Better Robustness Testing for Autonomy Systems

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Milda Zizyte

B.S., Computer Engineering, University of Washington
B.S., Mathematics, University of Washington
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May 2020

Abstract

Successfully testing autonomy systems is important to decrease the likelihood that these systems will cause damage to people, themselves, or the environment. Historically, robustness testing has been successful at finding failures in traditional system software. Robustness testing uses a chosen test value input generation technique to exercise the system under test with potentially exceptional inputs and evaluate how the system performs. However, assessing coverage for a given input generation technique, especially in black box testing, is tricky. Past work has justified new input generation techniques on the basis that they find a non-zero number of failures, or find more failures than other methods. Simply measuring the efficacy of these techniques in this way does not consider the complexity or uniqueness of these failures. No strongly justified metrics of comparison or systematic ways to combine test value input generation techniques have been introduced.

In this dissertation, we explore two main robustness testing input generation techniques: fuzzing and dictionary-based testing. These techniques represent two different ways of sampling the possible input space for a given parameter. Fuzzing can theoretically generate any value, but may generate wasteful test cases due to the size of the sample space. Conversely, dictionary-based testing may closer match the distribution of failure-triggering inputs, but is restricted in scope by the predetermined values in the dictionary. By introducing metrics to compare these techniques, we can highlight how these tradeoffs manifest on actual systems.

To perform this comparison, we have created an approach to test autonomy systems and apply both test input generation techniques to an assortment of systems. We introduce the comparison metrics of efficiency and effectiveness, and show that both test methods have areas of strength, weakness, and similar performance. By delving deeper into the reason for these differences and similarities, we justify combining the test input generation techniques in a hybridized way. We propose various hybrid testing methods and evaluate them according to our metrics of comparison.

We find that dictionary-based testing, followed by fuzzing, performs the best according to our metrics. We show that this happens because of a path dependency in testing, that is, deeper bugs cannot be found until fragile fields are eliminated from testing. We discuss how both of our metrics were necessary to reach this insight. We also include general insights from testing autonomy systems, such as low dimensionality of failure-triggering inputs. Our recommendations of testing frameworks, test input generation techniques, test case selection strategies for a hybrid testing method, and metrics of evaluation can be used to test robotics software effectively and efficiently in the future, which is a step toward safer autonomy systems.

Acknowledgments

This work would not have been completed without the help and support of many people. I would like to thank Phil Koopman, my advisor, for providing me with many opportunities for growth. Thanks also goes to my dissertation committee, namely, Lujo Bauer, Claire Le Goues, and Dan Siewiorek, for their guidance. Additional thanks to Nathan Snizaski, who was always available for help navigating the specifics of the degree requirements.

Some would say that the true Ph.D. treasure is the friends gained along the way. Special thanks to Henry Baba-Weiss, Casidhe Hutchison, and Thom Popovici. Thank you to Henry for maintaining a strong friendship over all these years and 2,500 miles of distance. Thank you to Cas for offering strong professional and emotional support since the first day of graduate school. And, thank you to Thom for all of the beer and concert breaks, and for always being a good listener. Further thanks goes to everyone else I met in graduate school, whether in the shared office space (John Filleau, Aaron Kane, Peter Klemperer, Anuva Kulkarni, Joe Melber, Malcolm Taylor, Richard Veras, and Zhipeng Zhao); as the biggest guys in the gym (Eric Gottlieb, Max Li, Zach McDargh, Mary Story, and Erik Trainer); or through various groups and activities (Ben Cowley, Kirstin Early, Susan Grunewald, Harlin Lee, Erin McCormick, Vince Monardo, Ben Niewenhuis, Rony Patel, and Nicole Rafidi).

Thanks also goes out to the great team at the National Robotics Engineering Center (NREC). Thanks again to Cas Hutchison for her help, and to Bill Drozd, Dave Guttendorf, Anne Harris, Deby Katz, Pat Lanigan, Adi Nemlakar, Zach Pezzementi, Eric Sample, Matt Schnur, Steve Stawarz, Trenton Tabor, Angela Wagner, Mike Wagner, and Mollie Wild. The knowledge and opportunities I gained at NREC were truly valuable.

Finally, I would like to acknowledge the sponsorships and grants that enabled my research. This dissertation was written with funding from the Robustness Inside Out Testing (RIOT) project. NAVAIR Public Release 2020-301. Distribution Statement A – Approved for public release; distribution is unlimited. RIOT is 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. The National Science Foundation Graduate Research Fellowship supported me for the first half of my degree. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-125252. Finally, some work was done under the Automated Stress Testing for Autonomy Architectures (ASTAA) project. ASTAA was supported 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-11-C-0025, “Stress Testing for Autonomy Architectures (STAA)”.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Effectively testing autonomy software is important to decrease the likelihood that these systems will cause damage to people, themselves, or the environment when the systems are released. A software fault, or a bug, is a section of code that, when executed, can result in erroneous behavior [96]. Robustness testing exercises systems using unexpected inputs to uncover some software faults, and has historically been successful at finding some software faults in traditional software systems [72]. While functional testing checks whether the implementation of software conforms to its promised requirements, robustness testing is mostly concerned with finding whether the software exhibits unexpected behavior in general. In this dissertation, we introduce some challenges to robustness testing in general, and discuss how these challenges are made even more difficult while testing autonomy systems. These challenges lead to several research questions, which motivate the contributions for this dissertation.

For even moderately-sized software projects, developers cannot guarantee bug-free code [15]. A methodical software development process can help reduce bug creation, but developers also rely on testing to validate some degree of functionality and robustness in their products. However, a long testing process in the development cycle limits profitability because it delays a product going to market [90]. This is why project managers impose testing budgets and why testers seek efficient testing techniques, i.e. those that uncover the most high-priority faults in a fixed amount

of time. Put otherwise, tests that do not find faults are costly to the development cycle, and undiscovered faults are costly to the market success of the product. Even safety-critical systems, which have a high development budget, benefit from efficient testing. A testing approach that can provide efficient fault coverage provides a degree of assurance that the system is less likely to malfunction.

The inherent problem with robustness testing is that, even with small sets of input parameters, it is impossible to exhaustively test the input space. The success of a test method is often measured by the number of bugs it has found, but because it is impossible to fully know the number of bugs that the test method did not find, this measure is mostly valuable relative to other test methods. By comparing multiple input selection techniques, and explaining the differences and similarities in the bugs they find, we form a better approximation of how well each input selection technique performs. This can be used as a metric of relative bug coverage for each test method. Furthermore, a deeper understanding of why the test methods perform differently can inform a hybrid test method that outperforms either method. To do these comparisons and evaluate the strengths of proposed hybrid methods, we define metrics of comparison that reflect the goals of a software team testing effort to find as many unique bugs in a system as possible for a given test budget.

1.1 Problem and scope

This dissertation addresses the problem of making robustness testing recommendations for autonomy systems. The research questions are:

1. What are useful metrics for comparing robustness testing techniques?
2. What are the tradeoffs between using fuzz testing and dictionary-based testing on autonomy systems, as evaluated by these metrics?
3. What modifications to the testing techniques and testing parameters are relevant to the autonomy systems under test?

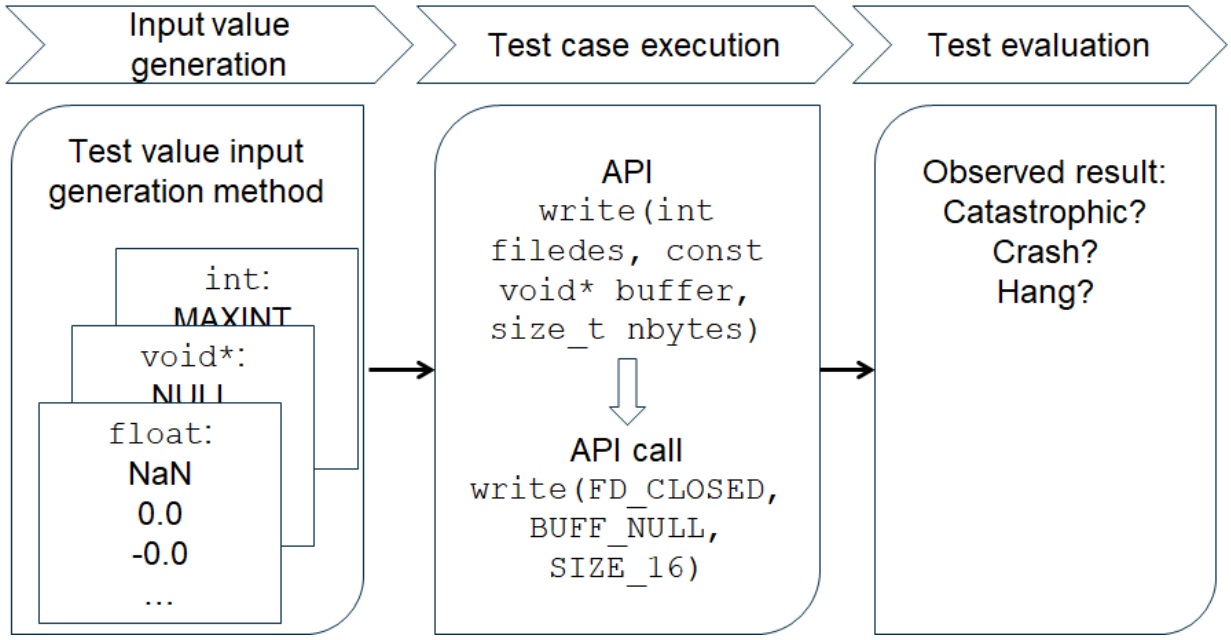

Figure 1.1: Traditional software robustness testing approach

4. How can we combine testing approaches into a portfolio of tests to achieve better testing outcomes than using any single testing approach alone?

5. What general recommendations can be made for testing and writing autonomy systems based on our testing results?

1.1.1 Systems under test

Traditional software robustness testing provides unexpected inputs to a system under test, usually by calling functions in the system's interface with test values as the parameters. This traditional approach therefore usually deals with APIs, where a test case is a single function call with a limited number of parameters (see fig. 1.1). The result of a test depends on whether the function call crashes or hangs, or the system has a catastrophic failure. This application space of robustness testing already has several challenges, as stated above and in Chapter 2. Furthermore, when the software in question does not match the simple model of a function call, testing becomes even more challenging.

We focus on autonomy systems, due to the increasing focus on robotic safety, and because the large input space, temporal requirements, and cyber-physical limitations pose novel questions for testing research [54]. We note that even systems that are not formally classified as "safety critical" have safety requirements, such as speed limits or prevention of self-collision, because these systems exist in the physical world. In this dissertation, our testing framework is primarily applied to three Robot Operating System (ROS)-based autonomy systems in simulation. While we do not claim that our robustness testing results will scale exactly to other systems, the goal is to demonstrate different testing technique outcomes and how they can be used in conjunction with each other for a better overall test campaign. We describe the use case scenarios for the systems under test and identify the interfaces for testing as well as safety invariant requirements to determine if the systems exhibit bugs.

1.1.2 Testing framework and input selection

Our testing framework builds on previous work in traditional software robustness testing and testing of autonomy systems. We delve deeper into the framework in Chapter 3, and note that, while the previous work in autonomy systems describes the mechanics of a testing framework, it does not perform an in-depth analysis of the test input generation methods. This dissertation performs this analysis by applying different test generation methods to autonomy systems using this framework and empirically evaluating them.

Justifying the selection of inputs for robustness testing given a testing budget has inherent trade-offs in terms of input space coverage versus failure-triggering parameter coverage. Choosing from too broad of a pool of inputs means that, in some cases, lots of tests will be wasted not finding any bugs. Drawing from too narrow of a pool carries the risk of missing a bug entirely. In both extremes, testers miss an ideal combination of tests that will uncover the most failures at the least cost. However, because the distribution of bugs that activate in a system is unknown and likely to differ from system to system, there is no universal optimal answer for this ideal combination. By comparing input stress testing methods, this work describes how various ways of

achieving input space coverage translate to fault coverage for a given system. Understanding the specific effects of input generation methods allows us to justify the use of a hybrid test strategy that draws on the strengths of several test methods.

1.1.3 Modes of comparison

Intuitively, a testing technique is better if it finds more bugs, faster. However, simply measuring the number of bugs a test method finds unfairly rewards test methods that consistently trigger the same underlying failure, or even the same symptoms of a failure, while a test method that can eventually trigger many unique failures is useless if it cannot do so within a given test budget. It is therefore valuable to distinguish between different symptoms of bugs, as a crude measure of underlying bugs. In this dissertation, we define two main metrics of comparison for test input generation methods that encompass these two goals of testing. Because determining whether two inputs truly trigger the same or different underlying bug is impossible in principle in black box testing, we include a discussion on how to approximate this measurement.

We illustrate how the metrics can be used to provide insight into the utility of test input generation methods. We also use these metrics to empirically justify the use of a hybrid testing technique that can provide more testing utility than a single test input generation method when applied to an autonomy system.

1.1.4 Assumptions

In this dissertation we deal with modifying an existing testing framework to expand the capabilities of the robustness testing method. We apply this framework to several existing autonomy systems under simulation. We assume that the test framework implementation is sufficiently correct (i.e. the framework successfully automates the act of setting up a robot simulation). In cases where we have encountered limitations of this framework, we have modified it to suit our needs. We also assume that the systems under test have valid documentation and accurate simulation as provided. We are using the software of the systems under test as-is and have not modified any of

the code.

1.2 Summary of terms

The following terms are defined in detail in the subsequent chapters, but, for reference, we provide a summary of them here:

A **use case scenario** is a behavior of a robotic system defined by the documentation of the system, that can be autonomously run. For example, a robot planning and following a path through a pre-determined set of goal points is a use case scenario.

A **nominal input** is an input provided to the system that, when played on the robot system, causes the robot to perform a given use case scenario. For the path planning example above, such an input would be the set of goal points and a start command.

A **test input generation technique** is a function that, given the data type of a parameter, outputs a test value. For fuzz testing, this would be a random value of that data type, while for dictionary-based testing, the value would be drawn from a pre-determined dictionary.

A **perturbed input**, or **test input**, is a nominal input that has been perturbed by some sort of test input generation technique, by replacing a fraction of values in the nominal input by values generated by the test input generation technique.

A **test case** is a single run of a robotic system with a perturbed input as the input.

A **test campaign** is a set of test cases for testing the robot. It may include testing several use case scenarios and using several different test input generation techniques to generate perturbed inputs.

An **invariant** is some property of the state of a robot that must always be true in order for a robot to be operating safely. A speed limit is an example of a simple invariant. An **invariant violation** results when this property is violated.

A test case **finds a bug** or **triggers a failure** if running the test case results in an invariant violation.

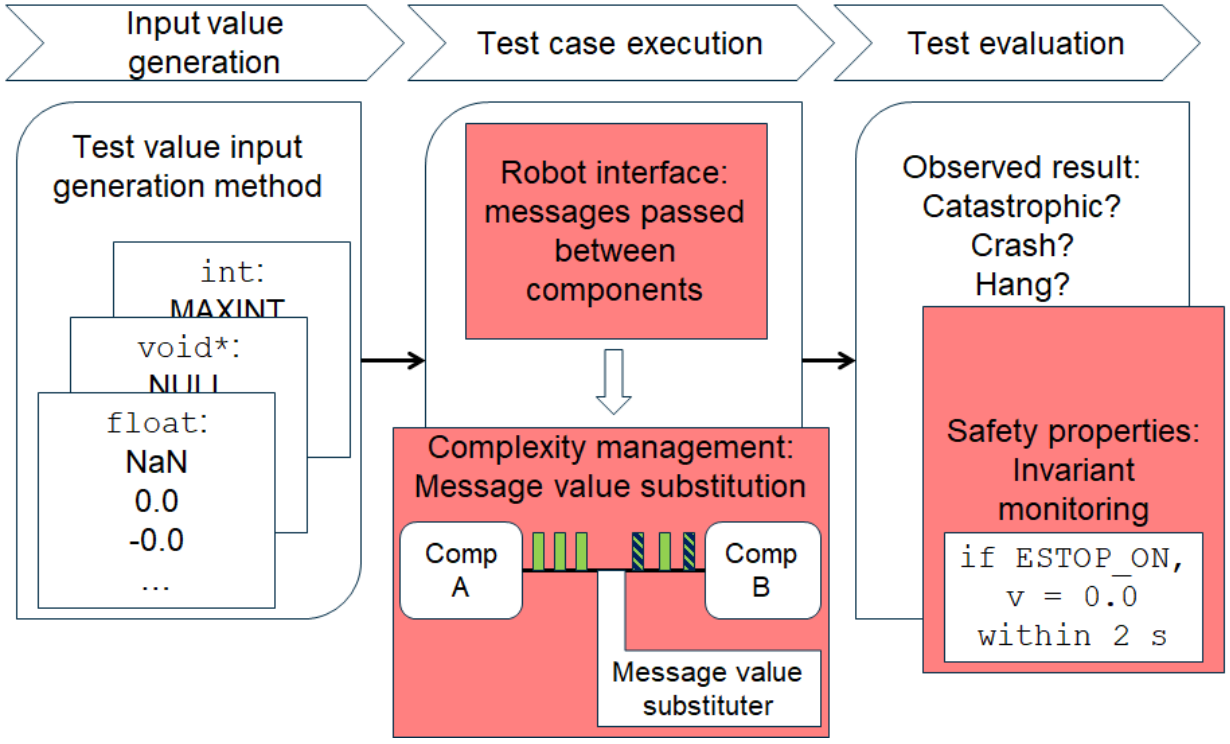

Figure 1.2: Changes to approach for autonomy systems

Test campaign **efficiency** is the reciprocal of the expected number of tests to first failure for a test campaign. Efficiency essentially represents the speed at which bugs are found.

Test campaign **effectiveness** is an approximation of the unique bugs found in a test campaign. In Chapter 4, we discuss several different ways to measure effectiveness, for example, the number of unique use case scenarios in which a bug is found.

1.3 Overview of approach

In this work, we explain how the properties of autonomy systems lead to the need for a modified robustness testing approach (see fig. 1.2). Contrasted with traditional robustness testing, this method uses messages passed between components as the interface to the system, uses message value substitution to preserve the complex state and timing requirements of an autonomy system, and detects invariant violations to check if the system has become unsafe. Using this testing framework, we apply existing test input generation techniques to autonomy systems, and

evaluate them using our defined metrics. Because these generation techniques have parameters of their own, and leave ample room for customization, even a single generation technique may have many variations. In this dissertation, we explore how changing these parameters affects the efficiency and effectiveness of a test campaign. Exploring different testing parameters expands the knowledge of what sorts of approaches may work for autonomy systems.

We show that the initial comparison between test input generation techniques highlights the tradeoffs between dictionary-based testing and fuzzing. This motivates devising a hybrid test strategy that results in an overall efficient and effective test campaign. We conclude with recommendations for developing and testing autonomy systems based on the lessons we learned while doing this work.

1.4 Contributions

This dissertation makes the following contributions:

1. An approach to metrics for comparing robustness testing techniques
2. Robustness testing results for three open source autonomy systems using dictionary-based testing, fuzzing, and certain variations.
3. A hybrid testing technique for each of the three open source autonomy systems, shown to outperform each of the basic testing methods.
4. A recommendation of heuristics for hybrid testing techniques and a list of lessons learned to inform testing autonomy systems in the future.

The rest of this dissertation is organized to present these contributions. Chapter 2 summarizes background and existing work. Chapter 3 presents our testing framework, and Chapter 4 introduces our metrics and experimental procedure. Chapter 5 performs an initial comparison of test input generation methods. We analyze various combinations of these methods when we empirically compare different hybrid test case selection strategies in Chapter 6. Additional input generation approaches are explored in Chapter 7. Finally, Chapter 8 gives some anecdotal

recommendations for writing and testing autonomy systems, gleaned from our experience in this work, and Chapter 9 concludes.

Chapter 2

Background

This chapter gives some background on software testing in general, with special focus on black-box robustness testing. We discuss existing methods of evaluating how well testing performs, especially in the context of coverage. We also explain how the field of autonomy system testing has opportunities for more research, such as the work presented in this dissertation.

2.1 Software testing

The field of software testing is vast. In this dissertation, we focus on black-box testing [21] using system input parameters. The majority of the works cited in subsequent sections of this chapter relate to this scope. However, for context, we begin with a broad overview of software testing.

Bertolino defines software testing as consisting of "observing a sample of executions, and giving a verdict over them" [25]. She outlines several key achievements that fit this definition and advance the field of software testing, including the introduction of systematic testing processes, component-based testing, and reliability testing. She names having a uniform theory of testing, i.e. one that "ties a statement of the goal for testing with the most effective technique, or combination of techniques, to adopt," as a "longstanding dream of software engineering research."

Further overviews of branches of the field of testing are given in Ammann and Offutt [15], Myers et al. [90], and Kaner et al. [63]. Namely, these books describe testing as a way to exercise assumptions about the execution of a program. They also discuss aspects and measurements of the testing process, such as coverage, the problem of test oracles, and the test execution process. Distinctions are made between automated and human-created test cases, testing at different levels of software abstraction, and testing for different steps of the software development process (functional testing, acceptance testing, regression testing, usability testing). In this dissertation, we are concerned with **black box**, **system-level robustness** testing with **inputs injected at an interface**.

We contrast software testing with static program checking tools, which have been effective at identifying defects, but do not involve dynamic execution of a program [55].

All of the sources in this section mention that testing can only show the presence of software faults, and not prove their absence. Metrics of coverage (see Section 2.5) can help approximate tester confidence, but this remains an inherent problem with testing, and one that this dissertation grapples with.

2.2 Black box testing

Nidhra and Dondeti give a survey of black box and white box testing techniques [91]. In summary, black box testing, or functional testing, "designs test cases based on the information from the specification," that is, with no view of the source code. Given a set of formal requirements of a system, black box testing can exercise the software interface to see if the requirements hold. Conversely, white box testing "designs test cases based on the information derived from the source code." In white box testing, knowledge of the flow of the program can inform test cases and their expected results. One advantage is that white box testing allows access to more comprehensive coverage techniques. Gray box testing is a middle road between the two, using the compiled binary of the source code as another basis for designing test cases [65]. For goals of our testing, we may be dealing with cases where the source code is unavailable due to contracted

confidentiality, or trying to find cases where the implementation does not match the specification of the system [45]. We therefore focus our dissertation on black box testing.

2.3 Robustness and input parameter stress testing

In this dissertation, we use the IEEE definition of robustness, which is "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [60]. Robustness testing sends invalid inputs to a software system or component and observes the result. We also use the term "input parameter stress testing" to describe the broader idea of testing using potentially unexpected or exceptional values on some interface. In this section, we explain how to generate such unexpected values.

2.3.1 Fuzz testing and variations

Pioneering work in input parameter stress testing is fuzz testing, or fuzzing, by Miller et al., which generates inputs randomly, according to a uniform sampling of the space of possible inputs [85]. For example, to fuzz a 32-bit integer parameter, each of the 2^{32} values is equally likely to be selected. Fuzzing has been applied in many domains, including operating systems [86, 87], mobile applications [77, 79], and general security applications [37, 50]. In practice, implementations of fuzz testing are not purely generational, due to the constraints of complex systems under test [26]. All three of popular fuzzer products PeachFuzz [11], open source security fuzzer American Fuzzy LOP [1], and Google's OSS Fuzz [10] support features such as seed selection, coverage criteria, special value dictionaries, and test case reduction. Several of these approaches apply white box testing techniques in combination with fuzzing. Some of these additional features turn these fuzzers into what this dissertation would consider hybridized approaches. To our knowledge, however, the past work has not systematically derived and empirically evaluated the hybridization.

Fuzz testing efficiency has been improved by constraining the input space, or by introducing statefulness [102]. Statefulness has been introduced using model-based testing, which at-

tempts to generate tests based on the functionality requirements of a model [38, 100]. Dalal et al. note that models of requirements make it easier to generate test cases that wouldn't otherwise be created, but that the selection of model is open-ended and has a large impact on the effectiveness of their tests, sometimes hindering automation [38]. Further work confirms these tradeoffs [40, 105]. The SAGE project replays program traces to discover input constraints that guide fuzzing [50].

2.3.2 Dictionary-based testing

A different approach to input generation is dictionary-based testing. Ballista was a project to robustness test software systems, using an approach where each parameter type had a pre-set dictionary of suspected exceptional values intended to trigger edge case behavior, such as `NaN`s for floating point parameters and powers of two for integers [72]. Main takeaways from the Ballista project are that failure-causing test cases generally have a low dimensionality (one or two parameters are sufficient for triggering a failure), well-formed values are necessary in the dictionary to get past any existing validity checks, and that even mature software specifically developed to be robust has robustness vulnerabilities [70].

2.3.3 Combination testing

Dictionary-based testing can be thought of as narrowing the large input space of a function by predicting that certain inputs can trigger boundary conditions in the code. Another way to approach the testing problem is by systematically creating combinations of input parameters. This is known as combination testing [52]. Work in this area involves either selecting inputs that are far away from each other by using approaches such as antirandom testing [80], or by generating disjoint subsets of combinations of input parameters, such as t-wise coverage. These approaches can be used to evaluate test input coverage, but generally do not scale well to the number of input parameters of our work.

2.3.4 Testing interfaces

Fuzzing, dictionary-based testing, and related methods test at the API level by calling functions with various inputs and evaluating their results. Testing may also be performed at other interface levels. Fault injection inserts software or hardware faults into a system under a workload and observes how the system performs [56]. Man-in-the-middle attacks, in which a malicious intermediate agent is placed on a channel between two nodes, are common ways to test networks [35]. Protocol fuzzing makes use of protocol specification to make well-structured inputs with test values for more complex interfaces [104].

2.3.5 Other approaches

Partition testing attempts narrow the input data space, by splitting the input space into disjoint subsets, within each of which the software is assumed to exhibit similar behavior [41, 107]. Partitions can be created in many different ways, whether by path coverage criteria, data flow semantics of the inputs, or according to responses to program mutants. Category partition testing outlines a process by which the requirements specification for a piece of software is used to create a set of equivalence classes for testing [93]. Related to partition testing is boundary value analysis, which focuses on the boundaries of the partitions [98].

Statistical software testing assigns probabilistic distributions to the input space [108]. RIDDLE is a tool to generate random inputs that conform to a specified input grammar that was used to robustness test Windows NT [48]. Metaheuristic techniques describe a large field of related test input generation techniques, some of which are informed by the source code or program binary [83].

Mutational fuzzing constrains the input space by taking a well-formed input and randomly mutates sections of it, from bytes to whole fields in a data structure [102, 111]. There has been previous work on selecting how to mutate the inputs [31], and can range from modifying single bits, to bytes within a word, to adding random noise to inputs. Mutational fuzzing and generational testing have been compared in the past by Miller et al., who note that mutational

fuzzing has the potential to create more meaningful test cases because it starts with well-formed inputs, but concludes that for an initial data set that does not have a diverse set of well-formed inputs, generational fuzzing fares better by introducing diversity into the input space [88].

2.4 Fault classification

Test case evaluation is another field of study. Ballista used the CRASH scale to evaluate tests based on catastrophic, restart, abort, silent, and hindering failures [72]. The first three are easily detectable using exception handling and give a notion of the robustness of a system. Other test oracles have been proposed, and are surveyed in Barr et al. [19]. These oracles fall in four broad categories: specified test oracles, derived test oracles, implicit test oracles, and handling the lack of test oracles. Specified oracles can be defined using formal languages, while derived test oracles involve usage of things such as documentation, n-version comparisons, or mutants. Of particular interest to this dissertation are invariants, which are discussed in more detail below (Section 2.10.1). Implicit test oracles can be the crashes and hangs detected by fuzz or dictionary-based testing, but also involve techniques such as anomaly detection and memory leak detection [32]. The goal of this dissertation is to provide an automated and widely adaptable testing tool, so we deal with oracles that detect software crashes, as well as some safety invariants.

2.5 Test Coverage

White box testing and black box testing use code and requirements coverage, respectively, to help assess the completeness of a testing method, with the argument that a testing method that exercises more of the implementation or requirements will find more bugs. However, these coverage metrics are used only to determine if the existing code or requirements are implemented correctly, and might not reveal missing code or requirements. Efforts to identify such gaps are referred to as negative, rather than positive, test case generation [92]. A coverage approach based on interface input data motivates a test method that exercises enough of the input space to check

for safety violations that stem from an incomplete implementation. However, the large size of the input space necessitates a more complex input selection coverage metric than "percentage of input space tested."

For context, white box testing, which considers the source code, typically uses branch, condition, or path coverage to assess the completeness of a test campaign. Branch coverage measures whether all branches of a particular piece of software have been exercised, and path coverage measures whether all paths, or sequences of branches, have been exercised. Another approach to white box coverage is data flow coverage, which measures whether all declared uses of variables, such as initializations, allocations, and deallocations, have been exercised [57]. Because full data coverage is intractable, this work constraints the data coverage problem to "feasible data flow," or testing that exercises only reachable data definitions [44]. A subset of this work compares and contrasts branch and data coverage and finds that, for most of their sample programs, data coverage testing was able to uncover more errors [43]. The underlying idea behind all of these metrics is that if all feasible logical combinations of a program are exercised, the tests are less likely to miss bugs stemming from faulty implementation. However, branch coverage cannot account for all possible execution paths, such as arbitrary iterations of a loop [24] or branches that are missing from the code [81].

Black box testing does not require access to source code, and thus does not rely on knowledge of branch or data path logic to measure completeness. Combination testing informs several data set coverage metrics [52] (see Section 2.3.3). One approach for black box testing is to measure requirement coverage, which means writing test cases for all documented functional requirements. There are also gray box coverage metrics, which involve exercising branches in the compiled binary of a piece of software [28].

More complex coverage criteria, such as MC/DC (modified condition/decision criteria) can also be used to create partitions of testing data [113]. Chockler et al. discuss how formal verification of software, in particular, benefits from the use of several different coverage criteria [34].

For our purposes, it is important not to not judge the value of a test campaign solely on these criteria. They can be considered secondary criteria to judge if some testing goals are being met, but the end goal of testing is to find defects as efficiently and effectively as possible.

