary and achievable for ensuring the safety of autonomy software.

II. Testing Framework

To demonstrate the importance of invariants in the robustness testing of autonomy software, we summarize our testing framework and highlight several systems from our work testing autonomy systems [1], specifically those systems that were tested using invariant checking.

A. Testing tool

We have previously discussed the challenges inherent to testing autonomy systems [1], and how we designed a tool to

NAVAIR Public Release 2021-153. Distribution Statement A Approved for public release; distribution is unlimited

This work was done under the RIOT program, which was funded by the Test Resource Management Center (TRMC) and Test Evaluation/Science & Technology (T&E/S&T) Program and/or the U.S. Army Contracting Command Orlando (ACC-ORL-OPB) under contract W900KK-16-C-0006.

We would like to acknowledge the rest of the RIOT and ASTAA project teams, whose work testing autonomy systems made this possible

[1]National Robotics Engineering Center, Carnegie Mellon University mzizyte and fhutchin@nrec.ri.cmu.edu

[2]Carnegie Mellon University, ECE rduvall@andrew.cmu.edu

[3]Carnegie Mellon University, SCS clegoues@cs.cmu.edu

[4]Edge Case Research pkoopman@ecr.ai

978-1-6654-3566-6/21/$31.00 ©2021 IEEE
DOI 10.1109/DSN-S52858.2021.00028

effectively robustness test autonomy software. Our approach mutates field values in messages passed along the system's internal network, in order to test how the software responds to unexpected inputs. We perform this mutation during log replay, which ensures that the message sequence is valid for normal operation of the system. These base logs, which we refer to as **scenarios**, are recorded from system demos or unit/integration tests provided with the System Under Test (SUT). Logs recorded during a test can be used to calculate whether an invariant holds during the run of a system.

B. Systems tested

While we discussed the many proprietary systems tested in the prior work, in this paper we focus specifically on those that were tested using invariant checking. Because of confidentiality, we cannot disclose many details on these systems, but they serve as important examples of how testing with invariant checking is important in the industry. System A is an unmanned ground vehicle autonomy platform, System B is a search and rescue robot, Systems C and D are middleware layers for robotics systems, and System E is an human assistance robot. All these systems are Technology Readiness Level (TRL) 6 or higher, meaning that they all have been validated in a simulated or real-world operation environment.

We also discuss several more recently tested open source systems, built on the Robot Operating System (ROS)[1]. The open source systems we discuss in this paper are Niryo One[2], which is a manipulation robot; Turtlebot[3], a rover-type development platform; Fetch[4], a warehouse assistant robot; and ArduPilot[5], an autonomous drone platform.

For all of these systems, robustness tests were generated by mutating system runs captured by running unit/integration tests and/or provided examples, such as the Disco demonstration from the Fetch documentation, which moves the arm through a trajectory of points.

C. Invariants used

Depending on the system, we check for various **simple** (such as limit checks and liveness) and **complex** (those having a stateful or temporal component) invariants. Several examples are given in Section IV. While a software crash may also be described as an invariant violation (the invariant being "software shall not crash"), for the purposes of clarity in this paper, we refer to invariants as safety properties beyond the absence of software crashes.

III. Empirical analysis

We examine the number of unique bugs found in several systems tested that would not have been found without an invariant checker. For the mature systems we tested, we classify a "unique bug" to correspond to a single complete, diagnosed bug report submitted to the developers of the system. For the rest of the bugs, we estimated that a failure found in a unique testing scenario (e.g. a unique example of behavior), triggered using a mutated value on a unique input field (e.g. "velocity" being a unique input from "position"), was a unique bug.

A. Proprietary Systems

In our historical testing of mature systems, we only tested one system with a provided concrete safety specification: System A. As shown in Table I, 22 out of 27 unique bugs were only found with a simple or complex invariant checker. Eleven of these invariant violations were from complex invariants.

For the other systems tested, no explicit system safety specification was provided. This partly motivates our paper, because testing may have missed safety properties that the developers intended but did not communicate. By putting in our own effort, however, we were able to glean some simple safety invariants from the functional specification or from nominal system behavior, such as message publication frequency. All of the unique bugs found in systems B and D were invariant violations, and one out of the three bugs in system C was an invariant violation. Of the bugs that were reported to developers, all were fixed. This shows that common patterns can apply to various autonomy systems, and should serve as encouragement that testers can build useful, if incomplete, safety specifications with less effort than may be feared.

B. Open-Source Systems

In all of the open source systems tested, there existed unique bugs that were found using invariant checks, as show in Table I. In Turtlebot and ArduPilot, bugs were found that violated the speed limits defined in the documentation. In Fetch, the discovered invariant violations were for violations of the expected message broadcast frequency. All of these violations present safety risks, because they cause the systems to behave outside of the specified physical behavior. Speed limit violations impact stopping distance and magnitude of momentum in the case of a collision, and broadcast frequency violations impact the ability of a system to stop or correct itself in time.

