# Robustness Inside Out Testing

Deborah S. Katz[1] Milda Zizyte[3] Casidhe Hutchison[2] David Guttendorf[2] Patrick E. Lanigan[2] Eric Sample[2]

Philip Koopman[3] Michael Wagner[4] Claire Le Goues[1]

***Abstract*—Robustness testing is an important technique to reveal defects and vulnerabilities in software, especially software for Unmanned Autonomous Systems (UAS). We present Robustness Inside Out Testing (RIOT) as a technique directed at finding failures in autonomy systems that are able to be activated from external interfaces. The technique consists of four main steps: unit-level robustness testing, generalization, permeability analysis, and activation. Each of these steps yields a valuable deliverable in the testing process, and, when applied in succession, expands a unit-level bug to an external interface. RIOT has the following advantages over traditional robustness testing: it finds faults faster, it can find faults missed by traditional approaches, it identifies faults that can be triggered from inputs at an external interface, and it produces useful artifacts to aid in fault diagnosis and repair. In this paper, we outline each step of the RIOT process and provide an example of RIOT finding a bug on a real system that would not have been discovered using existing techniques.**

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

## I. Introduction

Robustness testing has proven to be an effective way to find important vulnerabilities that threaten dependability in Unmanned Autonomous System (UAS) software [1]. However, while robustness testing at the unit level often reveals vulnerabilities in deeply-buried software modules, prior work does not present a scalable or automated way to determine which of these vulnerabilities can lead to actual system failures and thus need urgent fixes. Robustness testing at the system level, by contrast, often requires more time and may fail to discover internal vulnerabilities. In this paper, we summarize the Robust Inside Out Testing (RIOT) project, which addresses these shortcomings by finding unit-level failures (“inside”) and assessing whether those defects can be activated from external interfaces (“out”).

RIOT’s objective is to increase robustness testing’s ability to find defects deep within complex UAS software, specifically those that can be activated from external inputs. To do so, RIOT begins with unit-level robustness testing and uses a suite of chained tools to determine whether any discovered faults can be triggered via external interface. This approach benefits from the speed of unit-level robustness testing, while providing information about vulnerabilities from a full system perspective. The steps in the RIOT process generate results that are informative to developers for debugging and bug triage.

Compared to traditional testing approaches, RIOT

- finds faults faster;
- finds faults missed by traditional approaches;
- prioritizes faults by highlighting those that can be triggered through an external interface; and
- produces useful artifacts at each step to aid in the process of diagnosing and repairing faults.

This project was funded by the Test Resource Management Center (TRMC) Test and Evaluation / Science & Technology (T&E/S&T) Program through the U.S. Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI) under Contract No. W900KK-16-C-0006, “Robustness Inside-Out Testing (RIOT).” DISTRIBUTION STATEMENT A - Approved for public release; distribution is unlimited. NAVAIR Case #2020-129.

[1]School of Computer Science, Carnegie Mellon University `dskatz` and `clegoues@cs.cmu.edu`

[2]National Robotics Engineering Center, Carnegie Mellon University `fhutchin`, `dguttendorf`, `planigan` and `esample@nrec.ri.cmu.edu`

[3]Deptartment of Electrical and Computer Engineering, Carnegie Mellon University `milda` and `koopman@cmu.edu`

[4]Edge Case Research (work performed while at National Robotics Engineering Center) `mwagner@edge-case-research.com`

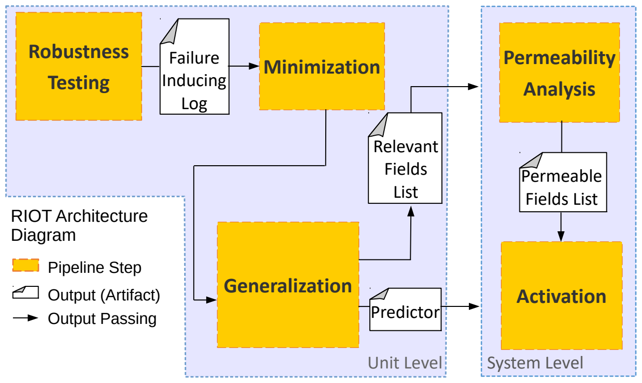

Fig. 1. The RIOT architecture diagram.