2.6 Testing comparison and evaluation techniques

Johansson et al. compare fuzz testing, Ballista-style testing, and input parameter bit flip fault injections for operating systems [61]. The four comparison criteria they define are error propagation, error impact, implementation complexity, and experiment execution time. Using these metrics, they find several tradeoffs between the error models in terms of severity and number of errors found with respect to execution time, and ultimately propose a hybrid model for their system. Winter et al. follow up on this work by refining the metrics to use relative coverage, experiment efficiency, execution time, and implementation complexity [110]. They compare bit flips, Ballista-style testing, fuzzing, and single-event upsets on the Windows CE kernel and describe several tradeoffs between the testing methods based on cost of testing. Their approach creates a good starting framework for comparing testing techniques and uses both efficiency and service coverage (similar to what we would call "service effectiveness," as in Section 4.1.2) to analyze the performance of the test methods. However, as we show in Chapters 5 and 6, a more granular notion of effectiveness, such as type effectiveness, would provide deeper insight into the tradeoffs between the test methods and allow for a possible recommendation of a hybrid metric. Hamlet does initial work comparing partition testing to random testing, and concludes that partition testing is not much better [53], while Loo et al. [75] argue that the performance of random testing depends on the software being tested. This indicates that more comparison, especially with dictionary-style testing in place of partition testing, is necessary.

Mayer et al. note that directed automated testing generates efficiency metrics when dynamically choosing test cases, and use these metrics to compare the efficiency of directed testing techniques [82]. Other papers compare testing techniques other than input parameter stress testing [20].

Robustness interface testing attempts to achieve input coverage by selecting from an input space that is suspected to trigger failures. Past work that presents techniques to automatically test systems for robustness tend to circumvent the impossibility of precisely measuring bug coverage by presenting the challenges of testing the system, and showing some novel bugs that were discovered by their result [16]. Sometimes, they favorably compare their results to some other testing method, typically fuzz testing [49, 97]. However, these papers look at the number of failures triggered per system or component, and do not necessarily discuss the relative uniqueness or depth of the bugs found by their method versus fuzz testing. It might even be the case that a system would benefit from being tested using two different test input selection techniques in combination to find different types of bugs, but if a paper is only concerned with presenting one testing method, this potential improvement is missed.

2.6.1 Software reliability

Software reliability prediction models use existing testing data to predict the temporal distribution of future errors when continuing the testing effort. These models allow project managers to decide whether the current testing method will yield significant results over time, which is an important consideration given finite testing budgets. While these models can give an indication of how well a certain testing method is performing, the smoothness of the models may mask interesting features of raw testing data or force asymptotes that do not actually exist. There have been papers that compare the accuracy of popular software reliability models [66] and which match attributes of testing data to the most relevant model [51, 112].

2.7 Test input reduction

Once a failure-causing test case is found, it is helpful to identify the specific parts of the test input that is relevant to triggering the failure. Statistical debugging compares correct and incorrect runs of a program to localize faults [74]. Delta debugging is an automated technique for iteratively trying smaller and smaller portions of the original test input until an irreducible

part is found [115]. Hierarchical product set learning identifies the fields within an input that are relevant to the bug [106].

2.8 Cause versus symptom of bugs

At its core, black-box software testing only finds bug symptoms, rather than underlying causes. Test input reduction may help more specifically describe these symptoms, but further analysis is needed to determine the root cause of the bug. Reversible debugging is a technique that allows for program execution to step forward and backward to "step out" from the source of a bug to the cause [42]. An approach by Zeller, et al. uses delta debugging to isolate code changes that introduced a bug [114]. These approaches use white-box techniques to analyze code and point to underlying causes of bugs. In black box testing, bug symptoms can be classified using their inputs (as above), their severity [84], and information gained from human-written bug reports [116]. While this classification does not definitely determine a root cause, distinct classes of bug symptoms can be indicative of distinct bug causes.

"Fuzz taming" can estimate the number of unique bugs by defining a distance function between test cases [33]. This approach requires distinguishing metrics between bugs, and, in fact, one approach is to use the distance between minimal delta-debugged test inputs as that distinction [95].

2.9 Hybridized testing techniques

Work in bolstering input parameter stress testing usually involves adding some sort of state-based mechanism to the testing process. For example, Cotroneo et al. create a state model of an operating system execution and apply dictionary-based testing to every state they traverse [36]. Pak et al. use symbolic execution to explore program paths before fuzzing nodes on this path [94]. However, this approach requires engineering a more complex test harness, while our work improves input parameter stress testing results by intelligently selecting the input space.

2.10 Autonomy Systems

Autonomy systems, or robotics systems, are systems that interact with their environments and perform some or all functions without human control [22]. This may mean that, given a map and a goal, they plan a path, or, given camera input and an articulating arm, they compute a way to move the arm to avoid hitting any obstacles. Autonomy systems are becoming increasingly popular and have applications in medical, military, manufacturing, and transportation domains.

2.10.1 Autonomy system safety

Testing autonomy systems is important precisely because these systems interact with the physical world and therefore have special safety considerations, even when they are not officially considered "safety-critical." There are many challenges in the field of autonomy safety [69], including novel sets of inputs and high safety requirements.

One way to evaluate autonomy system safety is by the use of formal specification and safety invariants. Invariants are properties of a system, defined by a formal logic, that must always be true for the system to be safe [27, 76]. Run-time monitoring of system safety has been extended to the use of invariants [58, 62].

2.10.2 Testing autonomy systems

Autonomy system software differs from traditional desktop software because it is distributed, very stateful, temporal, and cyber-physical. This means that these systems interact very closely with their environment and present challenges to test beyond traditional software system. Recent autonomy systems amplify the testing challenge because they internally tend to pass hundreds of messages per minute, and each message can contain complex data types such as arrays of doubles representing laser scan data.

Work in testing autonomy systems is limited in the literature, possibly because many of these systems are developed under proprietary licenses rather than as academic work. Open source software presents an opportunity, and Timperley et al. analyze bug reports in the ArduPilot

software to gain knowledge of autonomy bugs in simulation [103]. General work about the development of autonomy systems cites the typical software engineering methods, such as unit testing; as well as methods more uniquely suited to cyber-physical systems, such as runtime monitoring [39]. Exhaustive testing of any system interface is intractable, and even more so for the message passing interface of a distributed autonomy system due to large sensor datasets and system complexity [54].

The framework in our work builds upon the Automated Stress Testing for Autonomy Architectures (ASTAA) project [59]. This work introduced a framework for robustness testing autonomy systems, based on key differences between autonomy systems and traditional software systems. Chapter 3 describes this framework in detail, and how our implementation differs.

2.11 Summary

We have presented an overview of previous work in software testing approaches, including test input generation, test case evaluation, and the metrics of how well various test methods perform. Our work builds on this body of research to develop and evaluate a black-box robustness testing tool for autonomy systems. We use dictionary-based testing and fuzzing as our input generation techniques. We also use invariant checking and test input reduction to evaluate and classify our tests. Because existing metrics of comparison of test methods, and especially hybridized methods, do not exactly encompass the goals of this dissertation, we introduced new metrics that inform a more systemic way of evaluating hybrid test methods.

Chapter 3

Framework for Testing

In this chapter, we present a framework for testing autonomy systems. We begin with the differences between autonomy systems and desktop software systems that necessitate a different approach to testing than for existing software robustness testing approaches. We discuss progress that has been made in this field, including previous work at the National Robotics Engineering Center (NREC), and then explain how we instrumented our tool for gathering the results in this dissertation.

3.1 Challenges and motivation

As discussed in Chapter 2, desktop software robustness testing has been studied and the literature forms a solid basis for thinking about robustness testing frameworks and input methods. For traditional desktop software, a robustness testing tool makes a function call to a defined API (such as POSIX) and detects whether the result was a catastrophic system failure, a process crash, or a process hang. However, these techniques do not directly transfer to autonomy systems. Testing autonomy systems is challenging because they differ from traditional software systems in several ways. In the Automated Stress Testing for Autonomy Architectures (ASTAA) project, we identified four main differences between autonomy systems and traditional desktop software systems that create special considerations for testing [59]:

- **Distributed:** autonomy systems are made up of components (e.g. sensors, actuators, CPUs) that communicate with each other over a bus to share data and complete their tasks. While traditional software systems may or may not be distributed, autonomy systems as a whole are different because they are typically distributed by nature to accommodate all of their components. Ideally, testing a distributed system would encapsulate the communication between nodes rather than relying on a single external API.
- **Temporal:** traditional software systems are typically transactional and take in an input to produce an output. In contrast, a robotic algorithm is essentially a list of tasks to perform, and the robot will keep doing these tasks until some termination or change of protocol criterion is reached. A framework that tests temporal systems would need to be able to verify robustness at all steps of the checklist, rather than simply checking the outcome.
- **Stateful:** behavior for a given autonomy system depends on what mode the robot is in, which is in determined by the input the system receives. A robot's mode transition rules can often be described using a state chart, and the robot will behave according to different algorithms according to the mode it is in. For example, if a robot detects an obstacle, it might go into an obstacle avoidance mode rather than continue its previous behavior. In contrast, many traditional systems have state that is only incidental, such as copies of variables stored in memory. To test such behavior, the testing tool must be able to ensure that the robot is in a non-trivial, testable mode, and ideally test the robot in a variety of modes.
- **Cyber-physical:** unlike traditional software systems, which are generally devoid of feedback from the real world, robots are made to interact with the environment around them. This means that they must obey timing constraints and control loops, deal with a wide variety of input, and obey safety requirements. A tool that tests a cyber-physical system must be cognizant of these constraints, which presents a difficulty in automating tests.

These key differences motivated three concrete questions that allowed ASTAA to define the

difference between testing autonomy systems and traditional software:

What is the interface to an autonomy system? Traditional software systems have an API, where a robustness test case can be a function call to that API with certain parameters. In contrast, autonomy systems do not expose an API in the traditional sense. The closest analogue is the command and control interface to a robot. However, if a robot is fully autonomous, the external interface might be as limited to the commands "start," "stop," "reset," and "emergency stop." A more accurate model of the inputs to a robot is the sensor information that the robot receives from the world, but simulating the real world is complicated and expensive [68, 78, 101]. Furthermore, because a robot is a distributed system, simply passing world data to a robot will not test the messages passed between components to a robot. Therefore, it makes sense to use the message-passing mechanism itself as the testing interface to a robot. By injecting spurious values on this bus, we can emulate several fault models. Namely, this method encapsulates node failure, brownout, or malfunction, as well as channel failure and even faulty or unexpected sensor data. Any of these issues are relevant to a robot, because if a node malfunctions for any reason, the robot must still be able to maintain its safety properties.

How do we maintain the complexity of an autonomy system while automating testing? A robot is distributed and stateful, and has timing requirements and message-passing assumptions. Rather than just passing a single message as a test, an automated testing framework needs to accommodate this complexity in order to get the robot into non-trivial states before injecting values. One way to do this is do this by taking live runs of the system and injecting on them using a man in the middle [29] model. Interception testing takes messages sent from `Node A` to `Node B` in a live run, and mutates them, by either injecting new messages, modifying the entire message, or changing just some of the fields of the message [59]. This is done according to timing rules, such that the robot has the chance to enter a certain state before injecting values. Unfortunately, automated interception is costly. For example, for a Robot Operating System (ROS) system, it is necessary to emulate a star topology of all the nodes in order to intercept.

Therefore, some shortcuts can be taken. One approach is log replay: for nodes that do not have control loops and do not need real-time feedback, we can take a log of some input messages to a node or system, modify messages in the log, and replay the messages on the system, checking for erroneous or safety property violating behavior.

How do we enforce the safety properties of an autonomy system? In traditional software, it can be sufficient to check for system and process crashes and hangs. However, for a robot, it is not enough that the system or node does not simply crash or hang. The system must also obey its safety requirements, such as speed limits, at all times. To enforce this in automated testing, we can use invariant monitoring. An **invariant** is a mathematical expression of some property that must always be true during the run of a system. A **safety invariant** is an invariant that must always be true in order for the robot to be safe. The outputs of the autonomy system can be checked for invariant violations by implementing an automated checker. Invariants are defined according to the safety requirements of a system: for example, if a robot claims a speed limit of 2 m/s, we write an invariant to check that all velocity outputs do not exceed 2 m/s. Invariants can also be stateful or more complex. For example, a common safety invariant is that a robot must stop within a certain time once the emergency brake has been engaged. A stateful invariant monitor can check for the emergency brake engagement input, and then check that all velocity outputs after the indicated time period are 0. Thus, the complex safety properties of autonomy systems can be formally defined and monitored for within the testing framework. Any invariant violation is logged and recorded as a safety violation.

The ASTAA architecture is presented in fig. 3.1. It implements the answers to the three questions above to address the distributed, temporal, stateful, and cyber-physical properties of an autonomy system. The ASTAA project had much success finding faults in autonomy systems and provides a solid conceptual basis for our robustness testing work [59].

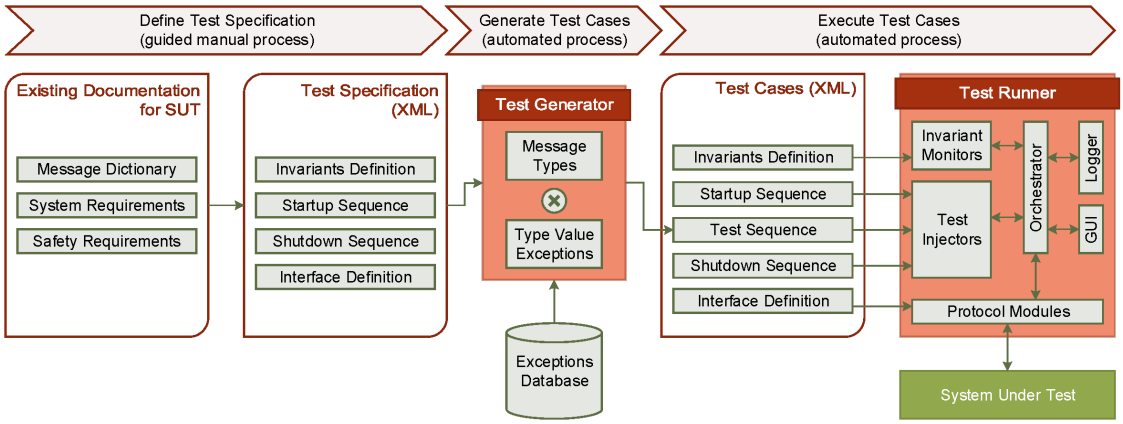

Figure 3.1: A simplified architecture of the ASTAA system [59]

3.2 Our Testing Architecture

While similar in fundamentals, the testing architecture for this dissertation builds on the original ASTAA architecture in several ways. Our method simplifies the traditional ASTAA approach by forgoing interception testing in favor of log replay with value substitution and by monitoring for invariant violations offline. This significantly cuts down on the overhead of the testing, as, for example, we do not need to emulate the communication bus for the system or infer the states of the robot to monitor for invariant violations. We also forgo the use of protocol buffers for instrumenting generic robotics systems in favor of working entirely with ROS for speed and reduced implementation complexity, but our tool can be modified in the future to support other systems. While this somewhat limits the systems we tested in this dissertation to ROS-based ones, the prevalence of ROS-based systems and the ability for our tool to make data assumptions when substituting on logs and analyzing results justified this choice. Finally, we perform invariant monitoring offline, by analyzing system logs after the test has run in simulation. This still detects safety violations, but does not add overhead to the tool itself.

3.2.1 Black-box testing

Our tool uses black-box testing. This means that the tool does not access the written code for a given system, and instead tests at the interface level. Even though the systems we test in this dissertation are open source, many of the systems we have encountered in the past have not been. Having a black-box testing tool hence allows flexibility, such as when testing is outsourced to a different team along with just the compiled system software. Furthermore, because the goal of our tool is robustness testing, exposing some interface is sufficient in providing a system with "unexpected inputs." White box techniques are useful for ensuring that each branch of code is exercised in testing, but do not necessarily account for missing branches made due to data assumptions. For example, if the code branches on a comparison that breaks with `NaN`, but only well-formed test values are provided to make the comparison true or false to test the branching, white box techniques are not sufficient to find this failure. Finally, because autonomy systems are distributed, stateful, and temporal, fully exercising every branch of code may require a complex coordination of all of the system nodes. The complexity of this problem is beyond the scope of this dissertation.

3.2.2 Testing tool procedure

Figure 3.2 illustrates an overview of our testing procedure. For a given system under test and use case, our tool takes the following inputs:

- A simulated representation of the system under test. This is an environment loaded with all the necessary software to emulate an instance of the autonomy system.
- A script to start the system up and run a use case scenario. This script initializes the system to log any core dumps, launches the system into a defined initial state, and plays the test input on the system. It is also responsible for shutting down the system and gathering any log files into a pre-specified directory. We wrote all of the scripts used in this dissertation based on the use case scenario tutorials provided in the documentation of each system.

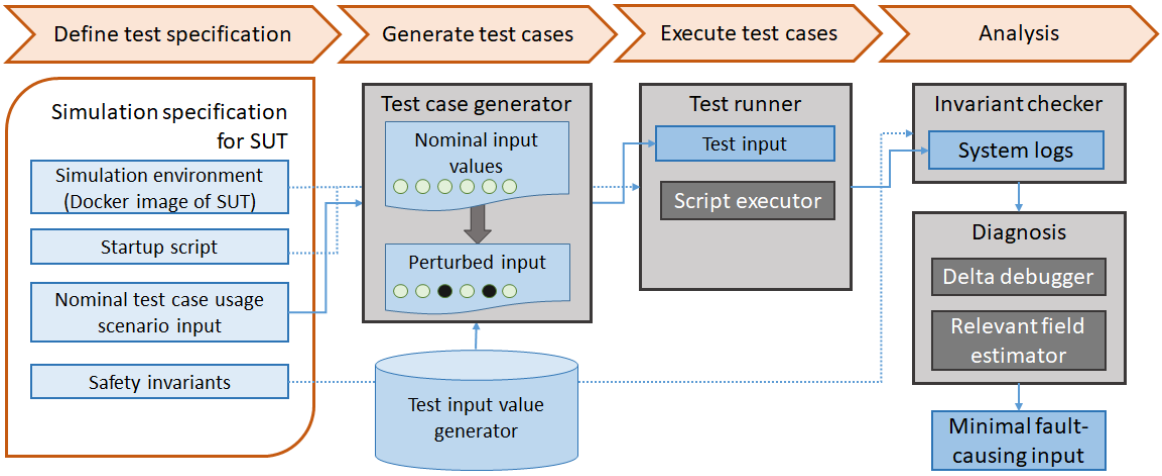

Figure 3.2: Our testing procedure

- A log of inputs from a nominal run of the use case scenario under test. This is a pre-recorded log of system messages that are inputs to various components or nodes of the system. The test script is written to correspond to the behavior of these inputs. For example, if the inputs contain arm articulation messages, the script should start up the arm node(s) of the system. When re-played on the system, this nominal input does not cause an invariant violation. This nominal input will be perturbed to perform a test case.
- Any additional configurations or constraints to testing. By default, there are no constraints and the testing tool may perturb all fields of the nominal log. The user may choose to exclude messages or fields from testing, or to specify a nominal input replacement percentage or other rule.
- A means of selecting or generating test case inputs. For the purposes of this chapter, this is either an exceptional value input dictionary, or a fuzzer that generates random values. Different test case input techniques are explored in Chapter 7.
- One or more invariant checkers. An invariant checker is able to parse the system logs to check for any violations to the rule it implements. A simple invariant checker that can be used for most systems is one that checks for the presence of core files to detect a crash.

Given these inputs, the test procedure is illustrated in fig. 3.2. We describe it in detail here:

1. Replace values in the nominal input set with spurious inputs according to any constraints. By default, we replace 20% of values, as it allows for a fairly large rate of message substitution while still leaving most of the data assumptions of the system. Historically, ASTAA had success finding bugs using this percentage. We leave time and duration fields untouched by default. All fields of an array are replaced, up to a maximum. This creates a test input.
2. Copy the test input and use case scenario test script to the simulator environment. We employ the Docker [3] framework to do this, which ensures a sandboxed emulated computing environment.
3. Run the simulation script provided on the system, with the test input. The core dumps and system logs that the script has enabled are stored in a directory on the computing environment.
4. When the script concludes playing the input log, copy any system logs to a working machine for analysis. Invariant checkers are run at this point, and any bugs that are found can be diagnosed.

Because nominal input values are pseudorandomly replaced according to the replacement rules, by repeating the process with different random selections, we can generate a campaign of many test cases for a use case scenario. In the rest of this dissertation, when we write that we "ran N test cases," N refers to the number of different random test cases generated according to this method.

Because our testing framework takes a nominal run of the system and modifies the values by replacing them with test values, we have to define some rules for replacement. It is possible to replace all fields and messages with the test values, but this might lead to a robot that never enters a meaningful state because it shuts down or does safety mitigation in the presence of overwhelming exceptional input. Therefore, we replace only some of the messages of the system.

By default, we randomly replace 20% of the fields with test values. Chapter 7 explores different replacement paradigms to see if we can find more or deeper bugs using different replacement percentages. Our previous work in autonomy testing also allowed us to replace messages according to more complex rules, for example only after a robot has run for a specific amount of time and has entered a certain non-trivial state, to test the robot's behavior in that state. However, this introduced an additional level of human input in testing, which hinders automation. In the implementation described in this dissertation, the system startup scripts, which are written once per use case scenario, ensure this initialization for the systems under test. There are no hard and fast rules for how input should be replaced, and the technique we use comes from past experience at NREC.

3.3 Input Generation

We describe our approaches to input generation methods here. A summary of these methods for various input types appears below, in table 3.1.

3.3.1 Dictionary Testing

Exceptional value testing involves selecting input values from a preset dictionary of suspected exceptional values for each parameter type. These values are selected according to developer experience. Suspected exceptional values include values such as `NaN`, $\pm$`INF`, ± 0.0, and denormalized values for floating point types; and maximum and minimum values as well as powers of two for integer types. The idea of the exceptional value dictionaries is to describe values that might trigger edge case conditions that developers have not accounted for. This approach is a response to the naivety of fuzz testing: searching an input space by randomly selected inputs is probabilistically unlikely to result in selecting edge case values that might trigger these kinds of errors. However, since these dictionaries are human-made, albeit based on years of tester and developer experience, the risk is that they left something out. Because we do not have an oracle of how many bugs actually exist in a piece of software, we do not know if the exceptional

value approach will simply never trigger certain errors because the error-triggering input is not in the exceptional value dictionary. This is in contrast with fuzz testing, which can theoretically hit all error-triggering inputs if given enough time (this is intractable in practice, especially if the number of messages one can send is limited by the timing restraints of the bus). In practice, a dictionary-based approach has shown results in crashing many types of systems, including autonomy systems.

3.3.2 Fuzz Testing

Fuzz testing inputs are provided to the testing architecture at the same step in the implementation as dictionary-based test cases, except that they are randomly generated rather than selected from a dictionary. The fuzz inputter can return inputs for every primitive data type supported by ROS, including signed and unsigned integers of various lengths, booleans, ASCII, floating point and double precision numbers, and variations thereof. The integer values are chosen uniformly from a range defined by the size of the field being generated. For example, `int16_t` would be from the range -32768 to 32767, whereas `uint8_t` values would be from the range 0 to 255. Derivative data types that can be statically cast to integer types, such as char and byte, are generated the same way. Floating point numbers are selected uniformly from the space of binary representations of the number. For example, a 32-bit float value would be generated by randomly generating a 3- bit value. Unit tests on these generators were run to verify that the generation was done correctly. For floats in particular, this unit testing checked that the ratio of `NaN`s selected from the pool was similar to the actual number of bitwise representations of `NaN` and $\pm$`INF`, that every power of two was equally represented, and that denormalized values are converted correctly. String generation is done character-by-character, where each character was chosen from the ASCII space 0-255. The string would terminate when a null character (ASCII 0) was generated. This is the way the original fuzzing work generated strings [85].

3.3.3 Other input methods

We also implemented several other input selection methods, which we primarily use in Chapter 7:

Nominal input mutation: Instead of replacing nominal values with values generated from a dictionary or from fuzzing, we attempt to mutate existing values. For numerical values selected by the test mutation percentage (mutating strings is a special case and is discussed below), we randomly choose to perform a mutation to the value, such as adding, subtracting, and multiplying small numbers. For fields that are determined to be mask-type fields, we may also flip a bit in the value.

String mutation: We created a special mutator to deal with string-type fields. We observed that the nominal input logs had strings with values of forms similar to "`left_joint_3`." In our mutator, we truncate strings, we replace digits with larger or smaller values, and we swap instances of the substrings "left" and "right."

Dictionary entry mutation: Instead of mutating nominal input values to change the test input, we mutate the values in the dictionary to create a bigger dictionary. We do this to determine if the dictionary can be expanded in simple ways – for example, if a dictionary depends on boundary values, subtracting or adding 1 to a suspected boundary value such as a power of two may result in another interesting value.

Semantically-specialized dictionaries: Previous work has found that a dictionary-based approach is improved when the dictionary includes values specially constructed with the semantic meaning of the fields in mind [70]. For example, POSIX functions such as `mlockall` and `open` take an integer argument that represents some flags that are set, based on the bits of the input value. Rather than simply using the generic `int` dictionary to test these fields, the authors defined specialized dictionaries that correspond to interesting flags being set. In our work, by examining the fields in the nominal input logs, we explore whether adding such semantically-specialized dictionaries provides any benefit for testing autonomy systems. To determine seman-

Table 3.1: Summary of test input generation methods

Data type	Dictionary Examples	Fuzz generation rule
Integer (`uint`, `int`, `char`, etc.)	-1, 0, `MAXINT`	Uniform distribution on range
Float (`double`, `float`, etc.)	`NaN`, ±`INF`, Denormalized values	Uniform distribution w.r.t bit-wise representation
String	Empty string, 'a', 'left_joint'	Each character randomly uniformly generated until '\0'

tically special fields, we examined the nominal input log for the scenarios we tested. These are discussed in depth in Section 7.6.

3.3.3.1 Test input summary and open questions

To summarize dictionary and fuzz testing for the data types used on our systems, we provide table 3.1.

In our description of dictionary testing and fuzzing, we made several assumptions about how inputs should be treated. We also did not deeply consider how the nature of autonomy system input may affect test input selection. In particular, autonomy systems take in and process large amounts of data from the physical world, and make ample use of floating point arithmetic and large inputs such as point clouds. The scope of this chapter introduces basic test input selection methods, but questions about tailoring test inputs to autonomy systems are explored in Chapter 7.

3.4 ROS-based systems

To explain the specifics of our framework, we give a broad overview of the ROS architecture, which is the platform on which all of the systems we tested are built.

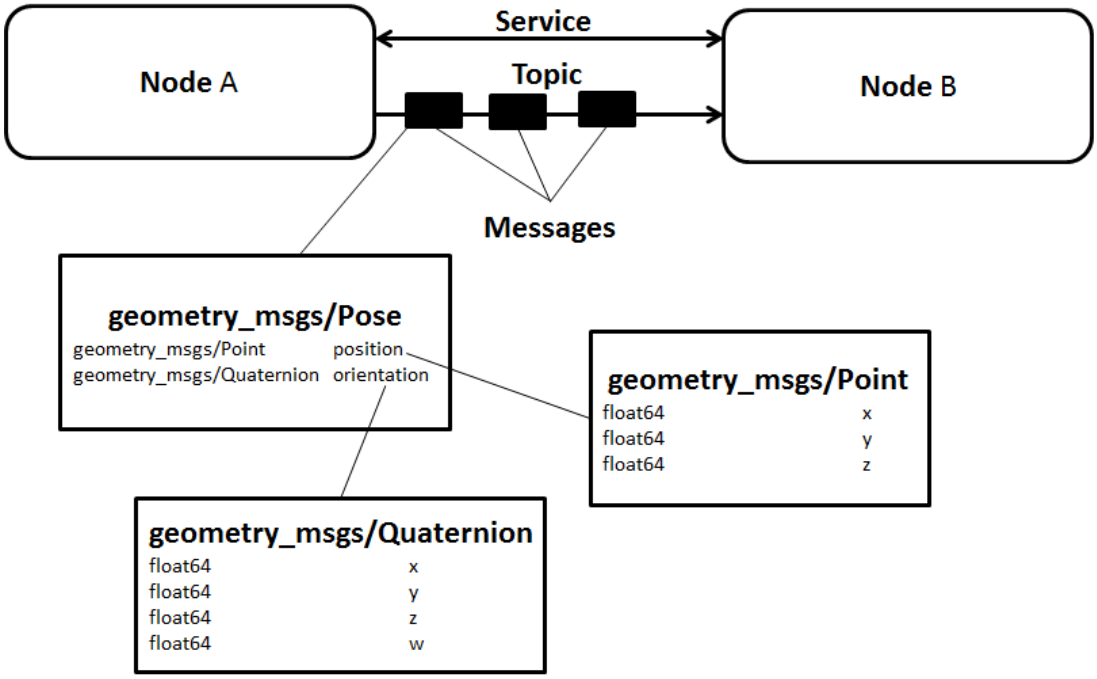

Figure 3.3: ROS communication example with Pose message

3.4.1 Overview or ROS architecture

ROS is a widely-used system that provides a distributed message-passing framework to implement the software for the components of an autonomy system. An overview of the ROS communication concepts is given in fig. 3.3. At a high level, a ROS system is comprised of ROS **nodes**, which are standalone components that can communicate with one another. Example ROS nodes within a system might be a laser scanner node, a move base node, and several nodes for articulating a robotic arm. The ROS nodes communicate using ROS **messages** and ROS **services**. ROS services are calls to a specific ROS node that return a response. A ROS message is a data structure that is made up of one or more fields. ROS provides some default ROS messages common in robotics applications, or developers can create their own. The ROS message structure of the Pose message, intended to define a position in 3-space, is given in fig. 3.3. ROS messages use channels called ROS **topics**, and a given ROS node can **subscribe** or **publish** to a given topic. For example, a laser scanner node can publish scan data messages to the topic /scan,

which is subscribed to by the move base. In fig. 3.3, Node B is subscribing to a topic that Node A is publishing to. ROS messages expose an interface to a robot built on ROS. By manipulating these messages in the ways described in Section 3.3, we can test these interfaces.

Logs of a run of a ROS system can be recorded in a ROS **bag**. A ROS bag is simply an ordered collection of ROS messages. It can be replayed on a ROS system, meaning that all of the messages in the bag will be published as-is on the same topics that they were originally published on, with roughly the same timing. By filtering only the input messages from a bag, and booting up a robotics system, we can simulate the real inputs to a system at any time. Because we use ROS in testing, we use the term **nominal bag** to define a nominal input log in the form of a ROS bag. The ROS API allows us to take a bag and modify the fields of the messages within a bag, which means we can change fields to fuzzed or exceptional values before replaying the bag. Because of timing issues, this playback is non-deterministic. We account for this by re-playing any invariant-violating bag to make sure we can reproduce the violation.