C. Discussion

Our investigation shows that a large number of bugs found when robustness testing autonomy systems can only be discovered when checking safety invariants. Every system tested exhibited at least one safety failure that would have been missed by standard software crash checking. We showed that, even when an explicit safety specification is not available, it is still possible to write invariant checkers that find non-trivial safety bugs in a system. While a system can be designed to fail safe in the event of a software crash, violations of safety invariants such as speed limits pose an explicit and unmitigatable risk to safety. Even in the case of a broadcast frequency violation, where the link to safety is not direct, the presence of such a failure invalidates the timing analysis of

[1] http://ros.org
[2] http://niryo.com/niryo-one
[3] http://turtlebot.com
[4] http://fetchrobotics.com
[5] http://ardupilot.org

	System A	B	C	D	E	ArduPilot	Fetch	TurtleBot	Niryo
Total Bugs	27	2	8	1	3	16	12	5	1
Total Invariant	22	1	4	1	1	2	4	2	1
Complex Invariant	11	1	1	0	0	0	0	0	1

TABLE I
INVARIANT CLASSIFICATION FOR BUGS IN PROPRIETARY AND OPEN SOURCE SYSTEMS

a system that a safety case is built on. Limiting testing of autonomy software to crash detection leaves an unacceptable hole in the safety arguments of autonomy systems.

IV. COMMON INVARIANT VIOLATION EXAMPLES

We suspect that lack of domain knowledge is one of the major barriers to entry to invariant-based testing. In this section, we show compelling examples of invariant violations we have found while testing autonomy systems. We discuss how these invariants are defined, what the consequences of violating these invariants are, and how to implement checkers for them. While invariants generated from templates such as this are not a substitute for an explicit set of safety requirements, they do help find potential safety risks beyond software crashes.

A. System Timeliness

Timing analysis represents an important part of the safety analysis of a system. Therefore, validating the robustness of system timing is an important check when robustness testing. In the case of periodic system components, checking the timeliness of the output involves verifying that time between messages does not violate a specified minimum broadcast frequency. For service-oriented (aperiodic) components, timeliness is about not exceeding the maximum response time. If a component is not meeting its timing requirements, system responsiveness as a whole is impacted, and system safety can no longer be guaranteed. For example if a planning node fails to respond in time to a new obstacle, it may not be possible to avoid collision, even when the system would have been able to respond appropriately if timing requirements were met.

Both broadcast frequency and response time are easy to check by examining message timestamps. Even in cases where the frequency is not explicitly listed in the specification, estimating timing bound by checking for large deviations from nominal behavior (e.g. triple the maximum nominal time between messages) can help find violations of system timing assumptions. In our testing we found violations of both explicit (in System B) and estimated (in System A and Fetch) timing requirements. In System A, further diagnosis of violations of estimated timing requirements lead to the discovery of potential infinite loops.

B. Limit Checks

Autonomy systems generally have safety limits on their physical components, such as an actuation angle limit or a speed limit. Violation of these limits can result in damage to the machine or potential risk to system operators. We encountered speed limit violations in the open source ArduPilot and Turtlebot systems.

We found a violation of angle limits in the Fetch system. When malformed values were sent on the `MoveGroup/Goal` message, the arm was sent a position that over-articulated several joint angles, exceeding the specification for these joints in the system documentation.

Notably for speed limits, negative or `NaN` inputs to the command speed caused an invariant violation in multiple systems. This leads us to believe that, even in commercially released systems, speed checks on the system may have been naively implemented as `! (speed < limit)`.

Another common pitfall of checking a speed limit is that it is not sufficient to merely verify that the $x-$ and $y-$ direction linear speeds obey the limit, but that the length of the velocity vector itself does so as well. In fact, we found violations in the nominal case in the ArduPilot system, where the individual components of the velocity did not exceed the speed limit, but the actual speed did.

Actuation and speed limits must be specified for most cyber-physical systems that are capable of self-displacement. Furthermore, most robotics systems usually output odometry and joint state messages that already report system actuation for use in control loops. Because limit violations pose serious danger if uncaught, checking these properties is critically important for checking the safety of autonomy systems. Furthermore, the relative ease of writing a speed and angle limit invariants means that there is no excuse for failing to check these properties.

C. Self-collision

A self-collision invariant violation occurs when some articulated component of a robot collides with another component of the robot. We found a self-collision bug in the open source Niryo One system, where an invalid input on the `/joy` ROS topic caused the robot's arm to collide with its body. This is shown in Figure 1. When this invariant is violated outside of simulation, it may result in permanent and costly damage to the robot. In fact, such an invariant violation was discovered in System E and reported to the development team, but it went unfixed and eventually did occur during use of the system, indeed resulting in irreparable damage to an end effector.