## II. Procedure

Figure 1 shows the general architecture of the RIOT system. RIOT begins with *unit-level robustness testing*, which is discussed in Section II-A. Robustness testing reveals failures in a software module and, for each failure, produces a log of the input messages that triggered the failure. The *generalization* step, explained in Section II-B, includes three sub-steps: *minimization*, *relevant field identification*, and *extrapolating a unit-level rule* that describes the failure-triggering inputs. Overall, generalization extracts a general rule (predictor) that explains the properties of the failure-inducing log that cause the unit to fail. For example, a predictor might be that a failure can be triggered when a particular input value is very large.

The *backchaining* process then attempts to discover a set of system-level inputs that cause this unit-level input rule to be satisfied, resulting in a failure. *Permeability* analysis, discussed in Section II-C, determines which system-level inputs affect the values of the fields relevant to the unit-level failure. *Activation*, discussed in Section II-D, finds values for the identified system-level inputs that result in the unit-level inputs satisfying the previously-identified rule.

When the RIOT toolchain is successful, the final product is a log of system-level input messages that can be replayed to activate a unit-level fault discovered in robustness testing.

## A. Robustness Testing

*Robustness testing* is a variant of software testing that evaluates robustness, or "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [2].

The robustness testing portion of RIOT is built on the Automated Stress Testing for Autonomy Architectures (ASTAA) technique that our team previously developed [1], which in turn, draws on earlier work on the Ballista project [3], [4]. These approaches use the core idea that exceptional values, such as `null` or `NaN`, can cause failures when used as inputs. To this end, the approach maintains a dictionary of exceptional values for each parameter type, gathered over years of experimentation. An ASTAA test-run exercises a representative execution of an autonomy system while injecting exceptional values into a portion of messages passed among system components. When ASTAA causes a module to fail, it generates a log containing all messages sent during that execution. These logs can be used to replay the execution and reproduce the test failure.

RIOT robustness testing applies the same testing approach at the unit level. This is much faster than testing a whole system, which often involves the time-consuming processes of spinning up a simulator or field testing on real hardware [5]. This means that more tests can be run in the same amount of testing time. Additionally, applying these techniques at the unit level may reveal bugs in software units that are not easily encountered when testing at the system level, even in cases where the fault can be activated from the system interface. See Section III for an example of such a fault.

Usefully, the output of robustness testing is a message log that can be replayed to reproduce the failure and assist in debugging. The results of a suite of robustness tests can also be used in summary, as a metric for the robustness of a system component. For example, a high failure rate can indicate low component robustness.

## B. Generalization

The next step, generalization, takes a failure-inducing log found in robustness testing and extracts a rule that more generally describes the unit-level inputs that induce the failure.

Generalization involves three steps: (1) minimization, (2) identification of relevant fields, and (3) extrapolation of a unit-level rule to describe the conditions, or input values, on the module that must be present in order for the violation to occur.

*a) Minimization:* Minimization, previously described in Hutchison et al. [1], starts with the failure-triggering log from the robustness testing step. It uses delta debugging [6] to produce a minimal set of log messages necessary to trigger the failure during replay. This minimal message log is much smaller that the original log, usually only containing a few messages. This allows tests to be run much faster, as well as reducing the number of messages that need to be considered in subsequent steps.

*b) Identification:* Our team previously outlined the procedure for identifying relevant fields in a failure-inducing log [1], [7]. Test logs can have very high dimensionality, both in (a) the number of messages they contain and in (b) the number of fields in each message. Rule learning can still be difficult or even intractable with a minimized log due to the large numbers of fields within each message. The ASTAA project found that many bugs could be activated by manipulating only a single field or several fields within a single message. Thus, identifying these fields provides more information about the test case and reduces the search space when extrapolating a unit-level rule.