3.4.2 Simulation and Gazebo

We test our robots in simulation, because we do not have physical access to the robots we tested and because setting up a simulation and running many tests in parallel is much more feasible than with physical robots. While simulation has some limitations, such as possible timing differences or physics engine mis-calibration, past work in ASTAA has shown that bugs found in simulation can be reproduced on physical robots [30]. Furthermore, because the software run in simulation is the actual software on the robots, the bugs we found represent real failures to correctly handle all input and are symptomatic of problems with the software.

ROS has several mechanisms by which to simulate robots. In particular, ROS interfaces with the Gazebo simulation platform. Gazebo is a popular robot simulator that has extensive integration with ROS [6, 12]. Gazebo can simulate world inputs, such as an environment that contains walls, paths, and obstacles. If a ROS node takes in laser scan input, for example, Gazebo can emulate this laser data and pass it to that node.

3.5 Systems under test

All of the systems we tested run on ROS Indigo[1] and have tutorials for simulating particular use case scenarios under Gazebo. We give an overview of these systems, as well as the scenarios we tested, here.

3.5.1 Fetch and Freight

Fetch is a company that develops robots to assist people in a warehouse setting [5]. In particular, the Fetch robot is made up of a base that can move around autonomously, a perception system, and an arm that can grasp objects. Fetch has path planning code, and can build a map of its environment and then navigate to a goal according to the map. Fetch is built on ROS and exposes several interfaces using common ROS semantics. Fetch has nodes such as `move_group` and `basic_grasping_perception`. To test Fetch, we used the tutorials available on the Fetch simulation tutorial. The functionality we tested was:

- pick_place: in this scenario, the robot goes through a basic grasping and placing sequence. It picks up an object in simulation and uses planning and obstacle avoidance to move itself and the arm to put the object down. The input messages to this scenario are various goal points for the grasping, move base, and head controllers, as well as collision objects (entities that describe bounding boxes that the robot should not intersect with) and goals for picking up and placing objects. Unfortunately, two nominal runs of this system (out of 100) resulted in invariant violations with no test inputs provided. While we include this scenario, with caveats, in our exploratory analysis in Chapter 5, we exclude it from the more advanced analysis in Chapter 6.
- wave and disco: The robot goes through some dance motions, based on preset points that the arm must articulate through. Both scenarios have goal point inputs and collision object inputs. wave goes through 160 goalpoints while disco goes through 7. Additionally,

[1] `http://wiki.ros.org/indigo`

disco has a single "cancel" input to stop the motion.

We also checked the teleoperation functionality of the robot, but did not detect any faults.

Freight is an armless robot built on the same platform as Fetch, and any simulation can be run with the Freight parameter to test the Freight robot. In practice, we have found that testing using the Freight parameter does not yield any additional results, because Freight only makes use of nodes that are also used in Fetch, and therefore the software tested is Fetch is a superset of the software in Freight.

3.5.2 Ardupilot

Ardupilot is a popular autopilot software for autonomy applications such as drones, rovers, and hobby planes [2]. While the software itself is not developed specifically for the ROS platform, it exposes an interface that can be controlled using ROS messages and a ROS layer for facilitating this communication [17]. The specific protocol used is MAVLink [8], which was developed for communication with small unarmed vehicles. The open-source MAVROS package serves as an intermediate layer to interface ROS systems with MAVLink [99]. Because this particular system uses ROS servers to set parameters, and because ROS servers are not interceptible in the way that ROS messages are, we also wrote a communication layer to directly translate ROS messages into ROS server calls. Ardupilot is able to process many commands for controlling a vehicle, including takeoff/land, modes to circle in place, waypoint and mission planning capabilities, mapping, and multi-vehicle control. For our testing, we focused on an Arducopter drone in simulation. The functionality we tested was the following:

- cmd_vel: The drone is commanded to move at a certain velocity. The only input here is the velocity message.
- mission: The drone is put in the "mission" mode and flies through several waypoints. The inputs to this test case are the messages that specify several waypoints, defaulted from a configuration file provided in the software package.

- fence_then_mission and fence_then_vel: The drone goes through the scenarios above, but with a virtual fence that sets the boundary in which it flies. The inputs are the same as for the corresponding cmd_vel and mission scenarios, with added messages to set the fence boundary.
- modes: The drone cycles through several modes, including takeoff, land, and arming. The inputs are the commands to set the modes.
- setpoint_pos and setpoint_raw: the drone's pose is commanded using either using raw MAVLink inputs or a pose message that abstracts these raw inputs to a human-readable format.
- pos_then_accel: The pose of the drone is set, and then the node is commanded to accelerate. The inputs are the pose messages (same as in setpoint_pos) and the acceleration message.

3.5.3 Turtlebot

Turtlebot is a small, configurable robot platform designed for robotics hobbyists [13]. It has a small footprint with wheels and is able to drive around and spin, and process scan or camera data. The simulation allows for some configuration of the robot platform (for example, whether it is in the "burger" or "waffle" form factor, and whether sensors are present on the robot). The functionality we tested was the following:

- follow_route: The robot is given a route to follow and must plan a path and navigate the route. The inputs to this scenario are the goal points of the route.
- move_base: The robot is given goals for its move base and moves accordingly.
- map_scan: The robot scans its surroundings and builds a map of its environment. The input to this scenario is the raw laser scan data.
- teleop_vel: The robot is commanded via a tele-operation node, which translates keyboard input to velocity commands. We record these velocity commands and use them as an input

to the robot.

- nav_goal and nav_scan: The robot is given a goal for moving and, in the nav_scan scenario, a laser scan of its surroundings. It must navigate to the goal.

3.5.4 Other systems

We also tested basic functionality on OP3 [9], Innok Heros [7], Erle [4], and PR2 [46]. We were not able to immediately find bugs in these systems or were not able to follow the documentation to set up enough functioning use case scenarios. This does not mean that bugs do not exist in these systems, but may mean that we were not able to expose the correct interfaces for testing. In fact, for some of these systems, the documentation was limited and we were only able to bring up the teleoperation scenarios, which limited the scope of testing.

The three systems we do test encompass a wide breadth of applications (hobby, warehouse, research and development) and use different layers of the software stack (flight middleware, full robot control, configurable platform). We therefore believe that finding bugs in these systems shows potentially wide applicability of our tool and methods.

3.6 Invariants

To evaluate a test case, we say a test has triggered a failure if it has violated some invariant. A software crash, such as when a ROS node stops unexpectedly and outputs a core dump, can be framed as a special invariant ("a node shall never output a core dump") that is checked by looking for the presence of a core dump file. When we mention "crashes" or "crash invariants" in subsequent chapters, we mean software crashes, rather than a robot physically crashing into something.

In addition to the crash invariant, we also check for speed limit violations in each of the systems under test, according to the specified datasheet documentation of each system [2, 5, 13]. These limits are as follows:

Ardu linear speed shall not exceed $5.0m/s$ in the $x-$ or $y-$ directions. Note that, in the use

case scenarios provided by Ardu, a velocity with both $x-$ and $y-$ components had magnitude that sometimes exceeded $5.0m/s$ as a vector value, so we check each direction separately rather than computing the magnitude.

Fetch linear speed shall not exceed $1.6m/s$ in any $x - y$ plane direction (we compute the velocity by taking the magnitude of the (x, y) velocity vector).

Turtlebot linear speed shall not exceed $0.65m/s$ in any $x - y$ plane direction, and angular speed shall not exceed $180°$/s.

All of the systems published their velocities on the */odom* (in the case of Ardu, */mavros/local_position/odom*) topic, and our invariant checker parsed the test output to make sure the relevant values did not exceed the limits.

3.7 Framework Scope and Summary

In this chapter, we described challenges to testing autonomy systems, our approach to meeting these challenges, our methods of input selections, and an overview of the systems we tested. Because robots are stateful, distributed, temporal, and cyber-physical, we use a testing tool that uses the message-passing channels between nodes of the system and intercepts on nominal data. Instead of simply detecting crashes or hangs, we also detect speed limit invariant violations to enforce safety properties on a system.

3.7.1 Simplifications made in baseline testing

In our testing tool, we make several assumptions. Some of these are explored further in Chapters 5 to 7, and some are left as assumptions that are based on previous success in testing robots.

- We exclude time and sequence messages by default because, in practice, we have found that they are particularly fragile and prevent testing from finding deeper bugs. We do not exclude any other fields from testing in Chapter 5, but Chapter 6 explores how fragile fields can affect testing outcomes and how they can be detected.

- Our default nominal input percentage value of 20% has worked well to find bugs in the past, but a higher or lower percentage may be beneficial. Because we randomly perturb fields in a nominal bag, a replacement percentage that is too small might mean that we need to run more experiments to find a pertinent field or might miss multi-dimensional test cases altogether. Conversely, replacement percentage that is too high might mean that we are simply replacing too many of the data and timing assumptions that the autonomy system is making and prevent the automated tool from getting the robot into a non-trivial testable state. Chapter 7 explores how changing this percentage affects the tool.
- We rely on a startup script to get the robot into a testable state, and then provide a perturbed input. This may prevent the robot from getting into deeper testable states, but to mitigate this risk, we test several different use case scenarios. This means that we do not need to include a means of delaying test input mutation in the tool.
- We test the system as a whole, instead of testing inputs to each individual node. For the scope of this dissertation, we are concerned with how interfaces to a robot system may affect the entire system, and we find enough bugs to perform an analysis of our test input methods.
- We limit our tool to test ROS-based systems. While this prevents us from testing robots that are implemented without using ROS, most autonomy systems are written as a distributed system with channels of communication [59]. Therefore, the justification for the log message value substitution aspect of our tool can still hold for non-ROS-based systems. It also may be the case that our tests may find bugs in underlying ROS functionality, rather in than the systems under test themselves. However, this is still relevant to evaluating autonomy systems. Because the systems under test encompass a wide variety of applications, there is limited overlap in the ROS functionality tested across systems.

Section 8.1 further discusses the tradeoffs between ease of implementation and testing outcome for some of these simplifications. The speculation on future work in Section 9.3 describes

how these simplifications may be removed

3.7.2 Conclusion

This chapter lays the groundwork for the testing described in Chapter 4, which leads to the results we discuss in Chapters 5 to 7. We have outlined the basic challenges to testing autonomy systems, but the remainder of this dissertation takes a deeper look at test case input selection step of the process, in order to improve the efficiency and effectiveness of the testing tool.

Chapter 4

Metrics of Analysis and Procedure

We use this chapter to define metrics of comparison between test methods, which will form the backbone of the analysis in the remaining chapters. We discuss how the two metrics complement each other to give a richer analysis, and also provide some related definitions that explicitly encode a comparison between test methods. These metrics form the basis for comparing a set of test input generation methods. In the following chapters, we show how the metrics can be used for a deeper assessment of the merits of a given test method and can even be used to guide improvements to an existing method. We conclude this chapter by outlining our procedure for experimentation and analysis, which includes deriving efficiency values given an already-run suite of tests.

4.1 Metrics of comparison

At first glance, it may seem that it is enough to simply define a metric that favors a test method that triggers invariant violations with the least number of test cases possible. This metric is important in the sense that an efficient use of testing budget minimizes the number of test cases that do not trigger invariant violations. However, this metric can favor a test method that can trigger the same underlying bug over and over using different but similar data values. A complementary metric would count the number of unique bugs triggered. However, black box

testing inherently does not allow access to source code. Furthermore, the root cause of errors found using black box testing is unknown. All we can know is the set of inputs that triggered the error. Therefore, given two errors triggered by black box testing, it is difficult to determine whether they are due to distinct underlying issues in the source code. In other words, black box testing doesn't tell us if two separate invariant violations correspond to different bugs. This is a difficulty in assessing the efficacy of a black box testing method, because it is possible that one method triggers errors more quickly than another, but might be "shallow" in the sense that all of those errors are due to the same bug in the code, whereas the second method might trigger fewer errors, but they might be distinct bugs in the code. For this reason, it is necessary to evaluate black box testing on metrics other than "number of errors triggered per number of test cases." To encompass both speed and breadth of testing, we define our two metrics of efficiency and effectiveness.

4.1.1 Efficiency

Given a use case scenario, the **efficiency** of a given test input selection method is the reciprocal of the average number of test cases to first invariant violation (colloquially in this section, "number of tests to first failure"). "Average" is respective to the randomness inherent in generating a test, because the nominal inputs are replaced at random according to a percentage and the inputs are chosen randomly from a dictionary or randomly generated. In Section 4.3 below, we discuss how to derive the average number of tests to first failure given a test suite. Because we use the inverse of the number of tests to first failure, we define an efficiency of 0 for methods that have not uncovered any invariant violations for the given test budget.

Our main definition of efficiency is with respect to a use case scenario, but the measurement can be applied in other ways. For example, the average efficiency across components for a system can give a single number for the test input selection method efficiency for a system, and, as shown in Chapter 6, can be used incrementally within a scenario to visualize a marginal efficiency as failure-triggering fields are progressively removed from testing.

At first glance, a high efficiency is beneficial because it means less of the testing budget is spent on tests that do not find bugs. However, a test method with high efficiency may just be uncovering the same easy-to-trigger bug over and over. Efficiency therefore favors shallow, easy-to-trigger bugs, and we need a second metric, effectiveness, to differentiate between bugs found.

4.1.2 Effectiveness

The purpose of this metric is to estimate the number of unique bugs found, by diagnostically differentiating bug symptoms. Given a test method, and a system under test, we define **effectiveness** to be the number of unique symptoms found by a test method, given a symptom class. "Symptom class" is purposely broad, because it tries to predict whether two different found *symptoms* have different underlying *causes*, which is impossible to definitively determine in a black-box environment. However, we present heuristics that can provide guidance. For example, use case-wise effectiveness measures the total number of use cases with at least one invariant violation within a system. In Section 2.8, we reference fuzz tamers. In the language of that work, a distance function for two test cases can be defined as 0 if the symptoms for a given symptom class are the same, and 1 otherwise [33]. We give some examples of ways to describe symptom classes that likely distinguish underlying cases of bugs, while also addressing the risk that they do not:

- Use case scenarios crashed: errors in two separate use case scenarios are more likely to be due to different underlying bugs, because separate scenarios are more likely to have separate implementation. There is still a risk that the underlying bugs are the same, for example, if the bug exists in a common library referenced by both scenarios. However, if one testing method finds an issue in a scenario that another testing method cannot, the first method has exposed a vulnerability in the scenario that was missed by the second method.
- Types of invariant violations: errors that are classified as different invariant violations within a test suite (e.g. a speed limit violation vs an obstacle collision violation) might

come from different underlying causes because they correspond to different behavioral requirements for an autonomy system and might therefore be implemented separately. There is a risk that the components are not distinct, for example, if both the speed limit violation and object collision violations happen because the robot is provided an input that overrides its stopping capacity. However, different invariants often have different implied severity levels, and a test case that causes the robot to crash and cause harm to itself will arguably be prioritized over a test case that simply causes the robot to move too fast, and if one method finds more severe invariant violations, it may be more useful to developers.

- Crashes in different nodes of a system: This is a special case of the invariant violation above. In our work, we have found that a crash invariant does not mean that the entire system was brought down, but rather typically means that a certain node malfunctioned. For example, a crash in `move_base` is likely distinct in implementation than a crash in the laser scanner. Again, the caveat is that both nodes may reference the same underlying library. However, as in the use case argument, if one testing method finds an issue in a node that another testing method has not, it has exposed a vulnerability in that node that the other method has not.
- Number of types of invariant violations within a single test case: if a test case violates two different invariants, it probably triggers a different underlying bug than a test case that only violates one. If one testing method is able to generate more test cases that violate different invariants, it is likely finding more unique bugs. In addition, the bugs are arguably deeper or more significant, since these sorts of test cases are rare.
- Types of messages or fields within a message activating an error: a test activated on one type of field or message might be distinct from an error activated by another type of field or message. This is because these inputs can be, at the interface level, separately handled in the code. Here, the risk is that, according to the data path assumptions of the implementation, the two data types can potentially be related and lead to activation of the same

underlying bug. However, if one testing method uncovers errors using a field type that another testing method does not, this testing method has an exposed a vulnerability in that field type.

- Number of messages in an error-causing input: similar to the types of fields or messages above, if one error is activated using just a one-message input while another error is activated using a multiple-message input, the inputs handled by the code are different and therefore might be due to distinct bugs. In particular, the single-message input can be thought of as more "shallow," since it does not require a complex sequence of messages to be triggered. There is a risk that the same underlying bug might be triggered by a very particular one-message input but also in conjunction with another input (for example, if one message is a "gatekeeper," that is, a check on the first message lets some values through but also forces a check on the second message later). However, again, if a testing method exposes a longer sequence of messages that work together to trigger an error, there is value in that testing method.

Analyzing along these axes can be thought of analyzing effectiveness in more or less granular ways. We show concrete measurements for some of these metrics in Chapter 5. In particular, we describe how field effectiveness was measured in Section 5.1.

4.1.2.1 Overlap and exclusiveness

Two test methods may have similar exclusiveness, but still find different symptoms within a class. Because of this, we need the notion of overlap. Given two test methods and a symptom class, the **overlap** between test methods is defined as the number of shared unique symptoms across all invariant violations found by each test method. For example, use case-wise overlap would count the use cases in which both test methods found at least one invariant violation. Conversely, the **exclusiveness** of a test method T1 in with respect to the test method T2 is the number of unique symptoms on which T1 found at least one invariant violation and T2 did not. Note that the effectiveness of T1 is therefore the sum of the overlap between T1 and T2 and the

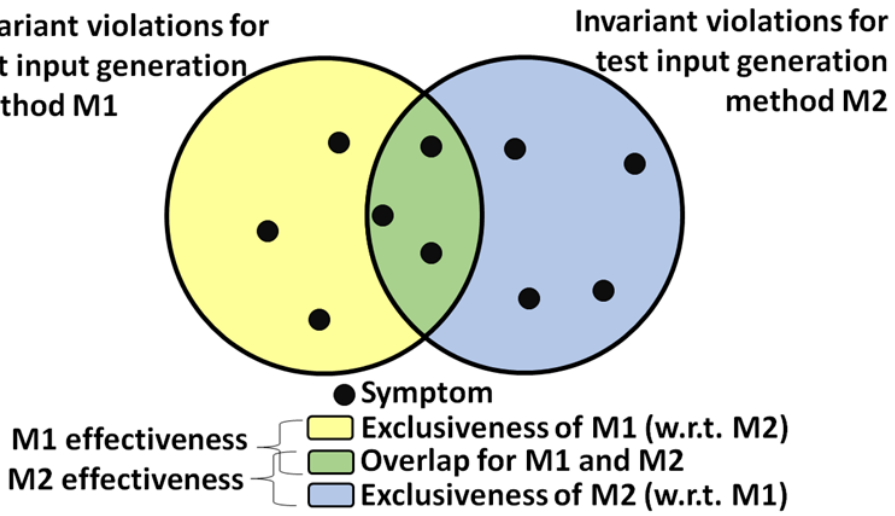

Figure 4.1: The relationship between effectiveness, overlap, and exclusiveness for two test methods, M1 and M2. In this diagram, M1 has effectiveness 6 and exclusiveness 3, M2 has effectiveness 7 and exclusiveness 4, and the overlap between M1 and M2 is 3.

exclusiveness of T1 with respect to T2. A component with higher exclusiveness has found more estimated unique bugs with respect to a symptom class and therefore likely has higher utility as a test method.

For cases where one failure is fairly difficult to trigger, it is possible that a test method cannot trigger it within the constraint of the budgeted tests. For example, if given infinite time, fuzzing would be able to trigger all bugs that can be activated at a given interface. For this reason, overlap and relative complement are metrics that depend on the given test budget.

The relationship between effectiveness, overlap, and exclusiveness is given in fig. 4.1.

4.1.3 Efficiency and Effectiveness as complementary methods

As mentioned above, effectiveness is necessary to define because a test method with high efficiency may just be finding the same shallow bug using slightly different inputs and therefore has less utility as a test method. However, a test method with a theoretically high effectiveness given a high enough test budget has little practical use if it is not efficient enough to find those

bugs within a given test budget.

Because efficiency and effectiveness optimize for different notions, it is necessary to consider both when evaluating a test method. There are, however, some situations in which the metrics are partially correlated. Extremely low efficiency might imply low effectiveness, because it describes the case where a test method has trouble finding any bugs at all. Similarly, high effectiveness for a reasonable test budget implies high efficiency, because high effectiveness means a high total number of bugs found and therefore a low expected number of tests to first failure. In Chapter 6, we show how a focus on both efficiency and effectiveness helps us determine which test method finds a diverse set of failures within a reasonable number of tests.

4.1.4 Advanced applications

The initial motivation of defining metrics of comparison is to discuss whether fuzz testing or dictionary testing fares better for a specific use case scenario on a given autonomy system. However, as we will show in Chapter 7, these metrics can be applied to any test input selection method and can even be used to compare three or more test input selection methods at once. More compellingly, we can also apply the metrics when exploring hybrid testing strategies. For example, as in Chapter 6, we can compare the total effectiveness and average efficiency across use cases of a system when fault-causing fields are iteratively removed from testing to give a progressive measure of efficiency. This may allow us to evaluate how efficiency changes as fragile fields are eliminated to allow the test method to find deeper bugs. We can even measure the efficiency and effectiveness of a hybrid strategy, for example, one in which a test input selection method is randomly decided according to a fixed probability before generating each test case. In Chapter 6, we explore these alternate strategies and use our metrics to explain their merits.

4.2 Testing experiment procedure

To compare the use of fuzz testing and exceptional value testing in autonomy systems, we tested the scenarios defined in Section 3.4. We created and executed 100 test cases for each of

these use cases according to the method in Section 3.2.2, and evaluated the robots for crashes and invariant violations. For any test inputs that triggered failures, we identified the relevant messages and fields to trigger the failure, and, in the case of crash invariants, evaluated which nodes were crashed.

Reproducibility of the results is assessed by replaying the failure-triggering input log on the system and checking for the same invariant violation. We checked reproducibility on all failure-triggering input logs of the system. We start with a fresh launch of the system for each test (by way of Docker containerization), and therefore do not need to account for previous tests contaminating system state and affecting the results of subsequent tests. This overcomes limitations addressed in previous robustness testing work, e.g. Ballista needing to run a suite of tests again in reverse to verify that side effects of a given test case did not affect subsequent test cases [70]. For quality control, we also ran nominal test cases (unmodified input logs of use case scenarios) to verify that they did not violate any invariants.

This methodology gives us a suite of 100 test cases for each test method. We can then calculate the efficiency and effectiveness of the test methods for the give use cases. In practice, we have encountered a high enough efficiency that we deem this number of tests to be sufficient. Section 6.3.2 explores running a larger test suite.

4.3 Application of metrics

Given a test suite of N test cases for a use case scenario under test, effectiveness is relatively easy to compute. Differentiating by symptom classes such as unique types of invariants violated is simple, as we can simply run the invariant checker and bin the failures by invariant type. Even more granular measurements, such as names of failure-triggering fields, can be done by first running the necessary diagnosis and then binning according to the metric.

Efficiency takes slightly more work, because it is the reciprocal of the *average* number of tests to first failure, rather than an absolute number. To compute this metric, we would have to run a number of test suites, count the number of tests to first failure, and average them. However,

instead of running many separate test suites, we can randomly shuffle our existing test suite and average number of tests to first failure from this set of random shuffles. This assumes that the test suite is a representative random sample of the test method for that test case. Instead of doing this shuffle experimentally, we can rely on the negative hypergeometric distribution [18] to compute this average, which describes the probability of having to drawing a certain number of failures before achieving m successes from a fixed set of successes and failures without replacement. If the size of the test suite is N and it contains K invariant-violating test cases, with the terminology in Balakrishnan and Nevzorov [18], we have the number of "successes" (invariant violations) in the set $r = K$, number of successes drawn before stopping as $m = 1$, and number of "non-successes" (non-violations) in the set as $b = N - K$. The value for the expected number of tests to first violation is thus the expected number of non-successful draws (as given by (10.9) in Balakrishnan and Nevzorov [18]) plus 1 for the successful draw, and the calculated efficiency of this set is the reciprocal:

$$\text{efficiency} = \frac{1}{\frac{mb}{r+1} + 1} = \frac{1}{\frac{N-K}{K+1} + \frac{K+1}{K+1}} = \frac{K+1}{N+1}. \tag{4.1}$$

4.4 Conclusion

In this chapter, we introduced our metrics for comparing test methods, and we discussed various caveats and tradeoffs between the metrics. We also explained the testing procedure by which efficiency and effectiveness are calculated for the analysis in Chapter 5. The discussion of these metrics lays the groundwork for the analysis in Chapter 6, where our metrics are used to analyze how well different hybrid testing strategies work and to make a recommendation towards one.

Chapter 5

Comparison of Test Input Generation

We have introduced our testing framework and two main test input generation techniques. In the previous chapters, we discussed the theoretical tradeoffs between dictionary-based testing and fuzzing. Namely, fuzzing does not eliminate any value from the pool of possible values but therefore risks oversampling; while dictionary-based testing may better estimate the distribution of failure-activating inputs but may risk undersampling by missing a value in the dictionary. In this chapter, we explore whether fuzzing or dictionary-based testing is more efficient and effective for our given systems.

5.1 Experimental setup details

We would like to provide an initial comparison of test input techniques for a system under test. To do so, we test each use case scenario of each system using dictionary-based testing and fuzzing, using the default replacement percentage of 20% and the setup described in Section 4.2. We then plot the efficiency of each input generation technique for each scenario under test.

From this data, we can directly measure *use case scenario effectiveness*, by counting the number of scenarios that have at least one failure-triggering test case. As discussed in Section 4.1.2, measuring effectiveness using different symptom classes gives various approximations of fault coverage, because effectiveness is a way of distinguishing faults by distinguishing the ways in

which they are triggered. *Use case scenario effectiveness* gives a broad notion of coverage before any deep diagnosis. *Invariant effectiveness* gives a secondary measure. However, when we diagnose the bugs found and identify the relevant fields in a failure-triggering input, we can measure field effectiveness. This gives a better estimation of the number of unique bugs each test method found. This is based on the assumption that different combinations of failure-triggering fields are more likely to correspond to underlying bugs in the code.

In order to do this deeper exploration, we applied the following diagnosis steps to each failure-triggering input we found:

1. Delta debug the failure-triggering input log in order to find the smallest sequence of messages that will trigger that failure. Our delta debugger is based on the Zeller work [115], and first splits the failure-triggering log into two subsets of roughly equal sizes. These subsets are replayed on the system, and if one causes an invariant violation, the algorithm has found a smaller failure-triggering input and begins anew on this subset. Otherwise, the large input is split into three chunks, and every sequential combination of these chunks is replayed (for example, if the three chunks are A, B, and C, this step would play six input logs, corresponding to messages represented by A, B, C, AB, AC, and BC). If this yields a smaller failure-triggering input, the algorithm begins anew with that input, and otherwise the algorithm splits the large input into more and more chunks, until the number of sequential combinations of the chunks exceeds the number of messages in the large input. At this point, the algorithm switches to a linear mode, where messages are eliminated one-by-one. The output at this step can give a measure of *message effectiveness* (the number of unique failure-triggering messages found by a test method).
2. Find the minimal set of relevant fields within the delta-debugged input that will trigger that failure. This is done using the delta debug algorithm, except by systematically changing the input field values back to the ones found in the nominal input log. This method makes the assumption that only perturbed fields contribute to a bug. In the case that two different

fields are necessary to trigger a bug but one needs the nominal value, this method will miss the field with the nominal value. This is contrasted with the HPSL method in Vernaza et al. [106], but is significantly computationally faster.

3. Classify bugs according to their minimal input log and relevant fields. That is, if two failure-triggering inputs have the same sequence of messages in their delta debugged input logs and the same set of relevant fields, they are highly suspected to trigger the same bug and both increase *field effectiveness*.

Because a distinct set of failure-triggering input fields likely corresponds to a distinct bug, field effectiveness is a more granular and arguably more accurate measure of fault coverage than use case scenario effectiveness.

This process above describes ways to measure the performance of test methods compared to each other. If one method has high efficiency and high exclusiveness with respect to the other method, we can conclude that it performs better on the system under test. If each method has cases where it outperforms the other, deeper analysis is necessary to determine why.

5.2 Results

5.2.1 Efficiency and use case scenario effectiveness

We plot the efficiency (inverse of average number of tests to first failure) for both test methods on each use case scenario in fig. 5.1. A higher efficiency is potentially better, because it means a test method takes fewer tests to uncover the first failure. Note that a bar of height 0 means that the test method found no faults for the duration of the test campaign. The graph also shows use case scenario effectiveness: any bar with a height above 0 means that the test method uncovered a fault in the scenario, which adds to the effectiveness.