Checking this invariant is a little more complex than checking limits, because it involves defining bounding boxes around the components of the robot and using the simulator to check if these boxes intersect. However, violation of this invariant can cause significant damage to the robot, or require expensive repairs to any hardware fail-safe mechanisms. Because it is very likely that one safety requirement of a robot with an articulating arm is for that arm not to hit the robot, effort

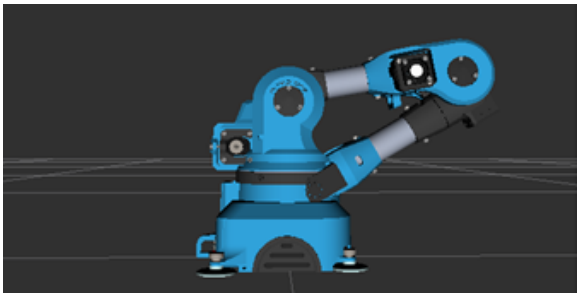

Fig. 1. Niryo arm self-collision

should be put into writing this invariant checker for testing any robot where self collision is a potential hazard.

D. More complex safety cases and discussion

The examples above show how even simple invariants can be violated in production systems, and how these violations may present more direct safety risks than just software crashes. Even so, they serve merely as a starting point for thinking about invariants that can be used for testing autonomy systems. A healthy project development process for autonomy systems must include a well-defined safety specification [8]. This safety specification may include more complex rules, involving temporal and stateful requirements, such as a robot being required to reach full stop within two seconds of the emergency brake being engaged.

In general, invariants for testing should come from the safety requirements and physical limitations of the system. For the open source systems that we tested, we directly referenced the online documentation to derive our invariants. Beyond the examples discussed above, pattern repositories, such as Dwyer et al.'s "Property Specification Patterns" [9] and Kane's specification patterns [10], provide starting points for thinking about checkable safety properties. We hope that these examples provide guidance for engineers to write thorough and quantifiable safety specifications when designing their systems, for testers to write thorough invariant checkers, and for the two parties to communicate.

V. Conclusion

In this paper, we argue that checking invariants that test for safety property violations is imperative for testing autonomy software. The typical approach of using unit tests that check for functional correctness, or robustness tests that only check for software crashes, is necessary, but not sufficient for ensuring system safety. This is because autonomy systems are cyber-physical systems that have temporal and physical requirements which are often part of the safety requirements of the systems.

We have shown empirical evidence that, when testing the robustness of autonomy software, checking safety invariants allowed robustness testing to find more bugs than could be found when checking solely for crashes. In some systems, a majority of bugs would not have been found without an invariant checker. Additionally, every system tested exhibited at least one violation of safety properties, but not all exhibited software crashes. These bugs may be even more severe than software crashes in terms of cost or liability, because, by definition, they put the system in an unsafe state. This leads us to conclude that checking solely for software crashes is not sufficient for testing the robustness of cyber-physical systems, and that robustness testing must include the use of invariant checkers to sufficiently meet the safety requirements of autonomy system software.

Safety standards for autonomy systems require both fault injection testing and run-time monitoring. While the use of run-time motoring in conjunction with fault injection is not required, using run-time monitoring to provide assurances that safety goals are still being met in the presence of faults provides substantial evidence towards a safety case, without incurring significant extra cost. Our work shows that doing so is effective for autonomy systems, as long meaningful invariants from the safety specification are used when run-time monitoring.

We described common invariants that can apply to a variety of autonomy systems, and referenced more complete libraries of safety requirement patterns. Testers and developers should use these resources as a starting point for their own systems. Ultimately, a safety-oriented testing and development setup for an autonomy system project involves considerations to safety at all steps, starting with writing a safety specification that can be used to write safety invariants for thorough testing. Testing-based validation of a well-developed safety case is an important part of guarding against disastrous failures when a robotics system is deployed.

References

[1] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. Le Goues, and P. Koopman, "Robustness testing of autonomy software," in Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. ACM, 2018, pp. 276–285.

[2] P. Koopman, K. DeVale, and J. DeVale, "Interface robustness testing: Experience and lessons learned from the ballista project," Dependability Benchmarking for Computer Systems, vol. 72, p. 201, 2008.

[3] L. Fix, "Fifteen years of formal property verification in Intel," in 25 Years of Model Checking. Springer, 2008, pp. 139–144.

[4] K. Y. Rozier, "Linear temporal logic symbolic model checking," Computer Science Review, vol. 5, no. 2, pp. 163–203, 2011.

[5] A. Kane, O. Chowdhury, A. Datta, and P. Koopman, "A case study on runtime monitoring of an autonomous research vehicle (arv) system," in Runtime Verification. Springer, 2015, pp. 102–117.

[6] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," SAE International Journal of Transportation Safety, vol. 4, no. 1, pp. 15–24, 2016.

[7] U. Laboratories, UL 4600: Standard for Evaluation of Autonomous Products, ser. Standard for safety. Underwriters Laboratories, 2020.

[8] P. Koopman, Better embedded system software. Drumnadrochit Education, 2010.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in Proceedings of the 21st international conference on Software engineering, 1999, pp. 411–420.

[10] A. Kane, "Runtime monitoring for safety-critical embedded systems," Ph.D. dissertation, Carnegie Mellon University Pittsburgh, PA, 2015.