The RIOT process uses a portion of the Hierarchical Product Set Learning (HPSL) algorithm [7] as a method for identifying relevant fields. This algorithm identifies both (a) fields whose exceptional values caused the fault, and (b) those fields whose nominal values are necessary for the fault to be exercised. For example, an exceptional `speed` value may cause a safety violation, but only when the `mode` field is also set to a certain nominal value. The HPSL algorithm would identify both of these fields as relevant, as both fields must take on specific values for the fault to be triggered.

This list of fields is necessary for the next step of generalization. It also highlights the variables that cause the fault in the software unit, which can be extremely useful for developers trying to track down the source of a bug.

*c) Extrapolation:* The final step of generalization is to find the unit-level rule that describes the fault-triggering input conditions. This is done by running versions of the minimized log, where the values of the relevant fields are systematically altered. This exploration is more efficient due to the shorter test case runtime of the minimized log and the reduced search space of the relevant fields.

We use a variety of approaches to learn the input constraints, including the latter part of the HPSL algorithm and decision trees [7]. While other learning algorithms could apply, we have chosen those that generate rules that are easily human interpretable, e.g., as a discrete set of values or range of bounds on a field.

This unit-level rule, in addition to being a useful description of the input conditions that activate the fault for debugging, also provides a large goal area for the RIOT process to aim towards when trying to activate a fault from an external interface. While a single unit-level input pattern discovered during robustness testing may not be reachable by manipulating system inputs, other unit-level input patterns that satisfy the generalization pattern may be reachable.

This step produces a *predictor*, which outputs whether a given unit-level test case would trigger the fault if replayed. This predictor is used in *activation* (in Section II-D) for finding a system input that will trigger the fault.

## C. Permeability Analysis

We begin the process of backchaining — or trying to find a system-level input that results in the fault-triggering inputs at the unit level — with permeability analysis. Permeability finds relationships between system-level input values and the values at the input to the unit where the fault occurs. In other words, permeability identifies which external inputs, if any, can be manipulated in order to change the values of the relevant fields identified during generalization.

Permeability analysis in UASs is complicated because the systems are often noisy and non-deterministic [8]. Rather than relying on direct causality, RIOT's permeability strategies use statistical and machine learning approaches. By applying a variety of different perturbations to system inputs, we use analysis techniques for time series data to get a summary of behavior of an internal system value under varying input patterns. By comparing the behavior of the internal system value with system inputs perturbed against the behavior of the value with fixed system inputs, we determine if perturbing a given system input affects that internal value.

While this technique will determine that there exists a relationship between a system input and a relevant internal value, it cannot determine the nature of that relation. Due to the noisy, non-deterministic, and temporal nature of these relationships, the relationship may not even be expressible. For this reason, we cannot simply calculate the system-level input values that could cause internal values to satisfy the rule found during generalization, and thus need the final step, activation.

## D. Activation

Activation completes the process of backchaining. Activation starts with the relevant system inputs found in permeability analysis and analyzes the System Under Test (SUT) to determine if there are values that can be sent to those external system interfaces that can produce internal values that match the unit-level activation rule, thereby triggering the failure.

To determine the values needed at the external interface, RIOT uses an optimization algorithm, which attempts to minimize the distance between the values that the internal inputs take during an experiment and the values those internal inputs need to take to satisfy the rule found during generalization. For the purposes of the optimization, the rule is represented as the surface of a polytope towards which the optimization algorithm can drive the experiments by changing the values of the relevant system-level inputs. In other words, we try to minimize the distance to the fault activation, by manipulating system inputs.

This step produces a replayable log of system-level inputs that activates the fault. A test case provides a strong argument that the fault is severe enough to need to be fixed.

## III. Example

To illustrate how the elements of RIOT can be used to find a bug that would not be found using previous robustness testing techniques at the external interface level, we present an example on the Clearpath Robotics Husky robot [9]. This is an autonomous rover that plans and executes a path.