As a caveat to these results, recall from Section 3.5 that the pick_place scenario in Fetch had a violation in the nominal input. Using delta debugging, we diagnosed this to an input of a `/move_group/goal` message and a `/pickup/goal` message. Because none of the

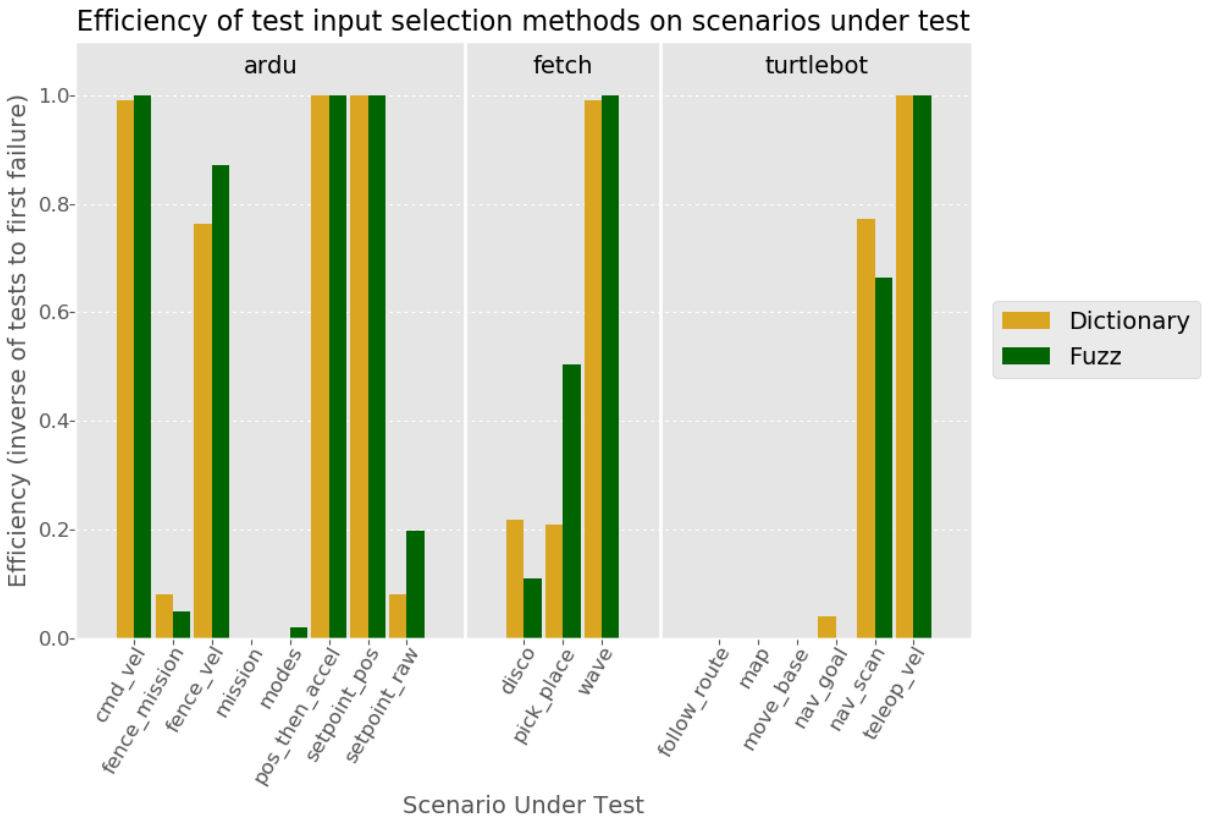

Figure 5.1: Efficiency comparison for fuzz and dictionary-based input selection.

dictionary and fuzz inputs had this combination of two messages as their minimal input log, we are confident that fuzzing and dictionary actually uncovered new bugs in this system. However, this scenario is excluded from analysis in Chapter 6 because the analysis relies heavily on field diagnosis and we cannot say for certain that the fields we examine were not contributing to the nominal failure.

One interesting result is that testing in some use case scenarios has an efficiency close to 1, which means only about one test on average is necessary to trigger a failure in that scenario. In Chapter 6, we discuss how this may be a symptom of masking, i.e., that a fragile field might always trigger an early crash in the scenario and prevent the test method from reaching deeper bugs. For now, we consider the exploratory results as-is, without accounting for masking, to give an overall assessment of the test methods.

From the graph, there is no clear winner between the test methods in terms of either efficiency or effectiveness. In the Turtlebot system, dictionary-based testing was able to find faults in a scenario where fuzz found no faults, but the opposite happened in the Ardu system. Furthermore, in some scenarios, fuzzing was more efficient at uncovering faults, whereas in others, dictionary-based testing was better. This suggests that testing autonomy systems would benefit from a combination of testing methods. To more concretely describe the strengths and weaknesses of the testing methods, we use the rest of this chapter to take a closer look at the test methods.

5.2.2 Invariant Effectiveness

Another way to measure effectiveness is by using the number of invariant types that are triggered using a specific test method. We plot the results of our efficiency calculations, broken down by invariant type, in fig. 5.2.

In this case, no use case scenario had two different invariant types violated in testing. However, as we will see in Chapter 7, measuring using invariant effectiveness allows us to make distinctions between the effectiveness of more complex test input generation techniques.

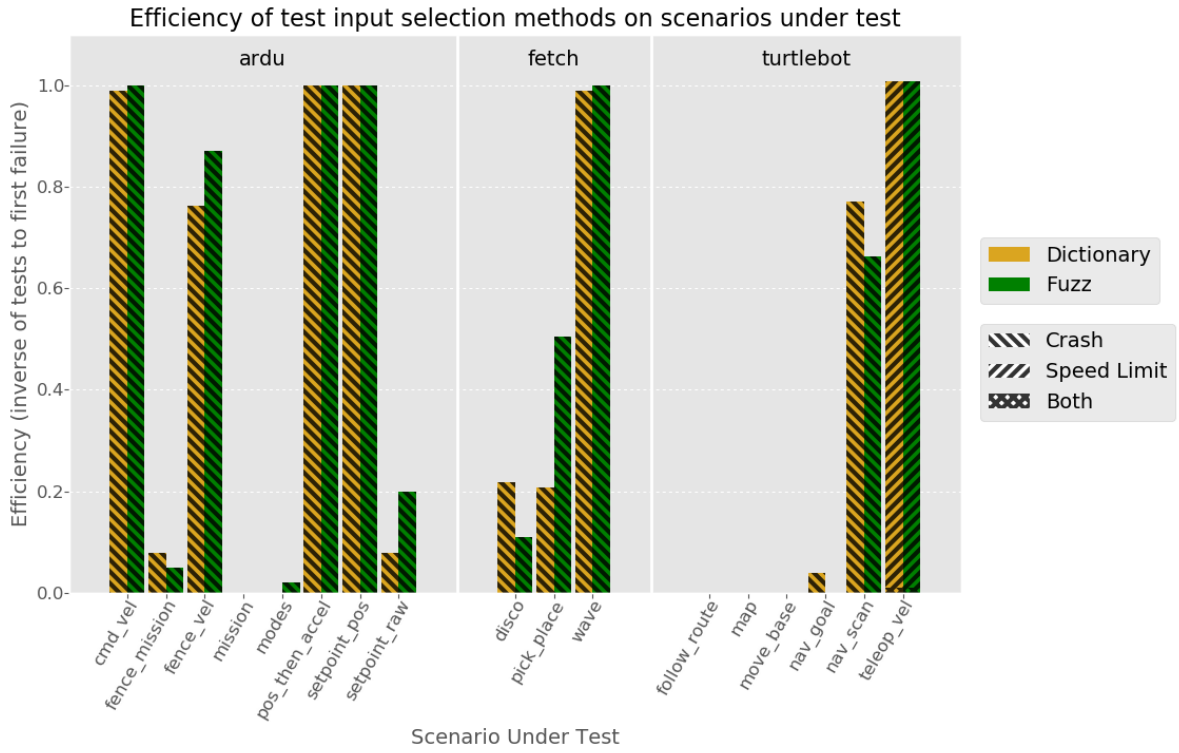

Figure 5.2: Efficiency comparison for fuzz and dictionary-based input selection, broken down by invariant type.

Table 5.1: Number of failure-triggering fields per scenario, by test method

Scenario	Dictionary only	Fuzzing only	Both
Ardu: cmd_vel	0	0	2
Ardu: fence_mission	1	0	1
Ardu: fence_vel	0	0	2
Ardu: modes	0	1	0
Ardu:pos_then_accel	0	1	1
Ardu: setpoint_pos	0	0	2
Ardu: setpoint_raw	1	1	2
Fetch: disco	1	0	1
Fetch: pick_place	0	0	2
Fetch: wave	0	0	3
Turtlebot: nav_goal	1	0	0
Turtlebot: nav_scan	0	0	1
Turtlebot: teleop_vel	0	0	2

5.2.3 Diagnosis and field effectiveness

We provide an overview of the fields that trigger a failure in each use case scenario in table 5.1. This table counts the total number of fields discovered to trigger failures by each method. For example, if some test value provided to a velocity field caused a scenario to exhibit a core dump, and some test value to goal position field in another test case caused that scenario to also exhibit a core dump, the field effectiveness of the test method on that use case scenario would be 2. Appendix A gives the actual names of the fields for this result. The numbers in each cell of the "Dictionary" and "Fuzz" columns represent the exclusiveness of the dictionary and fuzz methods, respectively, while the numbers in the "Both" column represent overlap.

In the Fetch system, dictionary-based testing exhibited non-zero exclusiveness with respect to fuzzing, while fuzzing did not find any failure-triggering fields that dictionary did not find. In the other two systems, each method was able to discover a failure-triggering field that the other did not.

Another observation is that the use case scenarios that exhibited the highest efficiency in Section 5.2.1 do not always have the highest field effectiveness. This provides further evidence to the idea that the scenarios where the test methods had high efficiency may be particularly fragile, and that the test methods are always finding the same shallow bug(s) in those scenarios.

5.2.4 Discussion of exploratory results

In each of Sections 5.2.1 to 5.2.3, we showed that both fuzzing and dictionary-based testing have instances of higher efficiency or effectiveness. In fact, each method was able to find failure-triggering fields that that the other did not. The main conclusion of these exploratory results is that there is no clear winner between the two test methods. This motivates Chapter 6, which proposes and evaluates several hybrid testing methods.

5.3 Follow-up: Input value efficiency

The exploratory testing in this chapter found a high incidence of field overlap between testing results. We follow up by asking whether, given cases of overlap, one method is still more efficient than the other. That is, even if both methods can find the same thing, is one faster than the other and therefore worth favoring? We answer this by measuring the efficiency of the input value sets themselves, for a given failure-triggering field.

5.3.1 Procedure

We classified failures using their relevant triggering fields in Section 5.2.3. For each failure-triggering test case, this yields a minimal input bag and a list of failure-triggering fields. By resetting all values in the minimal bag to the ones in the nominal input, and then manipulating the

values of only failure-triggering field(s), we can discover which input values trigger the failure. For example, it may be that negative values sent to a velocity field trigger a speed limit violation for a given use case scenario. For a given input, we systematically try all values in the exceptional value dictionary for that field, and record which values trigger a failure and which do not. We repeat this with fuzzed values, such that the number of fuzzed values tried is equal to the number of dictionary values tried for a given input. We do this over all distinct sets of failure-triggering fields for each use case scenario.

We find that fuzzing and dictionary-based testing can each perform better than the other method because of the nature of the bug-triggering input sets. We discuss the following cases of bugs:

- Bugs that require an input that can be described as a class with many members, for example, floating point values from 0 to 1. In general, fuzzing is more likely to generate values from this set.
- Bugs that require an input that can be described as an edge case, i.e. the input is a specific edge case value, such as INF or a single bad number.
- Bugs that can be triggered by a small set of inputs, such as integers that represent an ENUM-like object, or bitmasks.
- Bugs that require a combination of inputs from the above categories.

We provide illustrative examples of these categories of bug-triggering input sets to show that they occur in real systems and to illustrate cases where one testing method performs better than another. The inputs that were tried are plotted as a histogram. Values are plotted according to their distribution of the machine representation of the numbers (i.e. floats are plotted logarithmically because of the exponent bits, and integers are plotted linearly). We present the sets generated by fuzzing and the sets of the exceptional value dictionary as separate graphs, with red bars denoting the number of invariant violations from values in a specific bin.

5.3.2 Bugs that require an input that can be described as a class

- Field **`max_velocity_scaling_factor`** in Fetch: This field triggered bugs in both the disco and wave scenarios. That is, when this field was given a value from the red bars in fig. 5.3, it caused the system to exhibit a core dump in the `move_group` Robot Operating System (ROS) node. This field is a 64-bit `float`. Figure 5.3 illustrates the inputs that triggered this bug, for both exceptional values and fuzzing, plotted on a log scale.

  With our randomly generated set of fuzz values, the proportion that triggered an invariant violation was 0.243. With the exceptional value dictionary, the proportion was 0.012. Note that all of the values that trigger an invariant violation are between the values 0 and 1. The theoretical probability of generating such a value (i.e. with sign 1 and biased exponent between 0 and 1023) is 0.25. This example illustrates why fuzzing can be a more efficient test method to uncover certain kinds of bugs: while it is possible to skew the input dictionary to more proportionally include values between 0 and 1 to more efficiently trigger this bug, this bloats the dictionary and as a whole makes it less efficient at using edge case values to trigger bugs. On the other hand, it is an easy target to hit with random fuzz inputs. As shown in the cases in this subsection, fuzzing performs well for bugs of this variety, for various classes of input values. In the subsequent sections, we discuss how a dictionary approach is best used to discover other classes of bugs. This motivates using both methods together to create higher-efficiency test campaigns.

- Field **`twist.linear.x`** in Ardu: Interestingly, we also found bugs that were triggered in fuzzing by almost the complement of the set above. For the cmd_vel scenario, we found that values outside of the range 0 to 1 were likely to trigger an invariant violation (in the form of a core dump in the `arducopter` ROS node of the system), as illustrated in fig. 5.4.

  Note that, for the dictionary values, even the majority of values outside of the range 0 to 1 failed to trigger a violation. This indicates that we cannot categorically say that a value

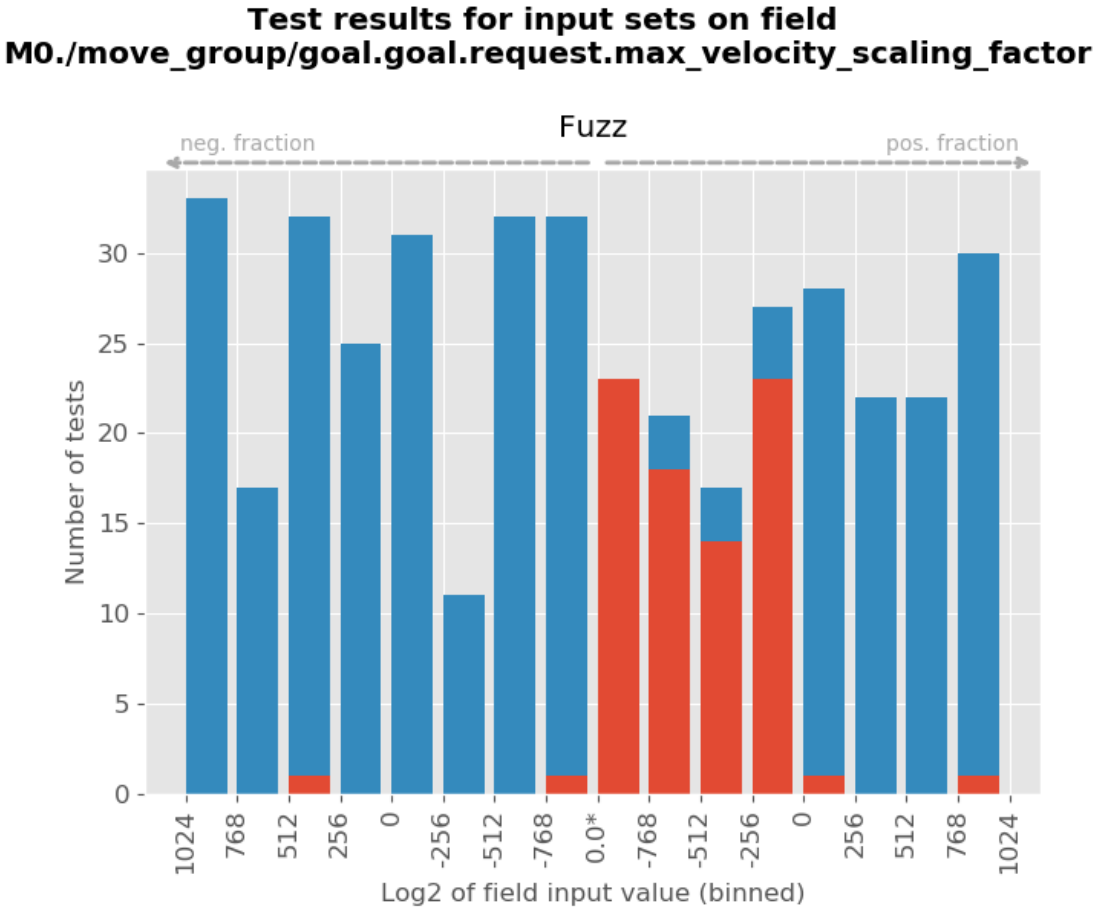

Figure 5.3: Input set comparison for `max_velocity_scaling_factor` *(cont.)*

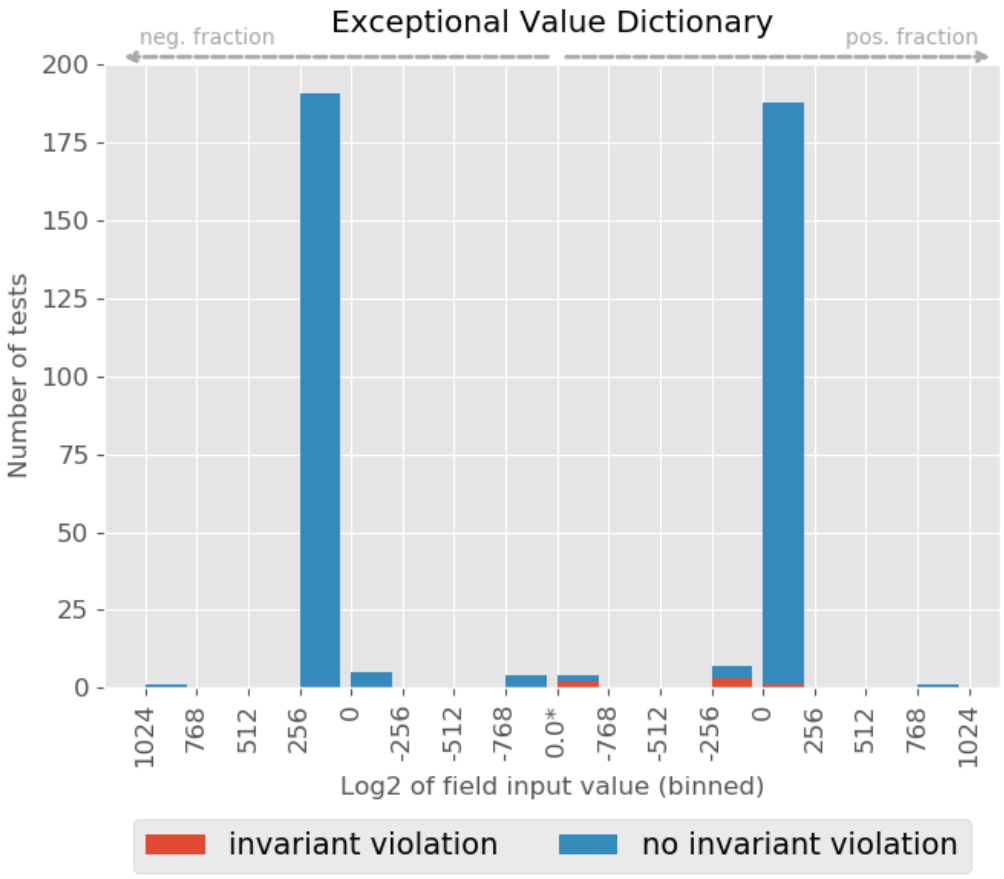

Figure 5.3: Input set comparison for `max_velocity_scaling_factor`

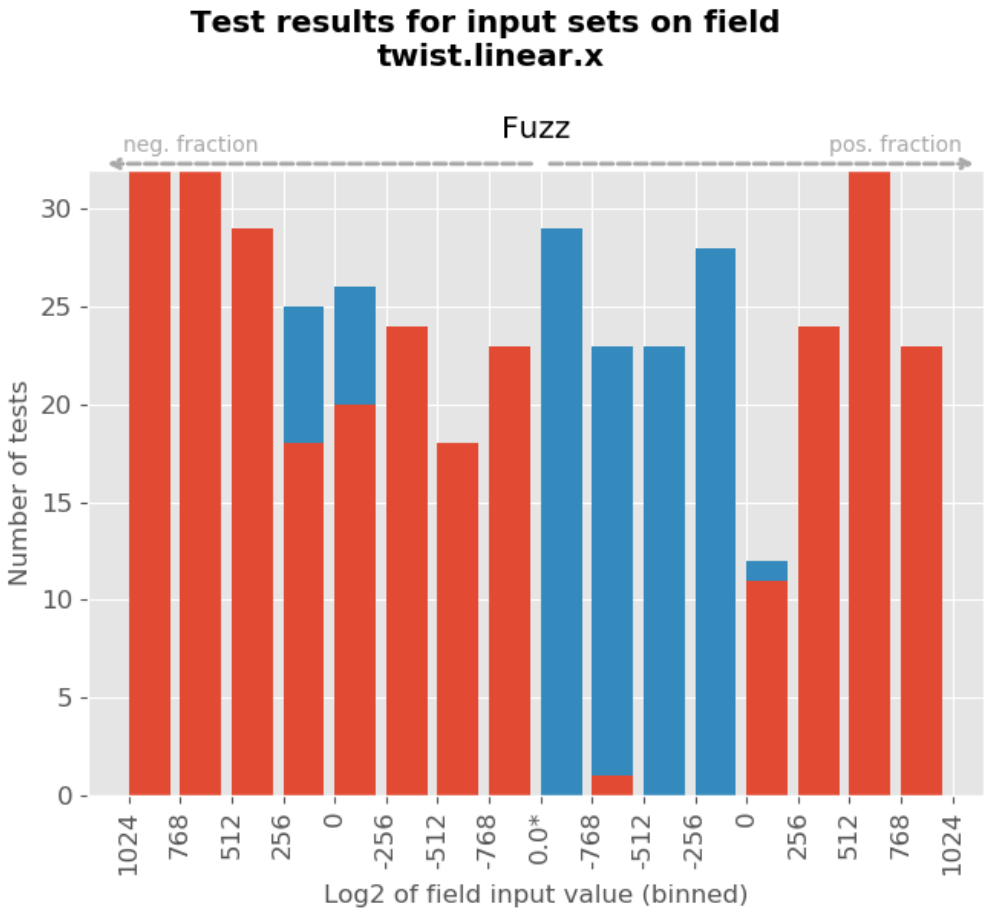

Figure 5.4: Input set comparison for `twist.linear.x` *(cont.)*

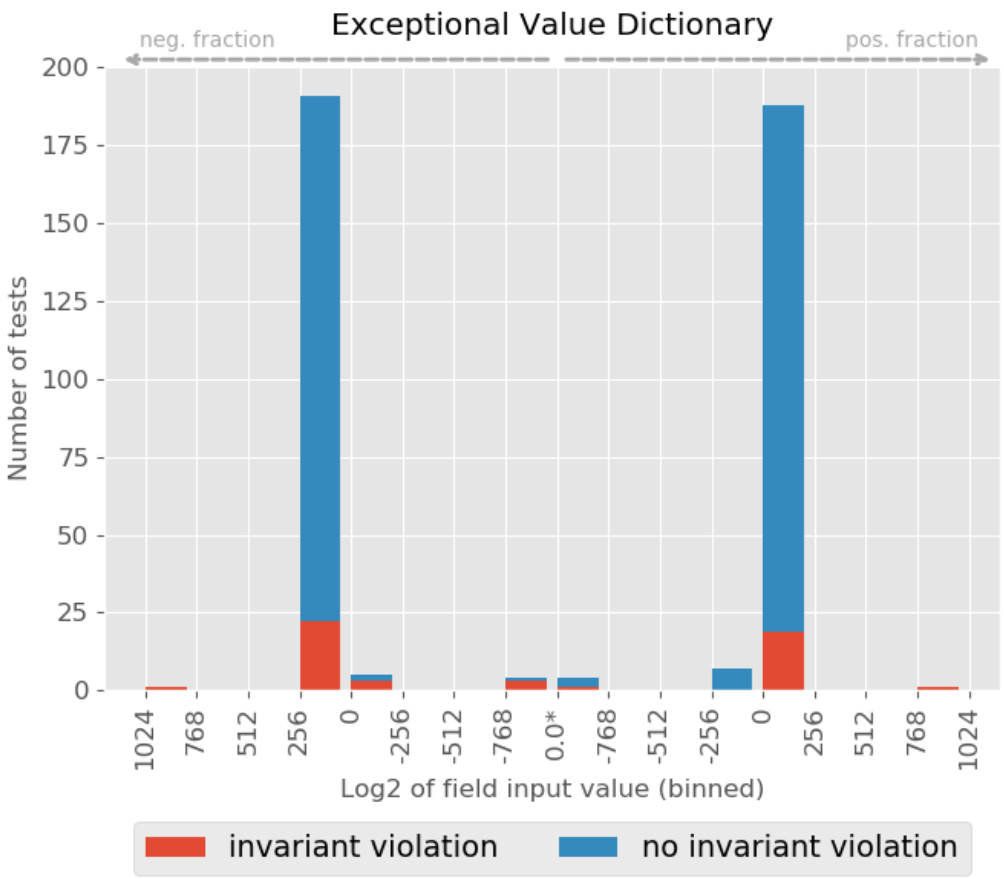

Figure 5.4: Input set comparison for `twist.linear.x`

outside of the range 0 to 1 will trigger a violation for this specific bug. For example, the very small value $6.7e10^{-181}$ triggered no invariant violation, while other positive values above and below it did.

- Field **`local.pose.orientation.z`** in Ardu: Some bug-triggering input sets are less intuitive to describe. For example, fig. 5.5 illustrates the inputs that trigger an invariant violation (in the form of a core dump in the `arducopter` ROS node) in the pos_then_accel scenario.

  There seems to be a piecewise distribution of failure-triggering values for this field. It may be that different ranges of values trigger different underlying bugs, or that the underlying computation is complicated and only succeeds on specific sets of inputs. However, we do not classify this as an "edge case" input, because the proportion of inputs that triggered this failure using fuzzing was 0.489 (compared to 0.035 for exceptional dictionary testing).

5.3.3 Bugs that require an input that can be described as an edge case

We have not found bugs of this class in the systems and scenarios we tested. Intuitively, this would be a single edge case value that can cause a computation error. These bugs do exist in robotics systems, as the previous testing efforts of the Automated Stress Testing for Autonomy Architectures (ASTAA) project have found bugs that are triggered by the value 0, presumably used as a divisor [59]. These values are almost impossible to generate using fuzzing (probability of $1/2^{64}$ to generate a 0 for a 64-bit integer) and therefore justify their inclusion in an exceptional value dictionary. Hence, while the above section makes it seem that a dictionary is less efficient at triggering certain kinds of bugs, a dictionary with a small set of edge case values can be beneficial in increasing the effectiveness of a test campaign by uncovering bugs that fuzzing is highly unlikely to find.

Note that `NaN` does not strictly qualify as an edge case, because fuzzing will generate it $2^{-8} - 2^{-31} \approx .39\%$ of the time for a 32-bit float. However, including it in the exceptional value dictionary of any size below 2^8 (256) will make it more likely than fuzzing to be used as a

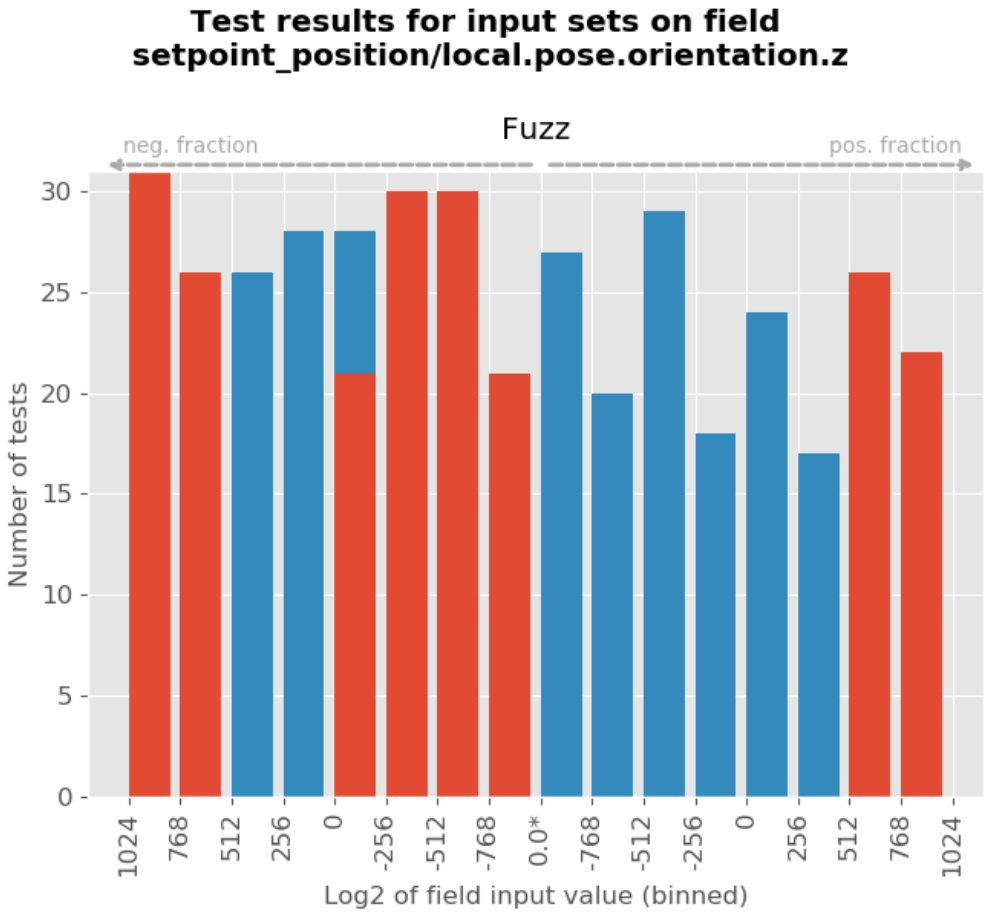

Figure 5.5: Input set comparison for `local.pose.orientation.z` *(cont.)*

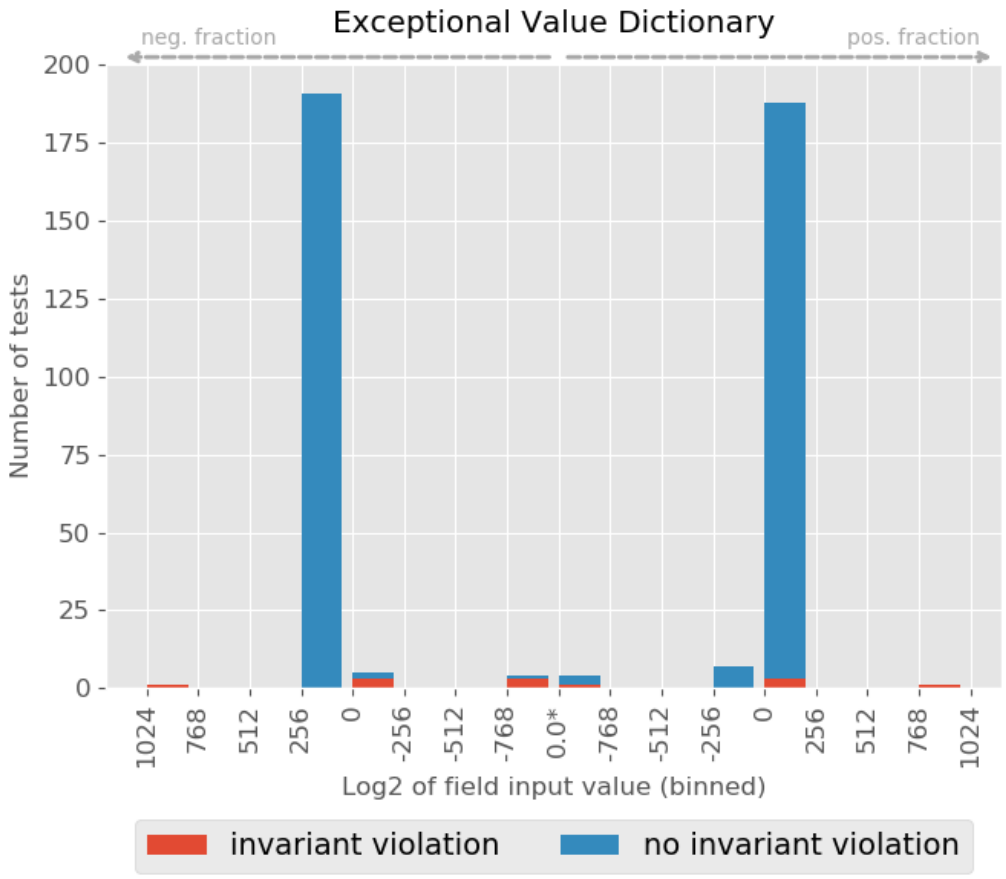

Figure 5.5: Input set comparison for `local.pose.orientation.z`

perturbed value. Because `NaN` has triggered bugs in robots in the ASTAA project in the past, we recommend including `NaN` in the dictionary.

5.3.4 Bugs that can be triggered by a small set of inputs

`local.coordinate_frame`: We have not found bugs that are triggered by a standalone set of inputs from this class, but one of the parameters for the input set discussed in Section 5.3.5 below is an 8-bit unsigned integer that triggers the most invariant violations when it is set to a value from the set $\{8,9\}$. On examination, this parameter is interpreted as a bit mask. The probability of generating these values for fuzzing is $2/2^8$. Because the specific parameter will not be perturbed for every test case, the probability of triggering this bug using fuzzing is much lower than with a dictionary that includes a set of small integer values.

5.3.5 Bugs that require a combination of inputs from the above categories

`local.coordinate_frame` and one of {**`local.velocity.`**$*$, **`local.position.`**$*$, **`yaw`**, **`yaw_rate`**, **`type_mask`**}: by sampling pairs of values for this combination of fields, we found that this bug performed best when the coordinate frame value was drawn from a dictionary and either the type_mask value was generated using the dictionary or the velocity, position, yaw, or yaw rate value was generated using fuzzing. In this case, we suspect this is because, in combination, coordinate frame and type_mask are triggered by small sets of inputs and benefit from dictionary-based testing, and the other fields is triggered by a class of inputs. This suggests that some testing may benefit from type-aware input generation, such as fuzzing for floating point values and a dictionary for 8-bit or 16-bit unsigned integer values.

5.3.6 Takeaway from input value set analysis

We have shown examples of bugs that are more efficiently discovered using fuzzing, and bugs that are more efficiently discovered using dictionary testing. Some of the results suggest that dictionary testing would benefit from a small dictionary that focuses on edge case values.

We evaluate this claim later, in Section 7.3. We also explore ways to tailor the dictionary to robotics systems in Section 7.6.

5.4 Discussion

In this chapter, we compared dictionary-based testing and fuzzing on the autonomy systems under test. We found that there are tradeoffs in efficiency and effectiveness between the two methods, and performed some diagnoses to better understand these tradeoffs. Ultimately, we found that the strength of fuzz testing lies in its generalized approach that can efficiently discover bugs that can be triggered by broad ranges of input values, whereas the strength of dictionary testing lies in its specialized use of edge case values. We also noted number of scenarios where the test methods exhibited a high efficiency, which may be an indication that deeper bugs may be masked by these fields.

We conclude that fuzzing and dictionary-based testing have significant enough differences in efficiency, use case scenario effectiveness, and field effectiveness to warrant a hybrid approach. Even in the cases where the methods exhibit field overlap, one method can be more efficient than the other, based on the underlying data assumptions of the field. In the next chapter, we use these observations to motivate and evaluate a hybrid testing approach that uses the advantages of both testing methods.

Chapter 6

Hybrid models

From the previous chapter, we have some evidence that fuzz testing and dictionary-based testing have different strengths. For example, we found that dictionary-based is more efficient at discovering vulnerabilities in fields that use an `enum`-like semantic to encode state. At a higher level, we saw that both fuzzing and dictionary-based testing each exhibit non-zero exclusiveness, that is, they each find things that the other method misses. Furthermore, a high efficiency in some use case scenarios under test does not necessarily correlate with a high field effectiveness and causes us to suspect that some deeper bugs may be masked by fragile fields. This chapter explores whether a hybrid model that iteratively excludes fields from testing and uses both test methods together performs better than a purely fuzz- or purely dictionary-based strategy.

6.1 Research questions

The exploratory study of Chapter 5 motivated the following questions:

1. Are bugs masked, that is, does eliminating failure-triggering fields from testing lead to an increased overall field effectiveness?
2. Does a basic strategy that chooses randomly between fuzzing and dictionary-based testing at each step of testing perform better than a standalone fuzz-only or dictionary-only strategy?

3. Does weighting the random strategy from question 2 towards one method consistently perform better than a basic 50/50 weighted model?
4. Does a sequential strategy, that is, one that performs fuzzing first followed by dictionary-based testing (or vice versa) perform better than the optimal (if any) weighted strategy?

Question 1 is necessary because it may affect the way we determine the field effectiveness of a test method and therefore affects the way we answer subsequent questions. This question is answered in Section 6.3. Questions 2-4 explore various hybrid methods and are explored in Sections 6.4 to 6.6, respectively. The previous chapter leads us to hypothesize that the answers to questions 1 and 2 are "yes." This leads naturally to question 4 – performing testing sequentially may allow one method to more efficiently eliminate fragile fields and allow the other method to find deeper bugs.

If we are able to determine one hybrid strategy that is better than the others, a natural next step is to examine the failure-triggering fields uncovered by each strategy to determine why the strategy is better, and if it can be improved further. We explore this in Sections 6.6.1 and 6.7.

6.2 Experimental setup

To answer **question 1**, we run additional tests using each test method, where previously discovered failure-triggering fields are excluded from testing. We iterate on this process, running campaigns until no more failures are found within a fixed number of consecutive tests. For example, if the initial campaign of fuzz tests from Section 5.2.1 found `velocity` and `position` to be failure-triggering fields, we run a new campaign of fuzz tests with `velocity` and `position` excluded from the test value substitution step of testing (as described in Section 3.2.2). That is, the `velocity` and `position` fields in the nominal input bag are left as-is, while the remainder of the fields may be replaced by fuzzed values. If this new campaign of tests finds `joint_state` to also be a failure-triggering field, we run a new campaign of tests with the `velocity`, `position`, and `joint_state` fields excluded from testing. We iterate on this

process and keep track of the total number of failure-triggering fields discovered, until a test campaign is run with no violations found. If this number is higher than the original effectiveness of the method, we say that a masking effect is indeed present.

To answer the subsequent questions, we expand on the iterative field elimination to create an average model of testing a system:

1. Begin with a data set that, for each use case scenario, contains tests for a given input selection method and list of excluded fields. Each configuration of {fuzz, dictionary} and failure-triggering fields found in testing has 50 completed experiments. In subsequent steps, the particular subset in use will depend on the results of the previous steps that were modeled. A full data set like this is required for comprehensive modeling.
2. At each step of the model, we choose a random use case scenario to advance, according to its modeled state in a **test-diagnose-exclude cycle**:
    a. **Test**: randomly select a test case based on the current list of excluded fields for the use case scenario and particular test strategy. For example, if the model has excluded the `velocity` field for the scenario and is performing a 50/50 random strategy, it would randomly choose between fuzzing and dictionary and then randomly select a test case from the corresponding set that has the velocity field excluded.
    b. **Diagnose**: if the selected test case resulted in an invariant violation, take note of the corresponding failure-triggering field(s) and increase the cumulative field effectiveness of the model by 1.
    c. **Exclude**: Add the failure-triggering field(s) to the list of the excluded fields for that scenario.
3. When a given use case scenario reaches 100 consecutive test cases with no invariant violations, remove the use case scenario from consideration by the model.
4. Repeat steps 1-3 until no use case scenarios are left to model.

5. Repeat steps 1-4 200 times. Graphically speaking, each run of steps 1-4 gives a step function, where the x-axis is the number of test cases run and the y-axis is the cumulative field effectiveness. Each step up of the step function happens when the test-diagnose-exclude cycle in step 2 finds a failure-triggering test case and adds to the cumulative field effectiveness. Averaging the cumulative effectiveness at each step of testing (that is, averaging the 200 step functions generated by 200 runs of steps 1-4) and plotting the result gives a curve where the maximum y-value is the expected cumulative effectiveness of the test strategy and the slope at any point of the curve represents the expected marginal efficiency.

It is important to distinguish the variables in these steps. When we say **runs of the model**, we mean the number of times we modeled steps 1-4 to generate the average result (in this case, we use 200). When we say **consecutive tests without violation**, we mean a set threshold of consecutive number of tests cases without violations that, when seen, causes one run of the model to terminate.

This graph is motivated by a simple testing and triage workflow within the software development process. Namely, we consider the behavior where a bug is found, diagnosed to the fields that are used to trigger the bug, and then logged into a bug database. Because testing and debugging may be handled by different personnel on a software project, the tester might have to wait for the bug to be fixed to continue testing. However, as to not waste time and to find different bugs or different ways to trigger the same bug, the tester may simply remove the relevant fields from the next round of testing and proceed with testing before the bug is fixed. Intuitively, the graphs described by our process show the number of unique sets of relevant fields (i.e. the unique entries in a bug database) found given a certain number of tests using this sort of workflow.

If the answer to question 1 is "yes," the subsequent questions are answered using the cumulative average model with failure-triggering fields excluded as they are discovered.

To answer **question 2**, we apply the cumulative average model where the strategy in step 2a. is a 50/50 random strategy, that is, a fuzz test case or a dictionary-based test case is chosen

with equal probability. If the average model achieves greater cumulative effectiveness than a fuzz-only or dictionary-only strategy, we consider the 50/50 random strategy to be better.

Answering **question 3** is similar, by changing the probability of choosing a fuzzing versus a dictionary-based testing test cases from 0.5 to values between 0.1 and 0.9.

For **question 4**, we model the fuzz-first strategy by always choosing a fuzzing test case at step 2a, until the model shows that no invariant-violating test cases were found for 100 consecutive tests. The model then always chooses a dictionary-based test case for that scenario for the remainder of the calculation. The dictionary-first strategy is the same, but with fuzzing and dictionary-based testing switched. If the average model achieves greater cumulative effectiveness than the optimally weighted random choice strategy, we consider it to be better.

6.3 Cumulative model

Table 6.1 shows the results of the iterative exclusion strategy described above, as compared to the initial field effectiveness results (from Section 5.2.3) for both fuzzing and dictionary-based testing. The entries in the table correspond to field effectiveness, that is, the number of unique fields found by each method. The bold entries in the table show where iteratively excluding fields showed an increase in field effectiveness, that is, excluding fields from testing discovered new fields.

Because it is possible that this increase effectiveness was due to the increased size of the campaign of tests, we report the average maximum field effectiveness after 200 runs with the average model, and the average number of tests it took to plateau, for the bolded entries from table 6.1 in table 6.2. That is, if a single run of the model reached a maximum field effectiveness of 3, and then found no violations for the remainder of the run, we save the number of tests it took to reach the field effectiveness of 3. We average this number across runs to get an estimate of how many tests it takes to plateau on finding unique failure-triggering fields. We use this calculation because not all runs of the model reach the same maximum field effectiveness. Runs with certain subsets of excluded fields do not find faults that other subsets do, whether due to

Table 6.1: Field effectiveness of test methods, before and after iterated exclusion

Scenario	Dictionary	**-excluded**	Fuzzing	**-excluded**
Ardu: cmd_vel	2	2	2	2
Ardu: fence_mission	2	2	1	1
Ardu: fence_vel	2	**4**	2	2
Ardu: modes	0	0	1	1
Ardu:pos_then_accel	1	**2**	2	2
Ardu: setpoint_pos	2	2	2	2
Ardu: setpoint_raw	3	**5**	3	**5**
Fetch: disco	2	**3**	1	**2**
Fetch: wave	3	**4**	3	3
Turtlebot: nav_goal	1	1	0	0
Turtlebot: nav_scan	1	1	1	1
Turtlebot: teleop_vel	2	2	2	2

Table 6.2: Average field effectiveness and number of tests to reach effectiveness plateau

Scenario and test method	average field effectiveness	Tests to reach effectiveness plateau
fence_vel, dictionary	4.00	55.8
pos_then_accel, dictionary	2.00	3.73
setpoint_raw, dictionary	2.51	29.98
setpoint_raw, fuzzing	3.08	31.02
disco, dictionary	2.50	32.22
disco, fuzzing	2.21	18.59
wave, dictionary	2.03	5.59

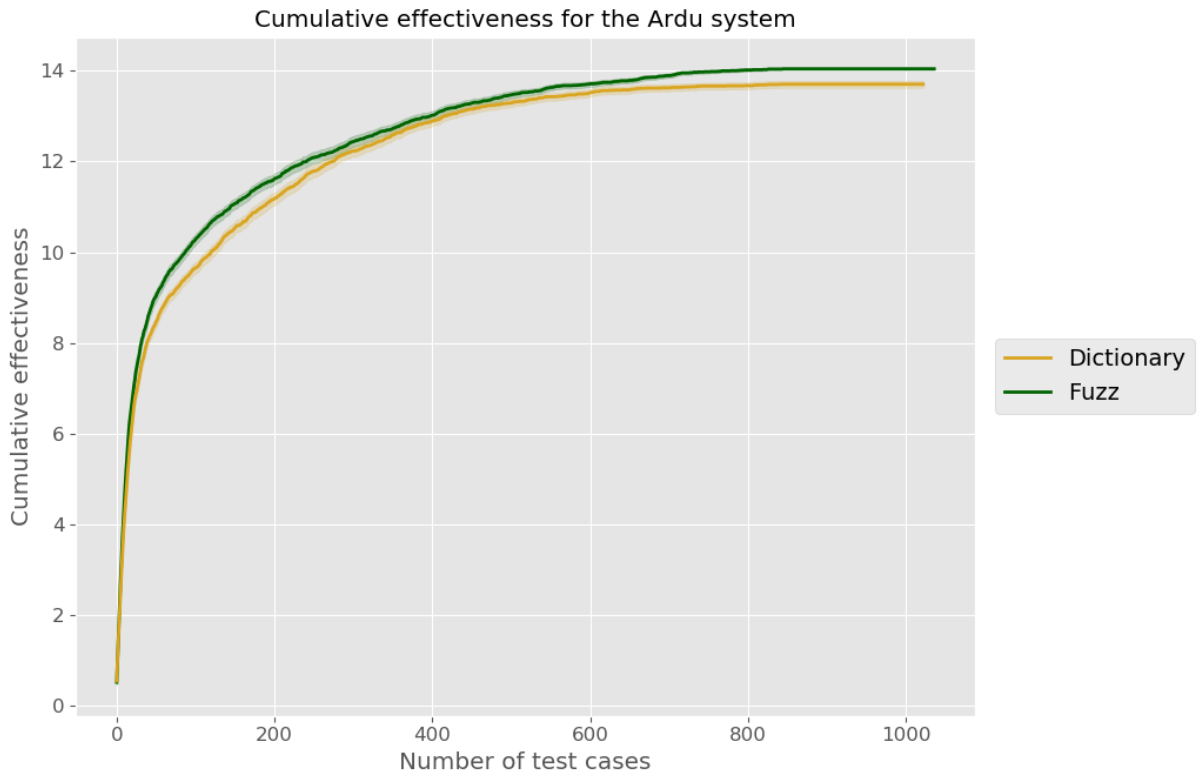

Figure 6.1: Cumulative graph for Ardu, with iterative failure-triggering field exclusion

masking effects or due to a small campaign size.

The reason that the setpoint_raw dictionary entry is lower than in table 6.1 is because finding some of the fields has very low efficiency, such that they were not discovered in some of the test campaigns. The number of tests in in the rightmost column is consistently lower 100, which was original test campaign size in Section 5.2.3.

We display the iterative field elimination average model graphically for both fuzzing and dictionary-based testing, in figs. 6.1 to 6.3. This shows how many tests are, on average, necessary to reach a given expected effectiveness. The results of this section show that, indeed, a cumulative average model is justified because shallow bugs mask deeper bugs.

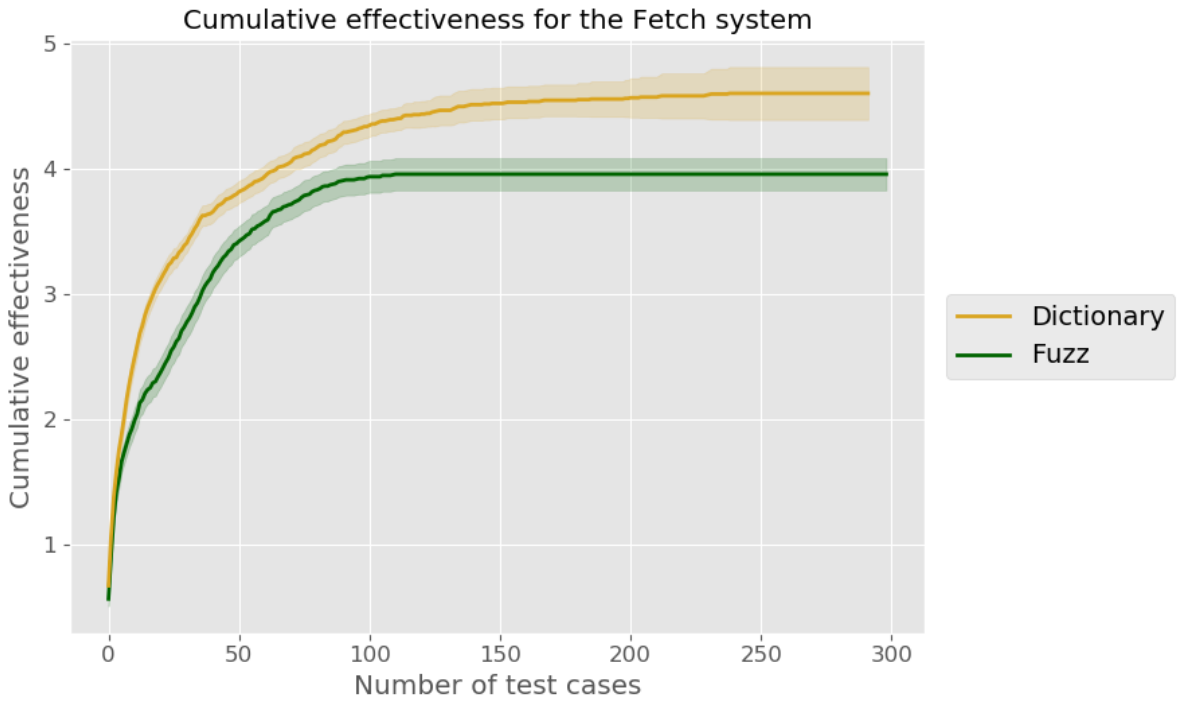

Figure 6.2: Cumulative graph for Fetch, with iterative failure-triggering field exclusion

6.3.1 Discussion of budget

One question that arises from these graphs is whether additional testing is worth the extra tests. For example, if one were to test well beyond 100 tests on the Turtlebot system (as in fig. 6.3), they would not see a gain in field effectiveness. However, it takes almost 500 tests to find one more failure-triggering field using dictionary-based testing. The y-value of each curve for a given x-value can give the expected field effectiveness of each test method with the test budget of the given x-value, but the real merit of this visualization is in observing a what y-value the curves level off. If one test method is able to reach a higher maximum expected field effectiveness, that means it triggered failures in fields that the other method did not. As per the discussion in Section 4.1.2, this means that that test method may have found a larger number of unique bugs. If the Turtlebot system had only been tested and debugged using fuzzing before being deployed, it would likely have the remaining vulnerability. This may pose a safety or at least monetary risk to the project. Hence, when examining the hybrid strategy graphs below, it is

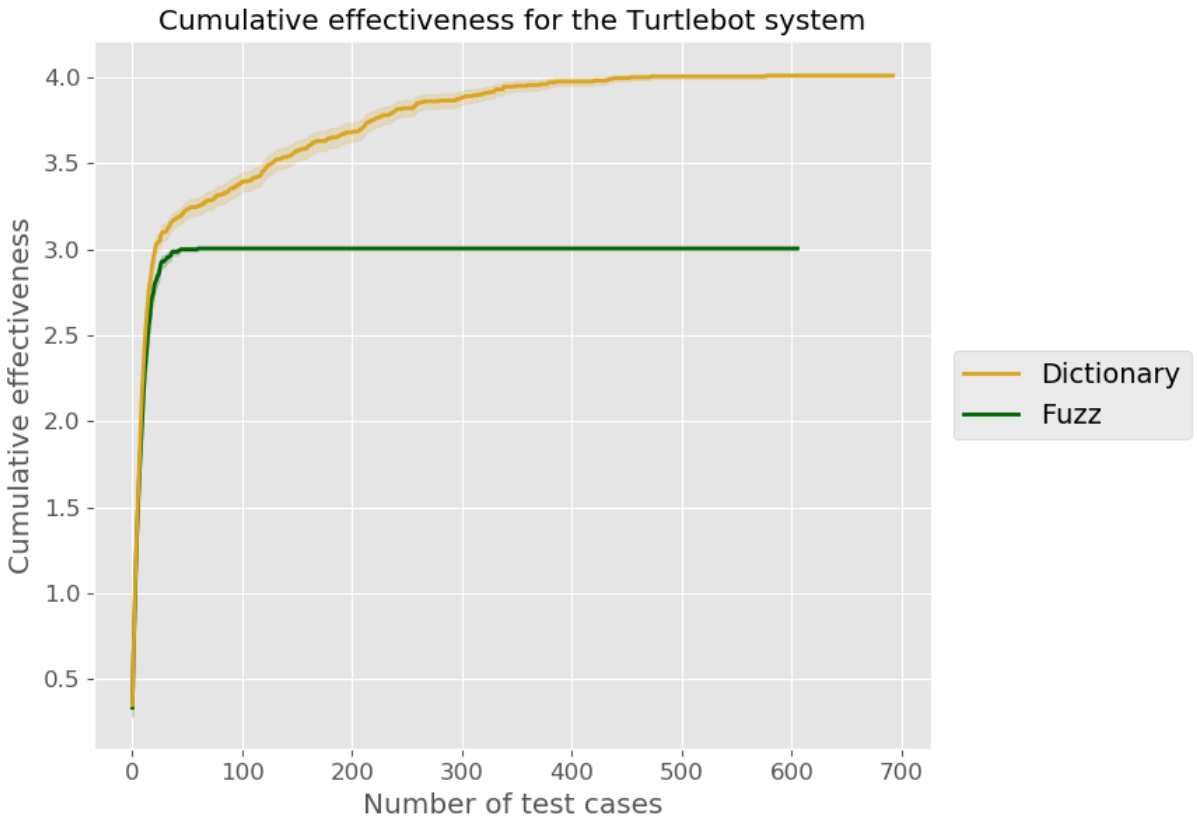

Figure 6.3: Cumulative graph for Turtlebot, with iterative failure-triggering field exclusion

important to view gaps between the maximum y-values of each curve as possible undiscovered bugs. A method that is not efficient enough to find the maximum number of relevant fields for a given test budget or not effective enough to find them at all with any reasonable budget will not achieve proper bug coverage.

6.3.2 Cumulative average model and large campaign analysis

By applying iterative field exclusion, we were able to reach a point where each use case scenario was tested with a campaign of test cases that yielded no more invariant violations for a fixed set of consecutive tests. This created the opportunity to verify that our initial testing budget was reasonable. We ran 300 tests on each scenario with all fields excluded, and found no new failures. This indicates that our initial test campaigns were sufficient for testing the systems.

6.4 50/50 random hybrid strategy

We plot the results for all three of the systems under test in figs. 6.4 to 6.6. The main observation is that, in all systems, the 50/50 random strategy outperforms either base method in terms of effectiveness. Note how, in the Ardu system, the random strategy illustrates the tradeoff between efficiency and effectiveness – because the fully fuzz strategy is generally less efficient, the random strategy efficiency is less than the dictionary efficiency. However, the fuzz strategy finds a relevant field that the dictionary strategy does not, so using 50/50 random leads to a higher effectiveness. In general, this result confirms our hypothesis that even a naive hybrid strategy can outperform either standalone test method.