By using *robustness testing* to manipulate the input to the `move_base` node, we found a test case that causes the node to crash. The input we mutated was a recording of one of the provided example runs of the system, filtered to only the few thousand messages that were inputs to the `move_base` node.

Given this failure-inducing test case, we used *generalization* to efficiently determine the rule for the input values that would trigger the crash. First, *delta debugging* reduced the mutated input log to a single `/map_updates` ROS message [10]. Next we identifed the *relevant fields* for triggering the bug: `/map_updates.x` and `/map_updates.y`. Using machine learning algorithms, we extrapolated a *unit-level rule*: `move_base` crashed when the product of `/map_updates.x` and `/map_updates.y` was negative or very large.

From the generalized rule for crashing the `move_base` node through the internal interface, we needed to determine if this bug could be manifested in actual robot operation, i.e., if it could be triggered by external sensor inputs. This *backchaining* happened in two steps: we first determined that the values of the `/map_updates.x` and `/map_updates.y` fields are affected by changing some external interface values, and then determined whether these external values could be changed in a way to steer `/map_updates.x` and `/map_updates.y` towards a negative or very large product.

*Permeability* showed that there were several external fields caused that affected `/map_update.(x,y)`, including the velocity `/odom.twist.linear.(x,y)` and the goalpoint `goal.(x,y)`.[1] We then used *activation* to find if a combination of values on these inputs would cause `/map_update.(x,y)` to have a negative or very large product, in turn causing the `move_base` node to crash. Indeed, we found that a set of goalpoints in the lower region of the map triggered this crash. This shows that the fault can be activated from an external interface and may warrant a high prioritization for repair.

The crash zone for these inputs is a very specific range: $|\texttt{goal.x}| < 100$ and $-100 < \texttt{goal.y} < -80$, as shown in Figure 2. This is because the bug was not activated outside of the map region, since the robot refuses to drive off the map. Traditional dictionary-based testing at the external interface level would not have tried this combination of values, because these values are largely valid values and thus do not exist in the dictionary. Fuzz testing would require aproximately 60,000 tests to have a 90% chance of hitting this region, assuming

[1]While the `/map_updates` inteface exists in the Husky codebase, it is ignored in favor of sending a full `/map` message. To demonstrate backchaining, we made a minor modification to the codebase to use the `/map_updates` interface instead. Because the `/map_updates` interface is not used, the fault is not activatable from an external interface in the original codebase.

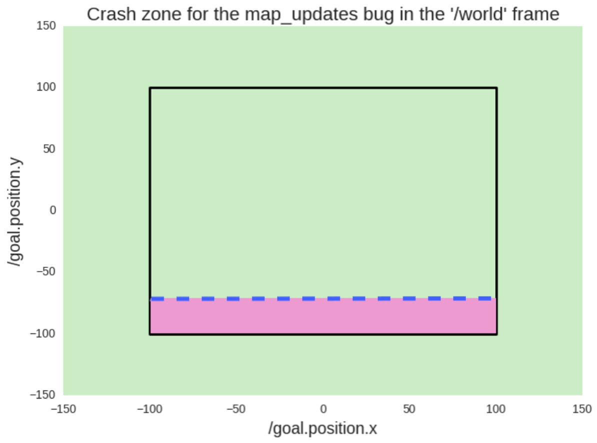

Fig. 2. The region in which the Husky fault occurs: here shaded pink, below the blue dotted line but within the map region, defined by the solid black line.

that the relevant system inputs are already known. Therefore, this bug is a prime example of how robustness testing at the unit level, generalizing the internal bug, and back-chaining to external interfaces can discover sophisticated input patterns for triggering bugs that can be missed when using traditional robustness testing at the system interfaces.