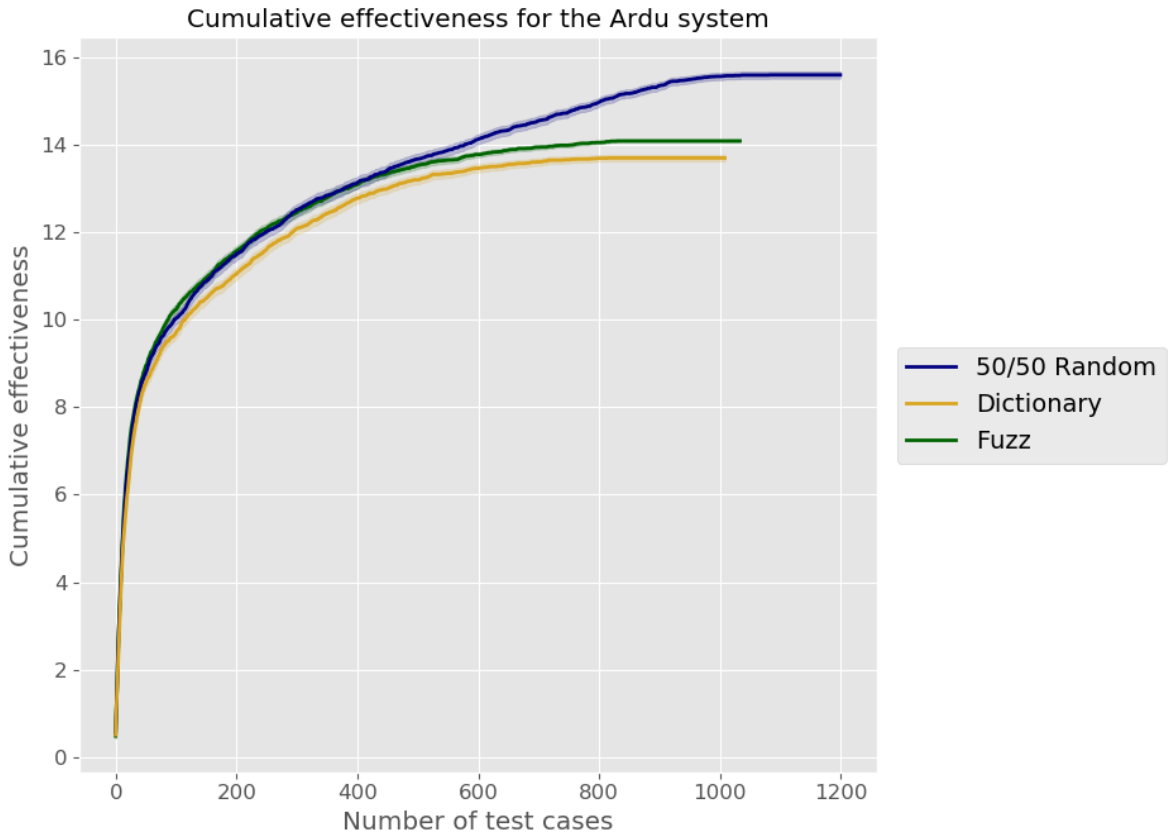

Figure 6.4: Cumulative graph for Ardu with 50/50 random hybrid strategy

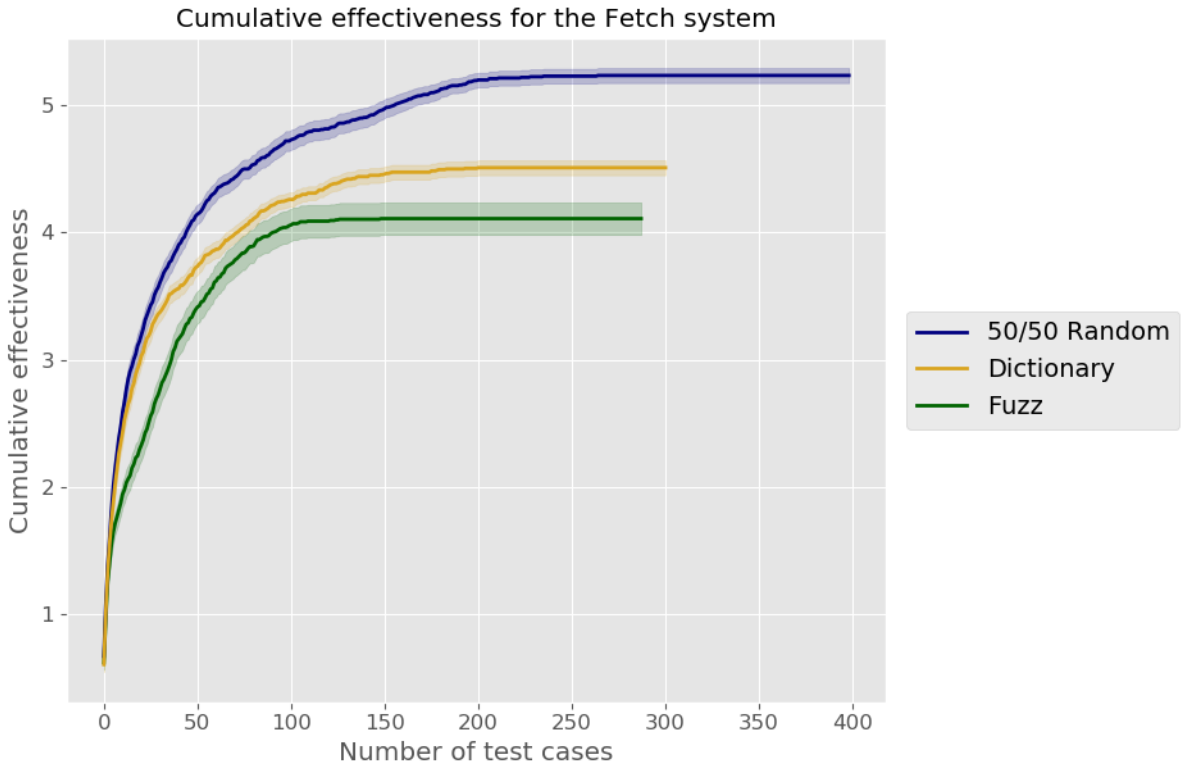

Figure 6.5: Cumulative graph for Fetch with 50/50 random hybrid strategy

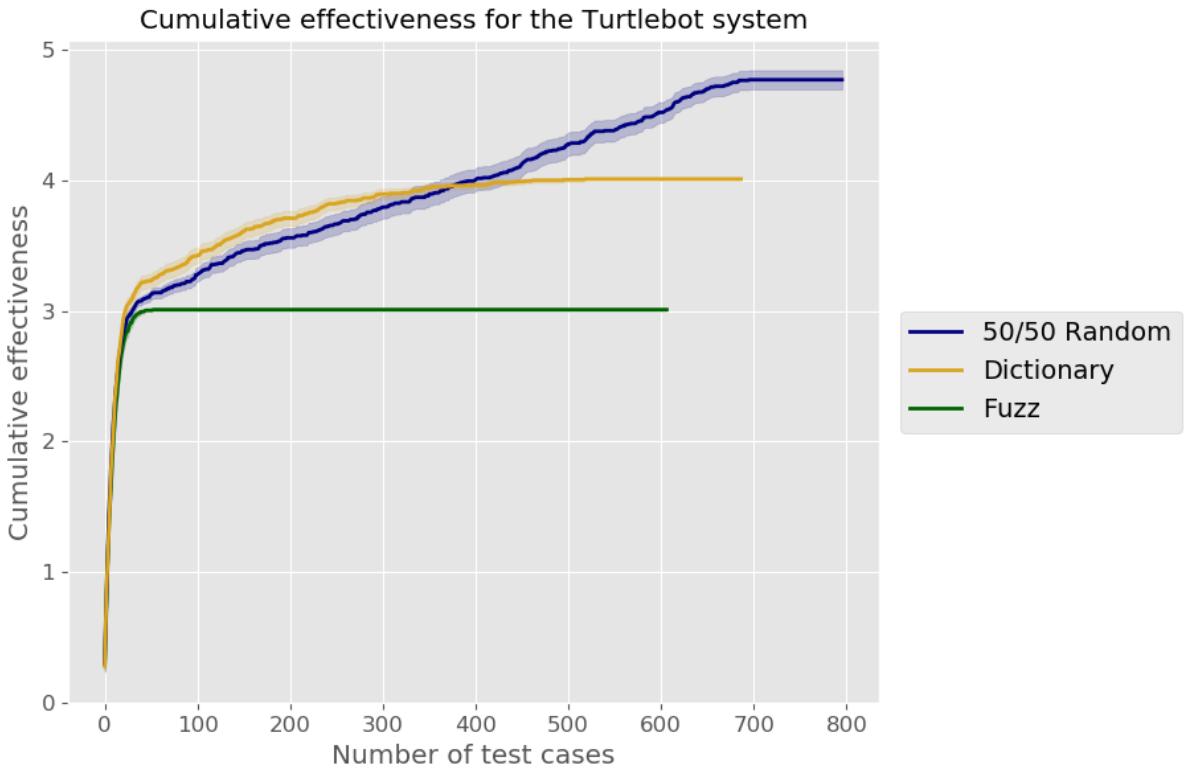

Figure 6.6: Cumulative graph for Turtlebot with 50/50 random hybrid strategy

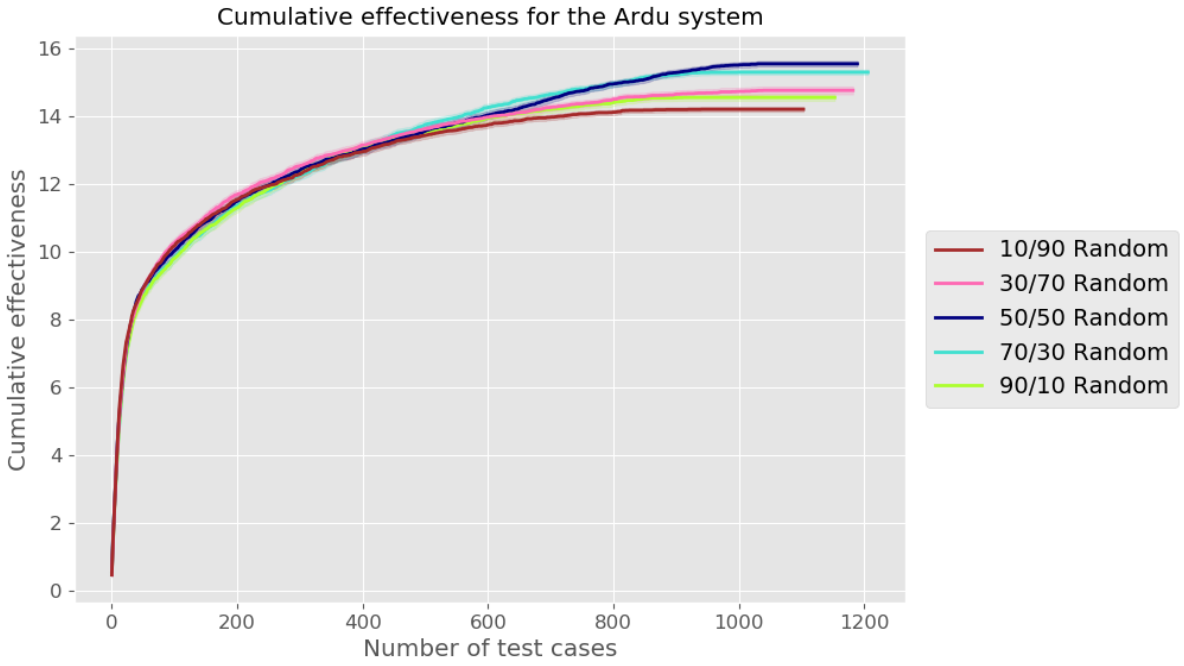

Figure 6.7: Weighted random strategy for Ardu (the first number is the percentage probability of selecting a dictionary test case, that is, 30/70 means a 30% chance of dictionary vs. a 70% chance of fuzzing)

6.5 Weighted random hybrid strategy

We plot the results for question 3 in figs. 6.7 to 6.9. In all cases, the 50/50 random strategy performs better than or within the error bounds of the best random strategy for the system under test, and is therefore used as a baseline for comparison below. The main takeaway is that any naive random hybrid strategy is beneficial, but the test methods perform well enough individually that tuned weighting does not largely matter.

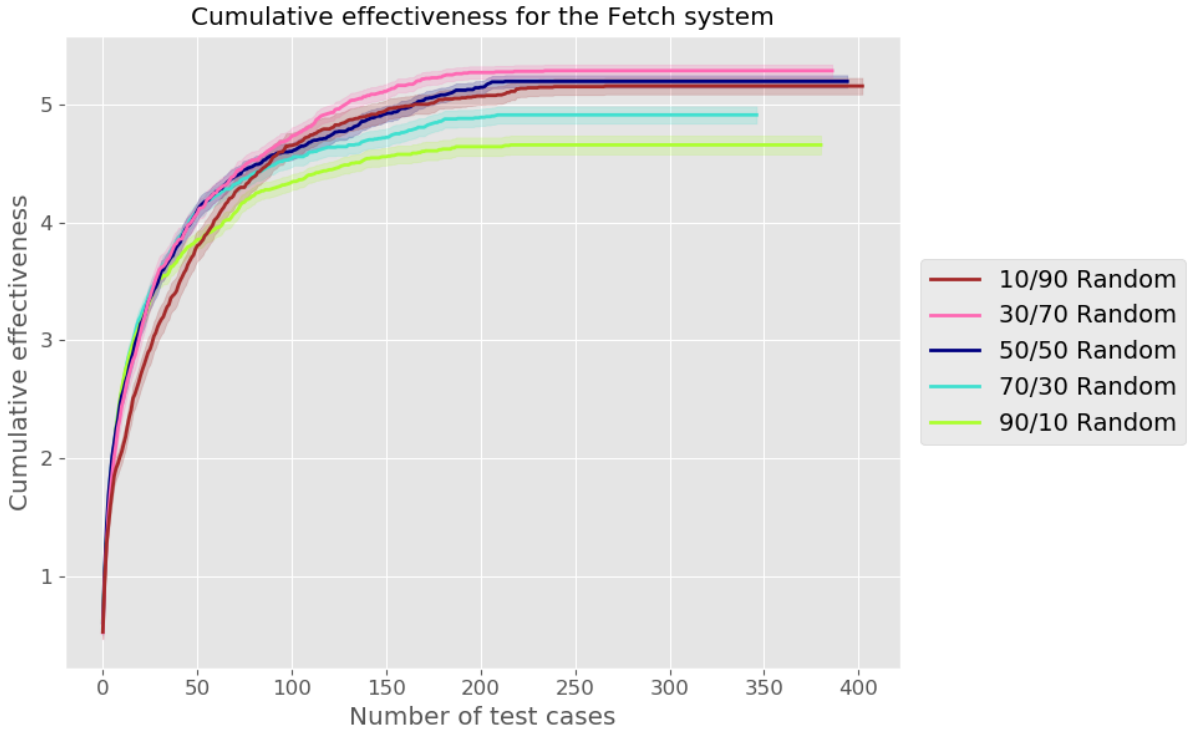

Figure 6.8: Weighted random strategy for Fetch.

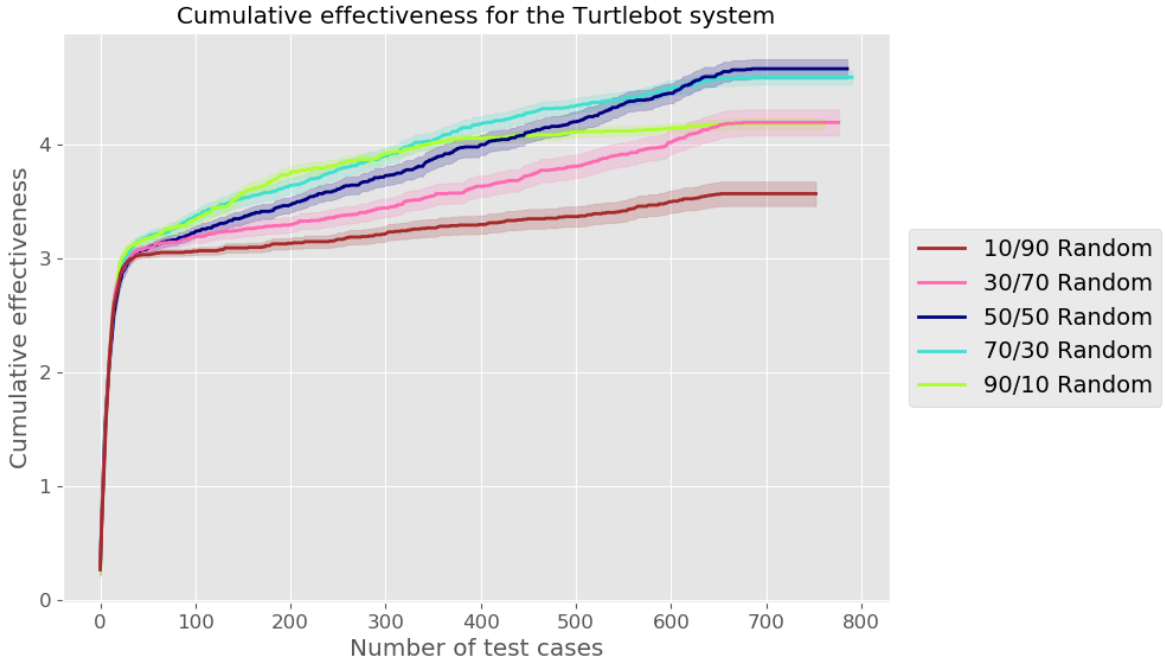

Figure 6.9: Weighted random strategy for Turtlebot

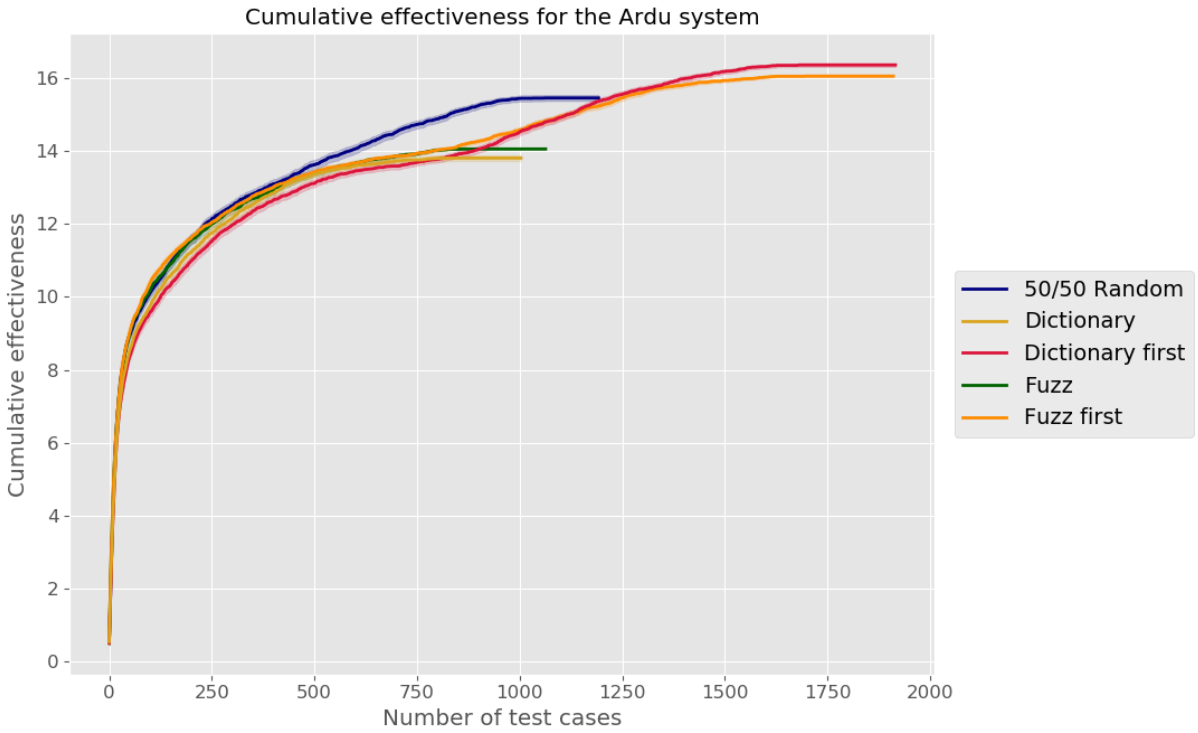

Figure 6.10: Comparison of fuzz-first and dictionary-first strategies for Ardu

6.6 Fuzz-first and dictionary-first strategies

We plot the results for question 4 in figs. 6.10 to 6.12.

Note that for all the systems, the first part of the fuzz-first strategy curve matches the fuzz-only strategy line, and the first part of the dictionary-first strategy line matches the dictionary-only strategy curve. This is consistent with the behavior of the strategies.

For all systems, the dictionary-first strategy outperforms the 50/50 random strategy and the fuzz-first strategy. We can conclude that dictionary-based testing systematically eliminates failures-triggering fields for each test and reduces masking effects faster. In theory, a 50/50 random strategy, if given enough time and a large enough threshold for consecutive non-failure-triggering test cases, would eventually reach the same maximum effectiveness, but on average it is less efficient.

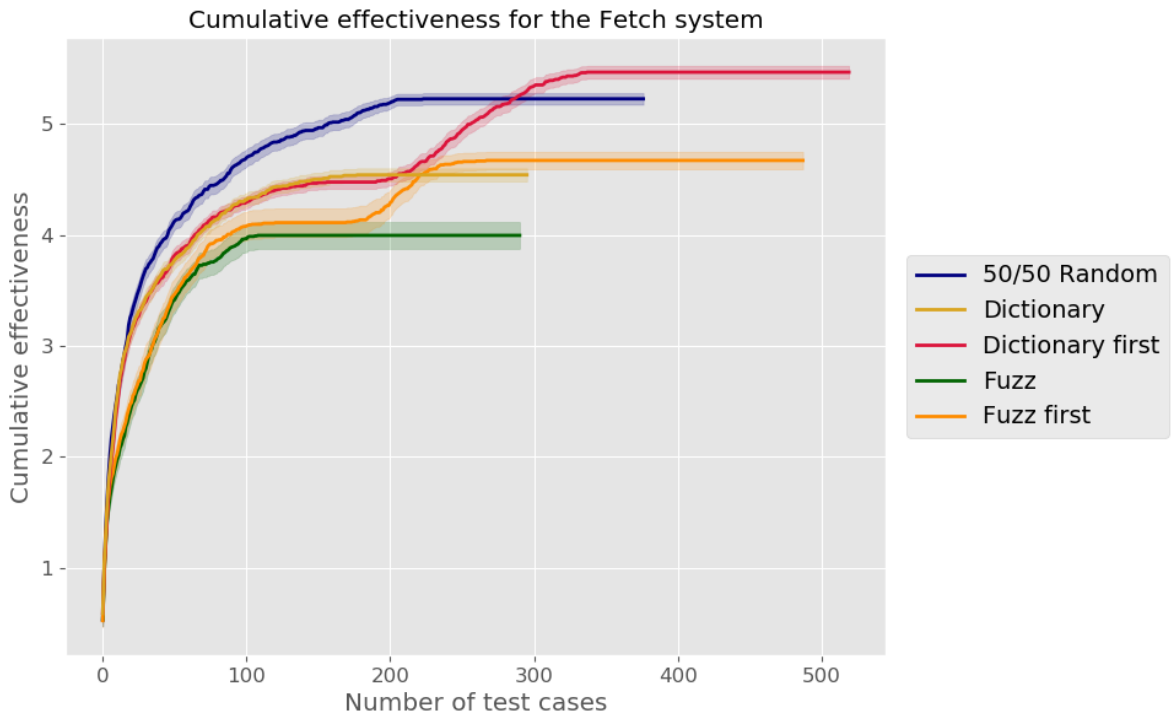

Figure 6.11: Comparison of fuzz-first and dictionary-first strategies for Fetch

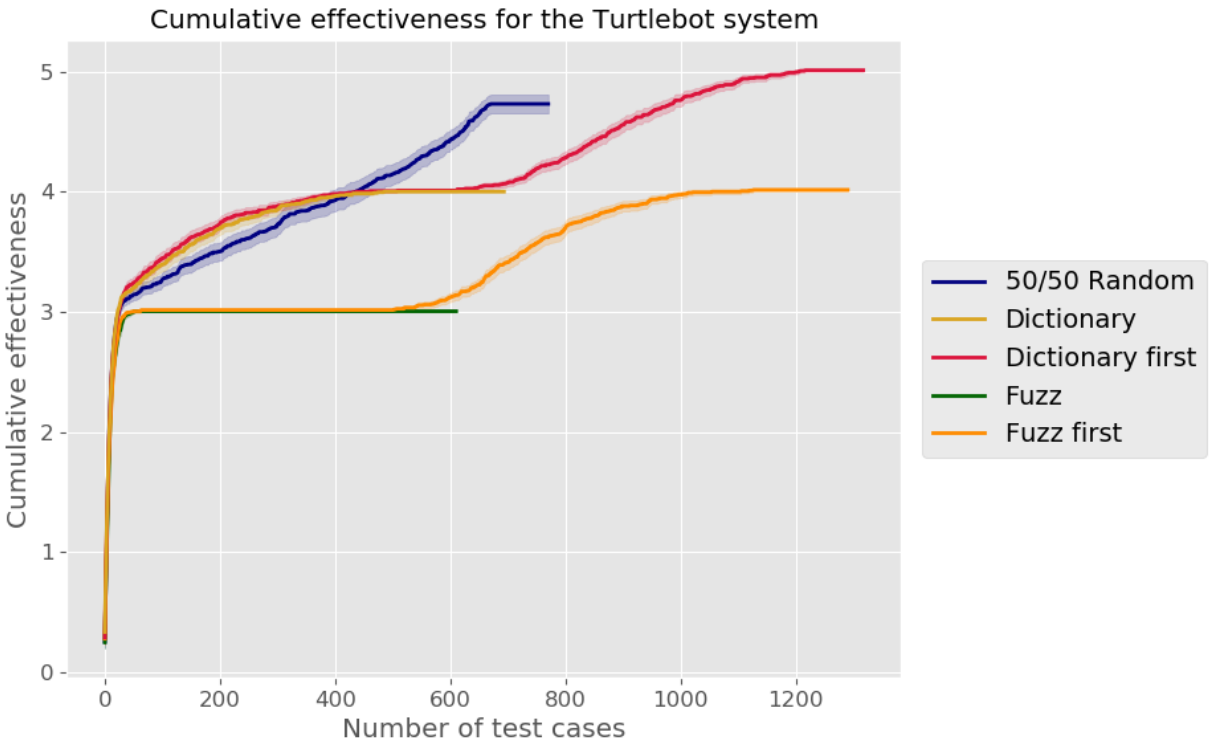

Figure 6.12: Comparison of fuzz-first and dictionary-first strategies for Turtle

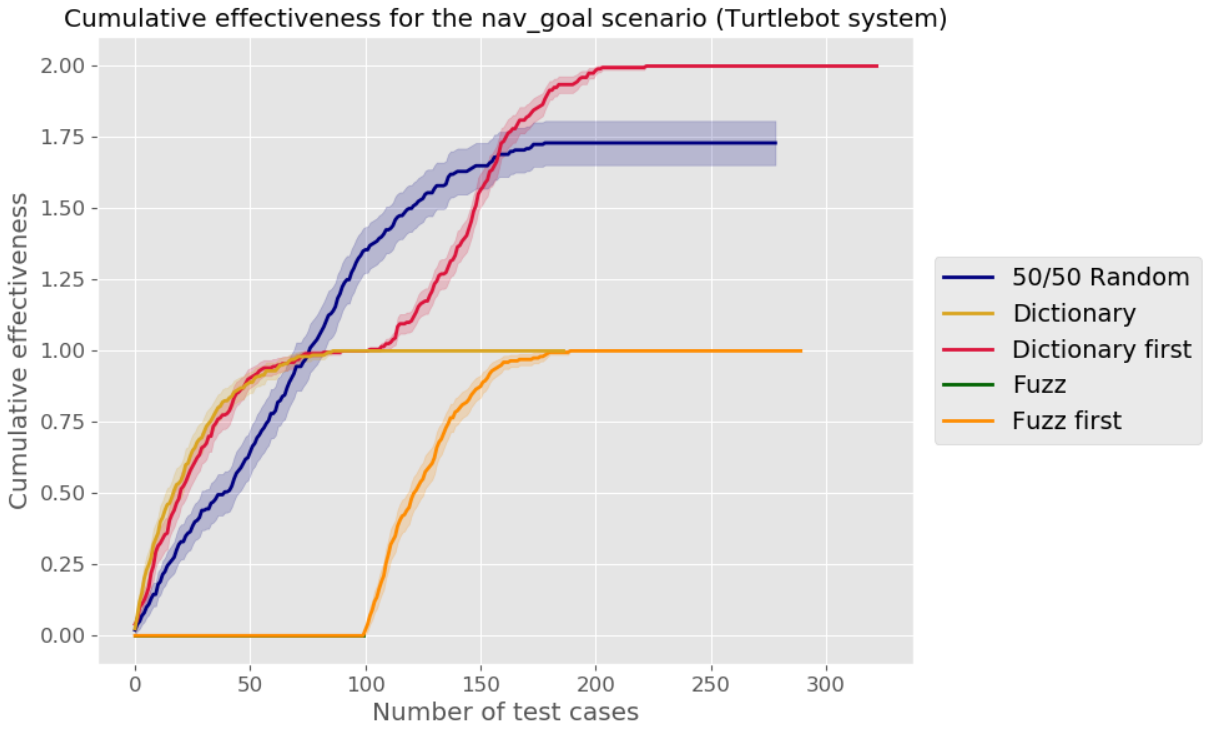

Figure 6.13: Comparison of dictionary-first and fuzz-first strategies for the nav_goal scenario (Turtlebot system)

6.6.1 Why dictionary-first is better

To answer why the dictionary-first strategy performs the best in all systems, we examine the performance of the strategies by use case scenario. We begin by displaying all of the graphs for the Turtlebot system as an illustrative example, and then only provide the relevant graphs for Ardu and Fetch. The remainder of the per-component graphs can be found in Appendix B. In most of these scenarios, all of the strategies performed about the same. In each of the cases of fence_mission, fence_vel, and modes (Ardu system), either the dictionary or fuzz method had zero exclusiveness with respect to the other, and hence the other method (and its corresponding "-first" strategy) performed better.

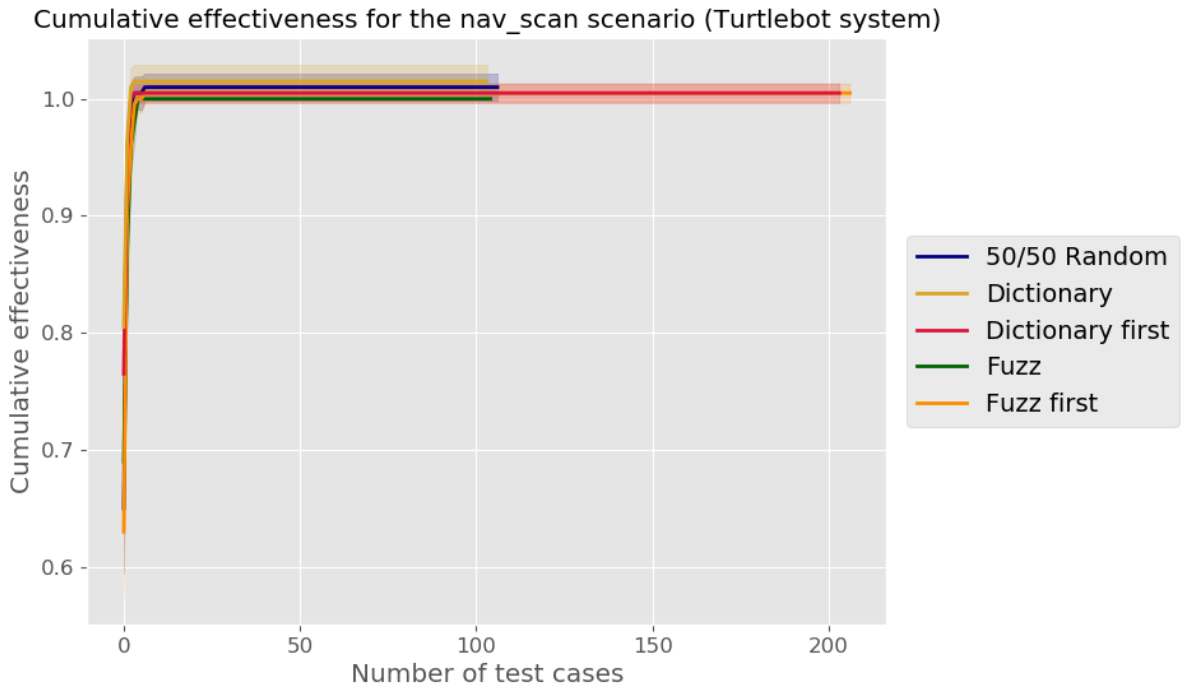

Figure 6.14: Comparison of dictionary-first and fuzz-first strategies for the nav_scan scenario (Turtlebot system)

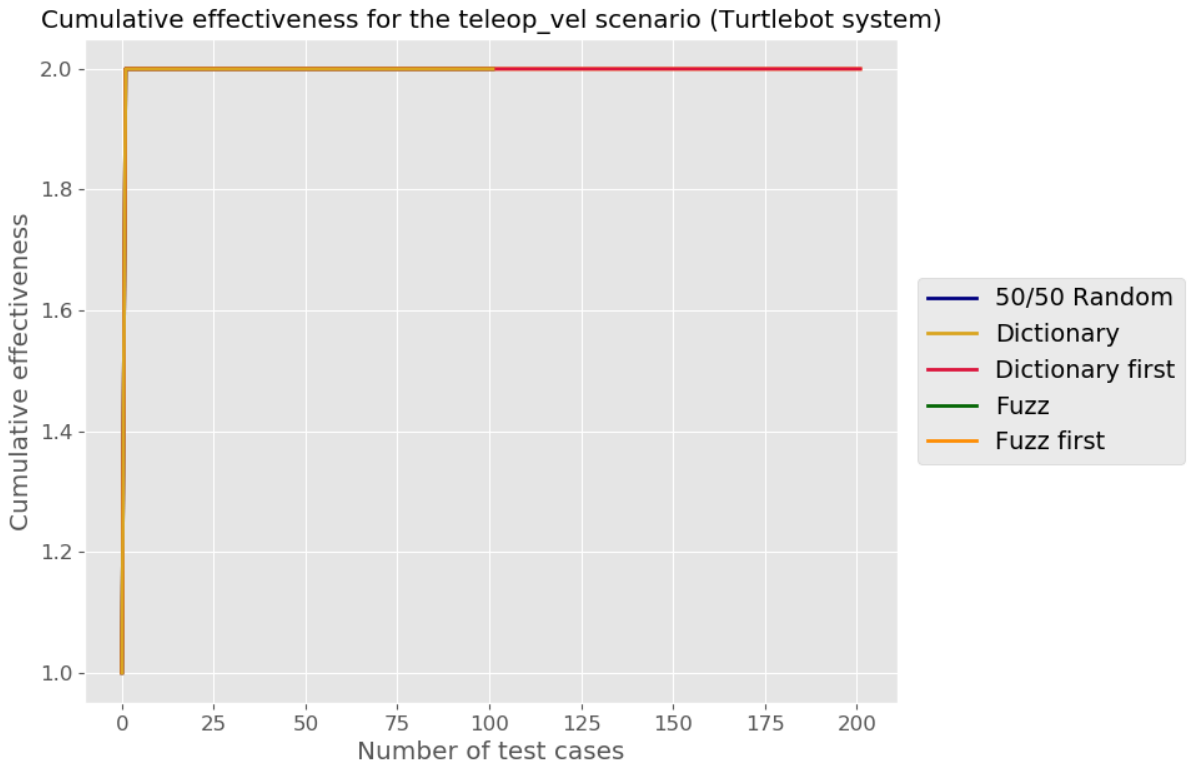

Figure 6.15: Comparison of dictionary-first and fuzz-first strategies for the teleop_vel scenario (Turtlebot system)

6.6.1.1 Dictionary-first in Turtlebot

For Turtlebot, the scenarios in which we found invariant violations are nav_goal, nav_scan, and teleop_vel. We plot the individual cumulative effectiveness graphs in figs. 6.13 to 6.15. We see that the nav_goal scenario is responsible for the differences in performances of the strategies.

Upon further examination, we see that the `covariance` field only triggers a failure in dictionary-based testing. The `position` field triggers a failure in fuzz-based testing, but only when `covariance` is excluded. Fuzzing is able to find some failures in extremely fragile fields, but needs `covariance` to be excluded to find failures in other fields. This likely means that `covariance` is a field that is robust to fuzzing and will ignore faulty inputs, which contributes to masking.

We verified that the system ignores faulty inputs by replaying the test inputs and observing the system in simulation. First, we note that the `covariance` field appears in every entry of a large array within the nominal input, and so a 20% nominal input replacement percentage ensures that multiple instances of this field are very likely to be perturbed within every test case. We indeed verified that, for every test case generated, at least one `covariance` field was perturbed using a test value. The nominal behavior of the system is to receive a navigation goal and begin to move to that goal. In the cases where dictionary-based testing causes the system to exhibit a core dump, the robot did start moving in simulation before the node (and thus the simulation) crashed. In all fuzz cases that we sampled for replay, the robot did not move at all. This is evidence that the system was indeed rejecting these inputs.

6.6.1.2 Dictionary-first in Ardu

In Ardu, the setpoint_raw scenario contributes to the dictionary-first strategy performing best, as illustrated in fig. 6.16. In this case, the `type_mask` field found only by dictionary masked the discovery of the `yaw_rate` field found by fuzzing.

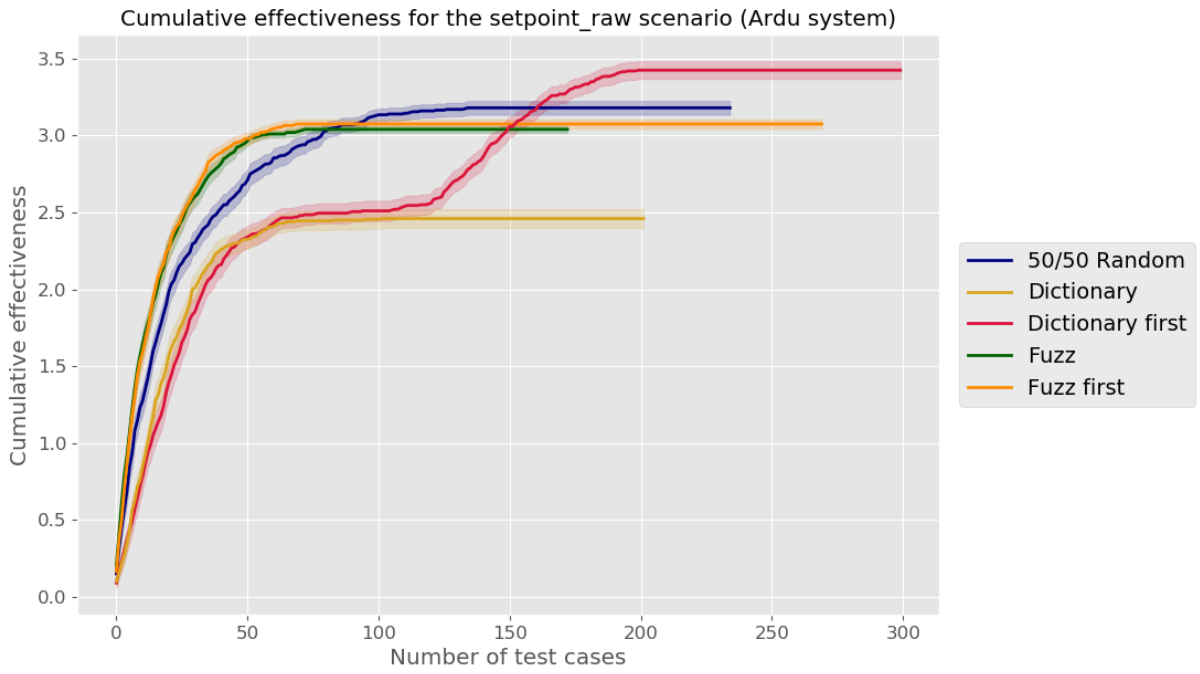

Figure 6.16: Comparison of dictionary-first and fuzz-first strategies for the setpoint_raw scenario (Ardu system)

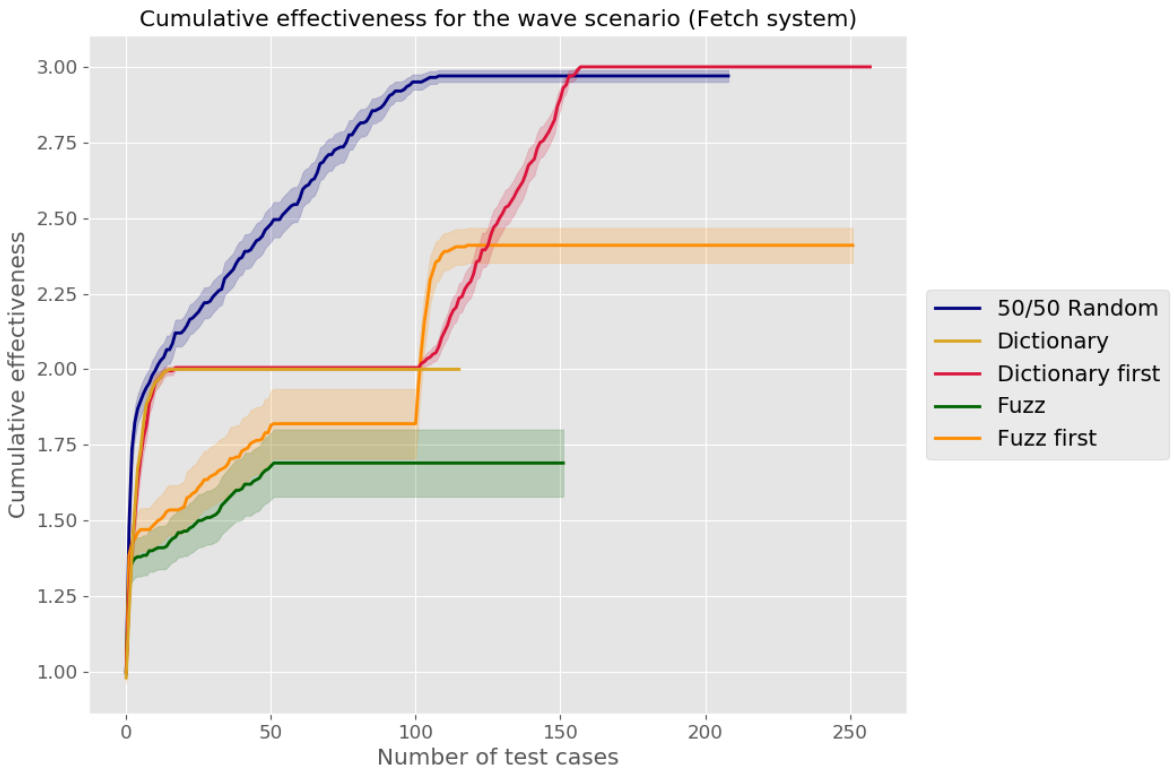

Figure 6.17: Comparison of dictionary-first and fuzz-first strategies for the wave scenario (Fetch system)

6.6.1.3 Dictionary-first in Fetch

Finally, in Fetch, the scenario in which the dictionary-first strategy performs the best is wave. This is plotted in fig. 6.17. Here, we find that the `operation` field, which only triggers a failure in dictionary-based testing, is required to be excluded for fuzzing to find failures in other fields. Similarly to Turtlebot, this likely means that `operation` is a field that causes the system to ignore most inputs if data assumptions on the field are not met. However, when `operation` is excluded from testing, fuzzing can uncover failures in other fields.

6.6.2 Dictionary-first takeaway

In all three of the systems we tested, we encountered scenarios in which the dictionary-first strategy performed better than any other strategy. In all of these cases, we discovered specific fields that had data assumptions that rejected fuzzed inputs and did not allow the system to be exercised in a deep way. However, these fields were vulnerable to dictionary-based testing, and once found using the dictionary technique, could be eliminated to find other bugs using fuzzing.

6.7 Other methods

The results of Section 6.3 as well as the high input set efficiency of fuzzing as demonstrated in Section 5.3 suggest that there might be some benefit to a strategy that fuzzes briefly first to sweep any very fragile fields, and then switches to a dictionary strategy followed by a fuzz strategy. Another strategy is to use weights at both stages of testing. For example, select dictionary-based test cases with a 90% probability versus fuzz, and, when no more invariant violations are found, switch to a scheme where fuzz test cases are selected with 90% probability versus dictionary.

We attempt this and plot an example in fig. 6.18. Note that in this figure, "90% D - 10% F" corresponds to the strategy just described, where dictionary test cases are selected with a probability of 90%. The strategy then switches to 90% bias towards fuzz test cases. "80% D - 20% F" follows the same convention, but with an initial weight of 80% towards fuzz. Ardu and Turtlebot, not pictured, had similar results as Fetch. We find that biasing towards fuzzing

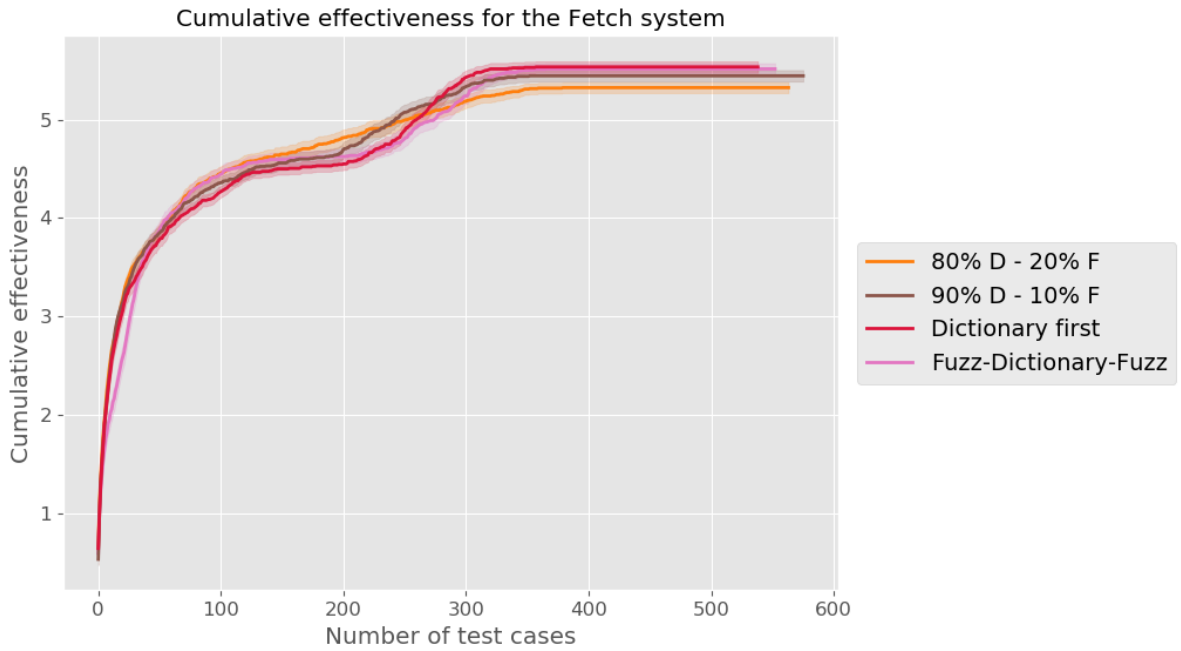

Figure 6.18: Comparison of additional hybrid methods on the Fetch system

in the first step, whether by performing fuzz initially or by weighting a small amount of test cases towards fuzzing, does not help. This is likely because dictionary-based testing, while less efficient for some inputs, can still find them.

6.8 Discussion

By comparing weighted random strategies against fuzz-first and dictionary-first strategies, we found that the dictionary-first strategy performs better than the 50/50 random and fuzz-first strategy in all systems. By performing deeper analysis, we found that this is because dictionary-based testing is able to find two specific fields in these systems that are masked by fuzz testing. We have shown how our modeling and analysis approach can lead us to this insight. We have observed this phenomenon in two separate use case scenarios in three separate systems, and suspect that this sort of masking will occur in other autonomy systems. We conclude that it is advantageous to use the dictionary-first strategy when testing.

Chapter 7

Additional test input generation methods

In previous chapters, we examined only dictionary-based testing and fuzzing as test input generation methods. While we showed that a dictionary-first hybrid approach yields the best results according to our metrics, we did not explore any other modifications to the test methods themselves. In this chapter, we show that our metrics can be applied to evaluate several other methods, including ones that were motivated by results in Chapters 5 and 6.

7.1 Guiding Questions

To outline this chapter, we ask several questions to guide our exploration:

1. How does testing efficiency change when the nominal input replacement percentage is changed, for both fuzzing and dictionary-based testing?
2. Does a smaller dictionary yield higher efficiency?
3. Does mutating the fields of a given nominal input result in a better testing efficiency/effectiveness when compared to replacing the field with a fuzzed or dictionary value?
4. Can mutating the values in the dictionary, as an attempt to systematically expand the dictionary, improve efficiency or effectiveness of the dictionary approach? Can we use these results to improve the dictionary?

5. Are there gains in efficiency/effectiveness when dictionary values take into consideration the semantics of a field (for example, discrete-value fields such as enumerator-style values or bitmasks)?

Question 1 is explored in Section 7.2. Section 7.3 answers question 2. Questions 3-5 are explored in Sections 7.4 to 7.6, respectively.

7.2 Nominal Input Replacement Percentage

As described in Chapter 3, the default replacement percentage used in our tool is 20%. The results presented thus far use this percentage. This percentage was chosen because it was historically effective at triggering failures in systems tested. Here, in order to explore how to optimize the test methods we use in our work, we measure how changing the replacement percentage affects the efficiency and effectiveness of each method. Intuitively, a replacement percentage that is too small will have a low probability of perturbing a relevant field or combination of relevant fields, while a replacement percentage that is too large might be rejected by the robot for being malformed and thus will not test any deep functionality. However, for a very fragile system, a high replacement percentage may more efficiently detect failures that allow the tester to eliminate fragile fields from testing, which would facilitate finding deeper bugs.

For scenarios that we found failures using dictionary or fuzz testing in, we ran exceptional dictionary and fuzz experiments with replacement percentages ranging from 10 to 100 in increments of 10. As a representative example, fig. 7.1 shows the effect of replacement percentage when using dictionary-based testing on the Ardu system. The remainder of the results are in Appendix C.

The main takeaway of these results is that efficiency does indeed vary by replacement percentage, and that several patterns jump out. Notably:

- Some test scenarios are "fragile," that is, for any replacement percentage, the testing efficiency will be very high. The next step in testing these scenarios should be to diagnose for

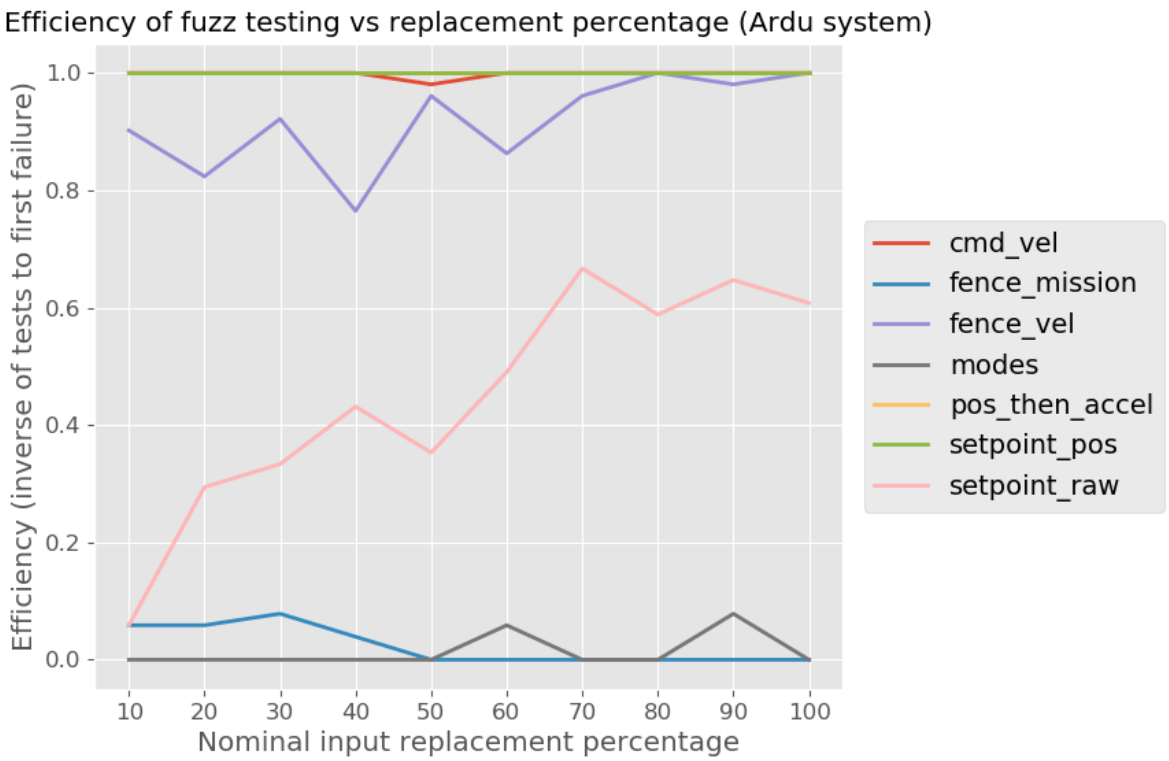

Figure 7.1: Efficiency comparison for replacement percentage using dictionary-based testing on the Ardu system.

the failure-triggering fields and discard those fields from subsequent testing to find deeper bugs.

- Some test scenarios can be eventually crashed using any replacement percentage, but show a clear peak in efficiency for a certain percentage.
- Some test scenarios need a high input replacement percentage to crash. We refer to these scenarios as "stubborn." In these cases, replacement percentage affects the effectiveness of a scenario, because a low replacement percentage may not discover any bugs at all.

These results suggest that replacement percentage can be changed during testing to find more bugs. In particular, if a test method has stopped finding bugs using one replacement percentage, it may be beneficial to increase the percentage. Detailed exploration of this is left to future work. Furthermore, we mention cases below where a higher nominal input replacement percentage can be used in conjunction with the other input generation methods, such as mutation.

7.3 Smaller dictionary size

Results in Section 5.3 suggested that a large dictionary may decrease the efficiency of testing. To see if dictionary efficiency could be improved, we reduced the dictionary entries for each field type to a very small set (about 10% of the original size) of suspected edge case values. We plot the results in fig. 7.2.

In some cases, this new approach outperformed the default dictionary, but the benefit is not consistent across systems and scenarios. This suggests that testing may benefit from a tiered dictionary approach, where the size of the dictionary is progressively expanded as testing goes on, but this is left to future work.

We hypothesized that a smaller dictionary combined with a higher nominal input replacement percentage may have a better efficiency and effectiveness, because the nominal input values are somewhat analogous to the well-formed values we removed from the dictionary. In the modes scenario for Ardu, we found that a small dictionary combined with a nominal input replacement

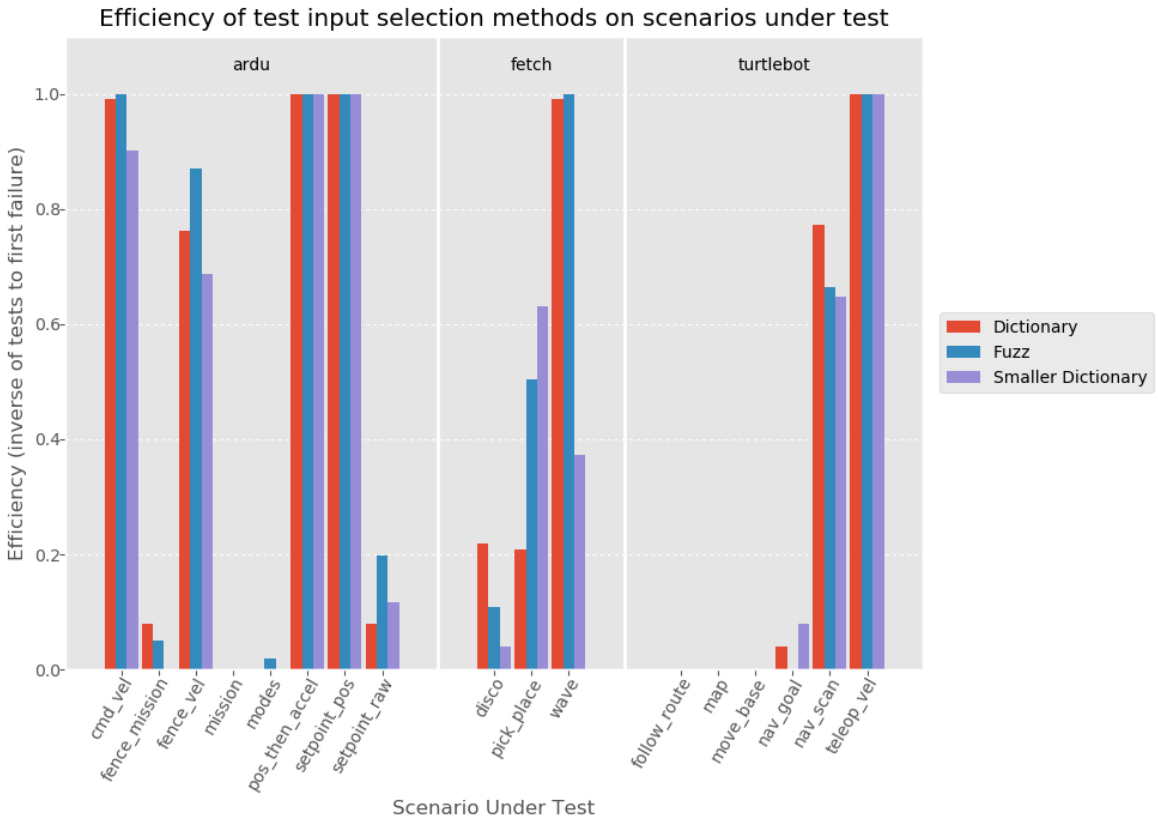

Figure 7.2: Efficiency comparison for fuzz, dictionary-based, and smaller dictionary.

percentage of 80% was able to trigger a failure using a field that was previously only found using fuzzing. While true that this does not improve effectiveness for the system when used in a hybrid technique that includes fuzzing, it is promising when compared to a basic dictionary-only technique.

7.4 Nominal input mutation

This section, as well as Sections 7.5 and 7.6, deal with methods that did not consistently show the same results for all systems. Instead, we present specific cases where these approaches managed to succeed at uncovering failures, and explain why this was the case.

We applied nominal input mutation, as described in Section 3.3.3 to all the systems under test. In summary, this involved adding or multiplying the original values by small numbers. We saw interesting results in two different cases:

- In the Ardu system, mutation violated the speed limit violation in several scenarios when applied to the `velocity` field. Recall from Section 5.2.2 that neither fuzzing nor dictionary triggered speed limit violations in this system. Interestingly, the `velocity` field was the one contributing to the high software crash efficiency for fuzzing and dictionary-based testing on this system. Therefore, we suspect that mutation changes the values by a small enough amount to only trigger the speed limit violation rather than a core dump, while dictionary and fuzzing are more heavy-handed.
- In the Fetch system, both nominal input mutation and dictionary mutation (discussed in Section 7.5) found failures in the `operation` field. While dictionary-based testing was able to find this failure eventually when enough fields were iteratively eliminated from testing, the mutational approach found it more efficiently without the need for iterative elimination. However, this may again be due to a masking effect – because mutation only found failures on this field and not on other fields, it behaved essentially as if the other fields had been excluded from testing already.

- In the Turtlebot system, mutation combined with an 80% nominal input replacement (that is, mutating 80% of nominal fields) found a **previously undiscovered** (by dictionary or fuzzing) set of failure-triggering fields: `range_max`, `range_min`, and an entry in the `ranges` array. The names of these fields indicate that they are semantically tied, and it may be the case that nominal input mutation is able to better exploit data assumptions because it only changes the values by a small but significant amount rather than replacing them.

7.4.1 Mutating strings

Because arithmetic mutations cannot be applied to strings, we also tested a special string mutation. This method did not improve the efficiency or effectiveness of the testing approach. For all of the systems we tested, only one string field (`group_name` in Fetch) was found to trigger failures, and it was sensitive to many fuzz and dictionary inputs.

7.5 Mutating dictionary values

Mutating the dictionary attempts to broaden the dictionary in a systematic way. When we initially ran these tests, we found that the efficiency of this method was comparable to the original dictionary-based testing. However, we suspected that this was because the mutations often duplicated values in the dictionary (for example, 1 added to 0 yields 1, which is already in the dictionary). We then changed the dictionary mutation to only create new values. That is, if mutation resulted in a value already in the dictionary, our testing tool kept mutating until a novel value was created.

One of the intended benefits of this method is to generate dictionary values that might have been left out of the dictionary. In practice, we found that this method actually performs the best for fields that Section 5.3.5 classifies as "categorical" fields – because this method expands the dictionary, it starts behaving more like fuzzing.

The specific cases where this method performed well were:

```
uint8 MAV_MODE_PREFLIGHT=0
uint8 MAV_MODE_STABILIZE_DISARMED=80
uint8 MAV_MODE_STABILIZE_ARMED=208
uint8 MAV_MODE_MANUAL_DISARMED=64
uint8 MAV_MODE_MANUAL_ARMED=192
uint8 MAV_MODE_GUIDED_DISARMED=88
uint8 MAV_MODE_GUIDED_ARMED=216
uint8 MAV_MODE_AUTO_DISARMED=92
uint8 MAV_MODE_AUTO_ARMED=220
uint8 MAV_MODE_TEST_DISARMED=66
uint8 MAV_MODE_TEST_ARMED=194
uint8 base_mode
string custom_mode
```

Figure 7.3: Definition of the mavinterface/MISetMode message

- In the fence_mission scenario on the Ardu system, this method triggered a failure in the `waypoints` field. This was a field that neither dictionary nor fuzzing were able to generate failure-triggering values for.
- In the disco scenario on Fetch, this method had higher efficiency than dictionary testing in finding "categorical" fields. It also found the `operation` field before dictionary did, similar to nominal input mutation.

7.6 Semantically-specialized dictionaries

Upon examining the nominal input bags, we noticed several opportunities for defining special dictionary entries. We describe them here.

7.6.1 Discrete values

This category encompasses enumerator-type and bitmask-type fields. The notable thing feature of fields is that they are often defined in a Robot Operating System (ROS) message description, which is an interface that is externally visible to the tester. Special values are defined as constants in the message itself. For example, in the Ardu message *mavinterface/MISetMode*, several special values are defined, such as `MAV_MODE_AUTO_ARMED=220` (see fig. 7.3). These values often seem arbitrary and might not exist in the exceptional value dictionary. We tested

the benefits of automatically detecting these values (which can be automated with a script by parsing the message definition for an "=" character) and using them for the dictionary entries for their corresponding fields. For a field in Ardu called coordinate_frame, we found that this specialized method had almost double the efficiency for triggering the failure than a basic dictionary approach. Even though effectiveness was not changed because both methods were able to trigger this bug, the increased efficiency indicates that this approach can be lucrative for this type of field.

7.6.2 Robotics Physics Values

In our examination of the nominal input bags, we determined several fields that represent values such as angles and rotational quaternions. We found a testing efficiency of near 1 when special quaternion values were used to test the `pose.orientation.z` field in some messages used by the Ardu system, while testing on angle-type fields on all systems yielded no gains in efficiency or effectiveness. This semantically-specialized approach is more time-consuming for the tester to set up, because it requires identifying special fields using manual inspection and coming up with special values, but may be worthwhile in a "long tail" debugging endeavor.

7.6.3 Strings

We observe that strings in the nominal bags represented things such as physical robot component names ("`shoulder_lift_joint`") and mode-determining variables ("`camera_depth_frame`"). We hypothesized that adding these values to the dictionary would cause the robots to act in modes that they were not expecting. We tested whether adding these values to the dictionary could result in testing improvements, but we found no gains in efficiency or effectiveness using this method. The only string-type relevant fields that we uncovered in our testing required simple values such as the empty string to trigger a failure, and did not need anything complicated.

7.7 Exploitation versus Exploration in a hybrid method

In this chapter, we have shown several specific cases where a more advanced testing approach may have been beneficial to use. However, we did not find consistent enough results to recommend an advanced approach over our hybridized dictionary and fuzzing strategy. We do have enough anecdotes that, even if a test method has not been fruitful for the systems we tested, it may prove beneficial in given use case scenario in a new system. In this case, it may be beneficial for the testing strategy to change in real time to reflect this. The general problem of dealing with unknown rewards is known as exploitation versus exploration [109], which has been studied in the context of reinforcement learning. Particularly, we reference a k-armed, otherwise known as multi-armed, bandit [23]. This problem is described using the metaphor of a slot machine player who can choose any of k arms and is trying to maximize their reward. The player has a model of the reward probability for each of the arms, and the model for a given arm may be improved by pulling the lever more times and observing the reward. The player has to balance "exploitation," or pulling the arm they believe to be optimal, with "exploration," or pulling an arm in order to gain more information about its reward probability.