## IV. Conclusion

The RIOT toolchain is a set of testing and diagnosis techniques, designed to detect unit-level faults and work backwards to expose them at the system level. It makes use of efficient message-based robustness testing techniques to uncover vulnerabilities deep within a system, and follows on with testing-driven approaches and machine learning to find additional information about a fault. At each step, RIOT is able to provide incrementally more information to the developer about a defect. The toolchain culminates in an external threat assessment, identifying which defects can be activated at the system interface level. RIOT's layered approach finds faults efficiently and can find faults that are missed by traditional system-level testing. In finding if faults can be exercised externally, it highlights important faults for debugging.

RIOT's procedure allows it to find defects with input values that would not have otherwise been tested. These techniques have the potential to find faults faster than many existing techniques, such as field testing or fuzz testing whole systems. These techniques also have the capability of finding faults that would likely be missed by existing approaches — such as in the example (Section III) in which RIOT finds a fault that would have not have been found by dictionary- or fuzz-based approaches at the external interface level.

To the best of our knowledge, little previous work tackles stress testing robotic or autonomy systems, and much previous work focuses on hardware. A historical analysis of robotic system failures indicates that while attention to hardware is warranted, software remains one of the major sources of error [11]. Chu [12] performed practical robustness testing of a middleware layer for autonomy systems. Among other observations, they noted the difficulty of testing large hierarchical multi-component systems. The RIOT approach is specifically directed at complex UAS architectures.

By using a suite of testing tools, RIOT can provide fault detection and some fault diagnosis and triage capabilities while reducing load on developers. By creating user-interpretable results at each step in the pipeline, RIOT can provide utility even when a limited testing budget does not allow for execution of every step. While the final system-level diagnosis steps (permeability and activation) typically require a simulator or other framework for executing the whole system, all prior steps can be performed by simply executing a single software unit. This means that RIOT can provide useful results even in cases where a full system simulation is unavailable.

Due to the automated nature of these testing tools, they are highly scaleable and benefit from parallel execution of test cases. The number of robustness tests that can be executed in a given timeframe, for example, scales linearly with the number of test execution threads. In cases where algorithms within RIOT are not trivially parallelizable, such as in tree-based searches, parallelization can still reduce overall testing time at the cost of increasing the number of total test executions. RIOT can thus scale well even in a high performance computing or cloud environment.

RIOT's automation, scalability, and utility at every step in the pipeline make it an ideal tool for detecting faults in software dependability at little expense to a development team.

## 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 *International Conference on Software Engineering - Software Engineering in Practice*, ser. ICSE-SEIP '18, 2018, pp. 276–285.

[2] J. Radatz, A. Geraci, and F. Katki, "IEEE standard glossary of software engineering terminology," *IEEE Std 610121990*, 1990.

[3] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *Fault-Tolerant Computing*, ser. FTCS '98, June 1998, pp. 230–239.

[4] P. Koopman, K. DeVale, and J. DeVale, *Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project*, 2008, ch. 11, pp. 201–226.

[5] A. Bihlmaier and H. Wörn, "Robot unit testing," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 255–266.

[6] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '99, 1999, pp. 253–267.

[7] P. Vernaza, D. Guttendorf, M. Wagner, and P. Koopman, "Learning product set models of fault triggers in high-dimensional software interfaces," in *Intelligent Robots and Systems*, ser. IROS '15, 2015, pp. 3506–3511.

[8] C. Pecheur, "Technical report: Verification and validation of autonomy software at NASA," Hanover, MD, USA, August 2000.

[9] Clearpath Robotics. Husky unmanned ground vehicle. Accessed: 2020-01-24. [Online]. Available: http://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/

[10] ROS. Ros.org — powering the world's robots. Accessed: 2020-01-24. [Online]. Available: http://ros.org

[11] J. Carlson and R. R. Murphy, "How UGVs physically fail in the field," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 423–437, June 2005.

[12] H.-N. Chu, "Test and evaluation of the robustness of the functional layer of an autonomous robot," Ph.D. dissertation, Institut National Polytechnique de Toulouse - INPT, Sep 2011. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00627225