Concretely, in the case of devising an optimal testing strategy, each arm represents a different choice of test method for a test case, and the reward is a potential invariant violation in that test case. Furthermore, because we eliminate fields from testing (as motivated in Section 6.3), we are discussing a "non-stationary" or "restless" bandit problem [71], where the reward probabilities change over time (i.e. when a failure is found, we eliminate the relevant fields from testing, and so the testing result probabilities change). At each step of this online testing strategy, we would like to *exploit* the method that would yield the most invariant violations, but we do not know which method this is. We can build up a model using *exploration*, by choosing a method and seeing if it yields results. We also may have a prior model of the reward functions, for example, based on results of other use case scenarios in the system.

7.7.1 ϵ-greedy approach

A common approach to the k-armed bandit problem is the ϵ-greedy algorithm [73], where the lever that is believed to be optimal is chosen with a probability $1 - \epsilon$ for a fixed ϵ, and at random from the other levers with a probability ϵ. Usually ϵ is chosen to be small, for example 0.1, such that exploitation is favored over exploration. While this method does not expressly address the non-stationary problem, given enough use case scenarios, this strategy can build up a model for how well the test methods work. Some initial exploration in applying this approach for our work yielded fruitful results that outperformed the 50/50 random strategy described in Section 6.4. However, because the number of combinations of replacement percentages, input generation methods, and eliminated fields is so vast, modeling this strategy comprehensively was infeasible for this dissertation. Application of the ϵ-greedy algorithm in selecting test cases that may involve advanced input selection techniques is recommended as promising future work.

7.7.2 More approaches

Other approaches include variations of ϵ-based greedy methods [73], algorithms based on UCB (upper confidence bound) [73], and Thompson sampling [14]. Woo et al. have applied various k-armed bandit policies to bit-flip mutational fuzzing, and have found good result with weighted random, round robin, and EXP3.S.1 strategies [111]. As research into k-armed bandits, and in particular restless bandits, advances, testers will be able to choose an advanced exploration vs. exploitation strategy to effectively and efficiently test systems in real time. The general observations we have made in this chapter about various test methods and their variations can be used as prior beliefs for any of these strategies.

7.8 Conclusions

In this chapter, we presented several ways to tweak the basic testing methods from Chapter 5 and explored whether these tweaks resulted in gains in testing efficiency and effectiveness.

The benefits of the approaches were not consistent across systems, and sometimes involved educated guesses at effective combinations (such as mutation combined with high nominal input replacement percentage). Nonetheless, we found several examples where these approaches may be promising on future systems. We concluded with a discussion on exploitation versus exploration problems, which encompasses the challenge of applying advanced strategies when their payoffs may be unknown.

Chapter 8

Lessons Learned and Recommendations

In Chapters 5 to 7, we presented empirical analysis of different test input generation methods and test case selection schemes. The recommendations that came out of this analysis pertain specifically to input parameter generation, but, along the way, we have discovered some general recommendations and lessons learned about autonomy software. This chapter summarizes these findings, in several categories:

1. Testing autonomy systems
2. Writing testable autonomy systems
3. Writing safe autonomy systems.

8.1 General recommendations for testing autonomy systems

One of the goals of this dissertation is to show how past testing experiences can concretely inform future directions. For example, if overflow bugs are common for one type of system, testing tools should be expanded to exploit overflow vulnerabilities for future systems of that type. Starting with the Automated Stress Testing for Autonomy Architectures (ASTAA) project at the National Robotics Engineering Center (NREC), we have had the opportunity to learn from many autonomy system bugs, as well as from the process of designing a testing framework for auton-

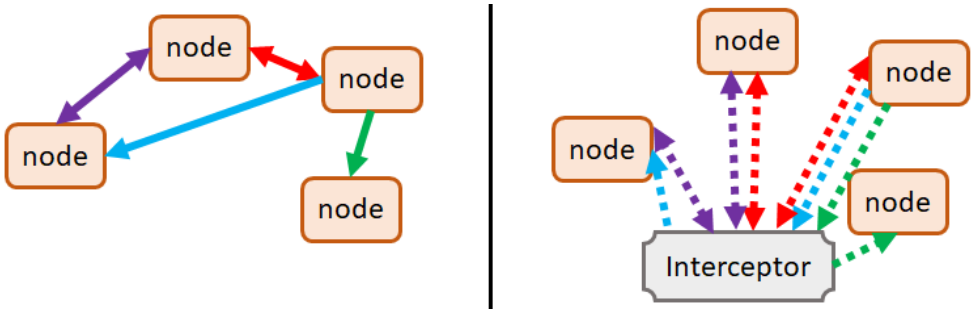

Figure 8.1: Channel rerouting instrumentation needed for a true interceptor

omy systems. Apart from input selection, we have several recommendations for architecting a testing project, as a whole.

- Interception testing vs. log replay: the original ASTAA project featured interception testing, which allowed for high-fidelity preservation of system state and timing requirements. However, simulating the network topology in order to allow for a man-in-the-middle in every communication channel creates a large overhead to create the testing infrastructure. For Robot Operating System (ROS) systems, to simulate communication between nodes, we would need to instrument our own star topology of ROS nodes, re-routing every channel we intercept on. Figure 8.1 illustrates this. The left side is an abstraction of the system as-is, with each colored arrow representing a different channel. The right side shows the same channels renamed and routed through an interceptor node. In this dissertation, we have shown that perturbing and replaying nominal input bags finds a non-trivial set of bugs, with less complexity. Interception would have allowed us to inject messages into control loops, as subsequent inputs to a control loop change based on behavior (e.g. laser scan data that causes the robot to change its laser sensor position, which affects what laser scan data is seen next). Instead, we had to be careful to find interfaces with minimal feedback, such as command velocity and goal points. Because we were able to find non-trivial results with log replay, we believe that it is practical to use log replay as a first step, before moving on to interception if time and budget allow.

- Extracting interfaces: a test is only non-trivial if the interface it tests is meaningful. In this dissertation, we relied on online documentation of open source autonomy systems to extract interfaces. We found that a basic knowledge of robotics is helpful: at first glance, a ROS node diagram may be confusing, and it may not be apparent which ROS topics are relevant inputs for testing. We have devised a few tricks in testing interfaces. For example, the teleoperation interface to a robot is usually not so interesting, as, in our experience, it is pretty robust to unexpected characters/keyboard commands. However, this interface abstracts the command velocity interface to a robot, and by recording the command messages output by the teleoperation node, we can gather nominal data to command the robot. Furthermore, some robots are commanded by interfaces that are not easily interceptible (i.e. ROS services), and it may be necessary to write a small amount of middleware to be able to access these interfaces using automated robustness testing and test the robots in a meaningful way.
- Decreasing testing infrastructure bottlenecks: as mentioned in Section 8.1.1, simulation is very costly. To test autonomy systems efficiently, it helps to test the smallest component possible. The Robustness Inside Out Testing (RIOT) [64] project focuses on testing single nodes and abstracting to the external controls, using robustness testing techniques. We suspect that the hybrid approach outlined in this dissertation can also be beneficial for testing at the component level.
- Measuring dimensionality: while we found that most bugs have one or two relevant fields, with possibly a handful of startup messages, it is difficult to exploit low dimensionality in testing. If the assumption is that most failures are triggered by just one parameter in a traditional software API call, 1-wise coverage of the input dictionary suffices [52]. However, in a robotics system, an input may have a hundred or more messages with many fields per message. We indirectly exploit low dimensionality by using a comparatively low nominal input replacement percentage of 20%. As far as we can tell, random perturbation of a

nominal input bag is an effective way to exploit autonomy systems, but using a hypothesis of low-dimensionality bugs to improve testing is left as future work.

8.1.1 Actual testing cost and optimizations

Until this point, we have discussed efficiency and effectiveness in terms of the number of tests, and not the time it takes to run a test. Furthermore, in the sections above, we have not incorporated the cost of diagnosing the relevant fields needed to trigger a failure. We present our observations of the real time cost of testing systems and suggest some optimizations.

8.1.1.1 Time to test

In our experience, the cost of a single test case for an autonomy system is much, much greater than the cost of a single robustness test API system call. This is because a robot must be initialized to a testable state, in simulation. To ensure more accurate results, we run our simulations in real time. In practice, we have found that it takes about 60-300 seconds to run each test. This testing time cannot be easily reduced by providing a smaller input bag, because most of the overhead comes in setting up and tearing down the simulation and getting the robot into a testable state. Because testing a robot is so costly, optimizing for testing efficiency becomes important, since a test budget is usually decided based on time and not number of tests, and a longer average test case time means fewer total test cases in the budget.

8.1.1.2 Time to diagnose

Once a failure is found, it must be diagnosed. So far in this dissertation, we have not been considering the cost of diagnosis. This is because diagnosis must happen in order to triage the bug, so the cost incurred by diagnosis is unavoidable, as opposed to the cost incurred by a test case that yields no invariant violation. However, this cost is fairly substantial. For example, for every invariant violation, we verify that the result is reproducible by replaying the test input, which doubles the testing time for that test case. We then delta debug the input, which reduces a bag of 9-253 messages down to a small set of messages. At best, delta debugging runs $2 \times$

$\log_2(\text{input\_size})$ tests, which is a substantial amount of time, especially given how long a test case runs. After delta debugging, these minimal messages must be reduced to their relevant fields, which is also costly, on the order of reducing 5-34 fields to 1-2. Again, this algorithm runs $2 \times \log_2(\text{perturbed\_fields})$ tests, at best. The costly diagnosis time motivates parallelization of testing, which we discuss below.

8.1.1.3 Optimizations

As robotic simulation becomes more advanced, and as simulation time is accurately able to be run faster than real time, the cost of testing overhead will decrease. However, care must be taken to ensure that a sped-up simulation does not introduce inaccuracies into the test results. However, the scope of this dissertation is for real-time simulation, so we discuss other optimizations here.

- **Parallelization**: Parallelization can be highly beneficial for testing. Because our tests are independent up to the diagnosis step, doubling the number of computers used for testing doubles the number of test cases run. For advanced strategies in a k-armed bandit problem (as discussed in Section 7.7), parallelization of test cases can be used for parallel solutions [47]. Furthermore, instead of using a "number of tests without invariant violations" heuristic to decide when to switch testing to another use case scenario, scenarios can be tested in parallel.

  Parallelization gives even more gains in the diagnosis step. Delta debugging is a fairly parallelizable algorithm, in that at least two independent test cases are run at each step of the algorithm. While the number of concurrent test cases can vary at every step of the algorithm, optimizations can make use of the extra computers by removing chunks of the input bag at random. Because diagnosis is costly but necessary, having multiple machines to run this step can greatly improve the usage of the testing budget.

- **Nominal input size**: As stated above, nominal input size does not have a big effect on test time, but directly factors into the time it takes to diagnose a found bug. For this reason, it is beneficial for the nominal input size to be small. However, if a nominal input is too

small, it may not encompass all of the functionality of a use case scenario and may reduce test method effectiveness.

We experimentally verified if truncating the nominal input bag at intervals of 20% has an effect on testing. For some systems, truncating at even 80% did reduce the efficiency of the test campaign, or even caused scenarios to exhibit no failures under test. We suspect that this is because the system was not exercised enough to expose the functionality where the failures lie. We conclude that, even though it may be tempting to truncate a nominal input to reduce diagnosis time, it is important to verify that the truncation still allows for a full execution of the functionality under test.

- **Not relying on fixed delays for system startup**: Our initial system startup scripts included a delay (in the form of a `sleep` command) between every step of startup, based on observations for how long the systems took to start up. For example, between launching `roscore` (the ROS master node) and the simulator, we would include a delay of 5 seconds to allow `roscore` to initialize. In our experience, when we instead parsed for key phrases in the log files that indicated that the software component has been initialized, we noticed an improvement in both running time of the tests and the reproducibility of the test case result. While a timeout is a simple way to get a system running, in practice, it is fragile and not recommended for testing at-scale.
- **Diagnosis by inference**: Instead of finding a minimal input bag and set of relevant fields algorithmically, we ran a large set of test cases and then examined the intersections of the sets of perturbed fields. In some cases, we were able to hypothesize that a certain input field triggered a bug because it was perturbed in very many failure-triggering input bags. If the diagnosis for a single test case is particularly costly, or if many tests have already been run and need to be diagnosed, this is a feasible method for making an educated guess about the relevant fields to trigger a failure. However, this method is not precise and requires a bug that can be found with high efficiency to produce enough test cases to make an inference.

Also, as nominal input replacement percentage rises, this method becomes less effective, because the set of perturbed fields approaches the entire space of inputs.

As robotics simulation gets better, and as developers learn to write test-friendly code, the time required for testing and diagnosis might decrease. However, because robots are cyber-physical systems that require startup time and resources for simulation and logging, actual testing time will remain greater than for simple API call-based desktop software testing. Because of this, this dissertation uses tests to first failure as our metric for efficiency, and focuses on making recommendations about test case generation itself. Optimization of test case runtime and diagnosis is left to future work. At a given point in time for a given system, the duration of a test case will remain roughly the same regardless of test input selection method, but a campaign of tests can be made more efficient and effective by choosing the inputs in a strategic manner.

8.2 General Recommendations for having testable autonomy systems

Apart from testing autonomy systems efficiently and effectively, the developers of autonomy systems can take a few steps towards better testability for their robots. Most of the recommendations we make here are specific to the autonomy domain, or are at least particularly applicable to the autonomy domain versus traditional software domains.

- Good logging: system logs that are accurate, replayable, and parsable are invaluable in both detecting and diagnosing problems. For instance, invariant detection would not be possible without accurate and parsable system output. Furthermore, replayable system logs allow us to instrument tests by replaying system state, and to verify that found bugs are reproducible. This is especially important because autonomy system outputs exhibit significant non-determinism, and replayable logs allow testers and developers to determine what behavior is a side effect of this noise.
- Graceful failure: system nodes should not exhibit a silent failure, because other nodes de-

pend on their operation and can propagate a hang or crash failure through the entire system. A node failure should not create an unsafe state of the system: safety-critical nodes should have redundant safety shutoffs that stop the system gracefully. Watchdogs are one way to detect node failure [67], and if a watchdog outputs a message with a guaranteed frequency, it is easy to write an invariant to detect node failure. Nodes should also not malfunction silently – some sort of checking should be done on the data of the node, especially if the data is going to propagate through several nodes. This ties in to the importance of checking for data assumptions at multiple levels. For example, `NaN`s propagate [89], so checking float variables for `NaN` values should be done at all critical junctures. Detectable failures not only help keep the system safe, but are invaluable for testing: the faster a node failure or malfunction is caught, the easier automated failure detection is in terms of testing.

- Write safety invariants, or at the very least, safety requirements: in our previous work, we stated that several of the systems we tested did not have safety requirements and therefore could not be monitored for invariant violations. We showed that, for the systems that had safety requirements, the majority of issues found were invariant violations. For this dissertation, we had to deduce the safety requirements from documentation and write our own invariants. This means we might have missed several key safety properties and did not test for them. A clear, formal safety specification would have alleviated this issue.
- Expose interfaces at appropriate levels: robots are distributed systems that communicate via messages, so the interfaces to a robot are largely dictated by this architecture. However, in our testing experience, interfaces can be exposed better by providing a clear architecture diagram of message passing, and show how input messages to one node translate to output messages to another node.
- Provide nominal scenarios and examples: getting the robot to a testable state is a big bottleneck in setting up a testing infrastructure for a robot. By providing nominal scenarios that test all the message-passing interfaces of a robot, developers give testers the necessary

infrastructure by which to get the robot in a testable state and provide templates of these interfaces. Nominal scenarios should include completion of all the basic intended functionality of the robot. For example, if the robot has a path planner, a provided nominal scenario should include a way to provide the relevant inputs to the path planning nodes of the system.

- Ensure system portability: we were not able to test some systems because they required a specific system set-up that was not compatible with our testing framework. Installation scripts, in particular, quickly become of little use as outside package dependencies may change their configuration. Instead of providing the system as a special virtual machine, distribute the system as a package in the ROS ecosystem. If the system has other software package requirements, provide installation instructions rather than, or alongside with, a system-specific install script.

8.3 General recommendations for writing robust autonomy systems

One of the goals of robustness testing is to catch bugs and fix bugs in a controlled environment so that they do not manifest when the robot is operating in the real world. Testing a lot of autonomy systems and analyzing the results can also reveal patterns that persist across systems. These patterns can be a valuable lesson in common programming pitfalls that should be avoided in order to write more robust systems.

- Speed limits: we have found at least one bug that we believe is an improper test of speed limit. A simple comparison of the form $!(speed > limit)$ does not suffice, because $speed$ may be a `NaN` or may be negative, which results in a false negative in the comparison as written.
- Floats: Using floats in iterators is dangerous because floating point error can accumulate and produce a completely unexpected result. Furthermore, comparisons with `NaN` always

return false (except in the statement `NaN != NaN`), and so every inequality involving a floating point number should check for `NaN`.

- Mask values/enumerator values: Some bugs we found exploited discrete-valued fields (e.g. mode numbers) in the nominal inputs. It is important to verify all assumptions made, such as if the robot does not expect to be commanded to a non-zero velocity in a stop mode.
- Simple bounds checking: for each of the systems tested, we encountered a failure that was triggered by either very large or very small values. If values for physical computations are expected to fall within a reasonable range, this expectation should be coded into the software to avoid violations. Furthermore, be cautious with values signifying "replan attempts," as we found that large inputs caused robot nodes to crash, likely from running out of memory in computation.

Chapter 9

Conclusion

9.1 Summary

In this work, we addressed the challenge of test input generation and hybrid test case selection methods for autonomy systems. To address the challenges of autonomy systems needing to be safe and having large inputs without a centralized interface, we presented a test framework that uses nominal input mutation at the communication layer and checks for faults using an invariant monitor. We applied several test input generation techniques to three autonomy systems and showed how these techniques exhibit tradeoffs. We used metrics of comparison between test input generation techniques to show how to empirically evaluate test case selection strategies to make a recommendation for a more efficient and effective test campaign.

In the end, we found that no single test generation strategy is as good as a hybrid strategy. While even a simple 50/50 random strategy that chooses between fuzzing and dictionary-based testing outperforms either method, there are further gains in testing efficiency and effectiveness when we allow for strategies such as nominal input mutation and high nominal input replacement percentage, especially later in testing. Ultimately, the strategy that consistently outperformed the others for the systems tested was dictionary-first. This was due to fault masking: certain fields were robust to fuzzing and were only detected as failure-triggering by dictionary-based

testing. Once these fields were eliminated from testing, fuzzing was able to discover other failure-triggering fields.

9.2 Research contributions

This dissertation claims the following contributions, with their locations in the text noted:

9.2.1 An approach to metrics for comparing robustness testing techniques

We provide this in Chapter 4, Section 4.1 ("Metrics of comparison"), and Chapter 6, Section 6.3 ("Cumulative model"). We found that two main metrics capture the tension between seeking to find many bugs versus wanting those bugs to be unique. Our main metrics are **efficiency** (the reciprocal of the average number of test cases to first invariant violation) and **effectiveness** (the number of unique classes of a bug given a way to describe the bug, with a detailed discussion of these classes in Section 4.1.2), along with the complementary notions of **overlap and exclusiveness**

9.2.2 Robustness testing results for three open source autonomy systems using dictionary-based testing, fuzzing and certain variations

We provide this in Chapter 5, Section 5.2.1 ("Efficiency and use case scenario effectiveness"), Section 5.2.2 ("Invariant Effectiveness"), and Section 5.2.3 ("Diagnosis and field effectiveness"). The variations are explored in Chapter 7. We not only apply fuzzing and dictionary-based testing to the systems under test, but also explore nominal input mutation, changes to the dictionary (namely, mutating dictionary values, lowering the size of the dictionary, and using semantically-aware values), and replacement percentage. We find that both dictionary and fuzzing outperform the other method in multiple instances, when measured in terms of efficiency and component and field effectiveness. Among the variations to input generation techniques, we highlight several cases where testing may benefit from more fine-tuned approaches.

9.2.3 A hybrid testing technique for each of the three open source autonomy systems, shown to outperform each of the basic testing methods

We provide this in Chapter 6, Section 6.6 (“Fuzz-first and dictionary-first strategies”) and explore it further in Section 6.6.1 (“Why dictionary-first is better”). We find that even a simple 50/50 random strategy outperforms each of the basic testing methods in terms of efficiency and effectiveness. We show that even more complex strategies can provide further gains, by evaluating fuzz-first and dictionary-first strategies and showing that dictionary-first outperforms the other strategies for the systems tested. Using our metrics and an investigation into the failure-triggering inputs found by this strategy, we explain that this is due to a masking effect. That is, fields that are robust to fuzzing prevent the system from exercising the full system state, and finding and eliminating these fields using dictionary-based testing allows us to find deeper bugs.

9.2.4 A recommendation of heuristics for hybrid testing techniques and a list of lessons learned to inform testing autonomy systems in the future

We provide this in Chapter 6, Section 6.6.1 (“Why dictionary-first is better”) and touch upon more advanced techniques in Chapter 7, Section 7.7 (“Exploitation versus Exploration in a hybrid method”). Most of this contribution comes from the entirety of Chapter 8. We evaluated the benefits of our wining dictionary-first strategy empirically, but there were some recommendations for testing and writing autonomy systems that we arrived at anecdotally from years of experience working on these problems. These are summarized as lessons learned in Chapter 8. Taken as a whole, we hope that all of these recommendations can be used to more efficiently and effectively test autonomy systems, and inform how robotics software is written, leading to a safer future in autonomy software.

9.3 Future work

Future work in the input generation for autonomy systems space can build upon the work in this dissertation. Our work focused on testing at the system level, but a comparison of input techniques can be done at the unit level to compare what inputs can generalize to the system interface, as in the Robustness Inside Out Testing (RIOT) project [64]. Testing at the unit level can also allow for the introduction of white-box techniques, as branch coverage on an individual node is easier to achieve than coordinating branching over a distributed system of nodes.

While this dissertation focused on log replay rather than interception, applying the different test input generation techniques using an interceptor may pay off, by exploiting the control loops of an autonomy system.

It would also be interesting to incorporate repair into the process, rather than eliminating fields from testing. This would give a more accurate measure of the unique failures discovered in testing.

Some optimizations to testing may be possible, such as verifying if sped up simulation preserves the test results run in a real-time simulation, or more parallelism to diagnosis algorithms such as delta debugging. Furthermore, because we saw that most faults are triggered by a small number of relevant fields, it may be possible to apply combination testing techniques that are meant for large inputs and perform 1-wise or 2-wise parameter coverage.

Chapter 7 leaves several avenues for further exploration. Work on nominal input replacement percentage and autonomy-specialized dictionary entries showed that these approaches would benefit from more investigation. For example, using a higher replacement percentage after fragile fields have been eliminated from testing may reveal more failures, especially when combined with techniques such as dictionary entry mutation. The input generation itself may be improved with a deeper investigation into semantically-specific entries, or with a tiered dictionary approach that begins with a very small dictionary for efficiency and adds on more values if the testing budget allows. We also suspect that a hybrid model that takes a mixed approach within a single

test case, for example, dictionary values for ENUM-style fields and fuzzing for continuous value floats such as speed, would perform well. In addition, our examination was limited to two invariants (software crashes and speed limits), but showed that invariant-specific exploitation, such as NaNs and negative values for speed limits, is a promising space. Finally, the exploration versus exploitation problem, as discussed in Section 7.7, may be fruitful for determining if an advanced test input strategy should be used.

Appendices

Appendix A

Effectiveness tables

Table A.1: Fault-triggering fields per scenario, by test method, for the Ardu system

Scenario	Dictionary	Fuzzing	Both
cmd_vel			`twist.linear.*`, `twist.angular.*`
fence_mission	`value.real`		`value.integer`
fence_vel			`twist.linear.*`, `twist.angular.*`
modes		`altitude`	
pos_then_accel		`pose.orientation.*`	`pose.position.*`
setpoint_pos			`pose.orientation.*`, `pose.position.*`
setpoint_raw[a]	`type_mask`	`yaw`	`position.*`, `velocity.*`

[a] Each field in the setpoint_raw entries is implicitly paired with the `coordinate_frame` field, which appeared in every failure-triggering input

Table A.2: Fault-triggering fields per scenario, by test method, for the Fetch system

Scenario	Dictionary	Fuzzing	Both
disco	num_planning_attempts		max_velocity_scaling_factor
pick_place			goal.possible_grasps, goal.group_name
wave			num_planning_attempts, max_velocity_scaling_factor, replan_attempts

Table A.3: Fault-triggering fields per scenario, by test method, for the Turtlebot system

Scenario	Dictionary	Fuzzing	Both
nav_goal	pose.covariance		
nav_scan			time_increment
teleop_vel			twist.linear, twist.angular

Appendix B

Hybrid strategy performance by scenario

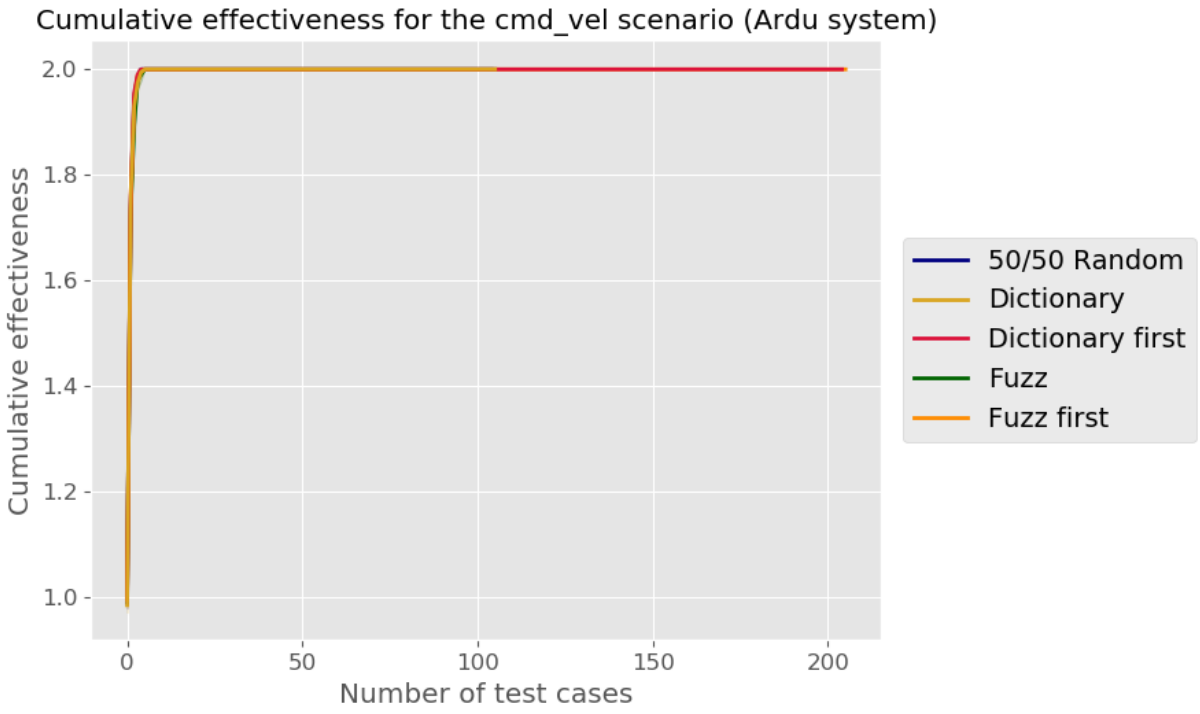

Figure B.1: Comparison of dictionary-first and fuzz-first strategies for the cmd_vel scenario (Ardu system)

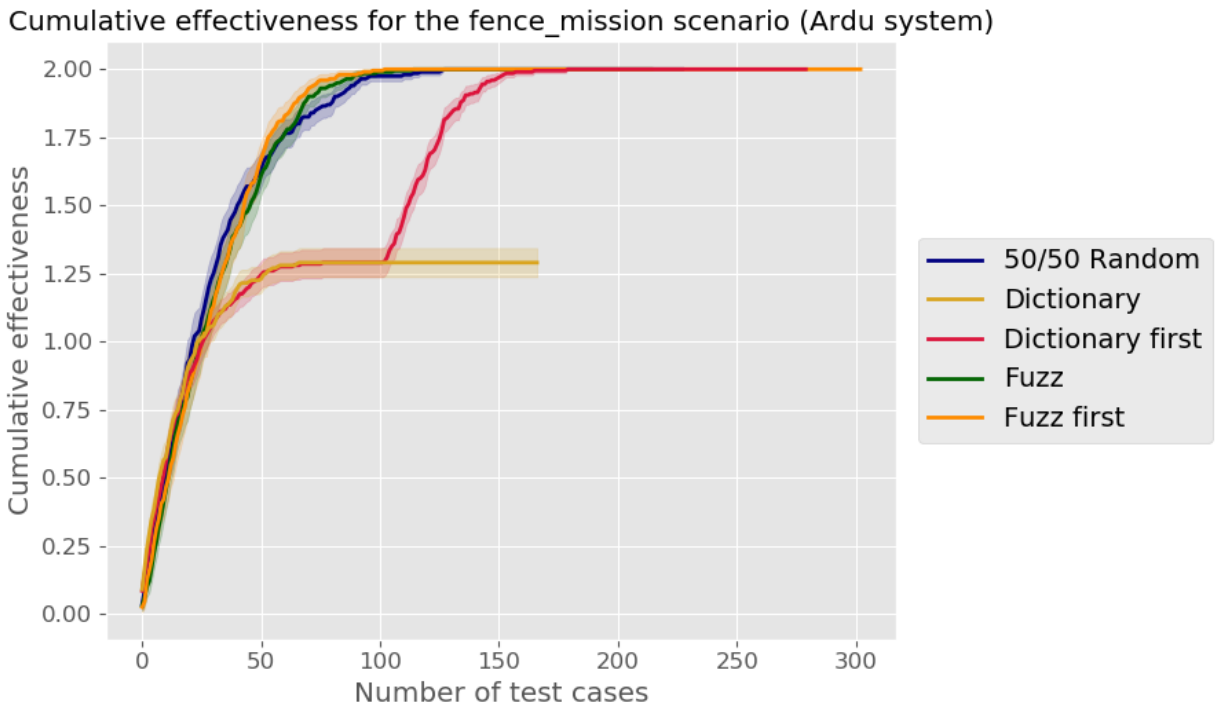

Figure B.2: Comparison of dictionary-first and fuzz-first strategies for the fence_mission scenario (Ardu system)

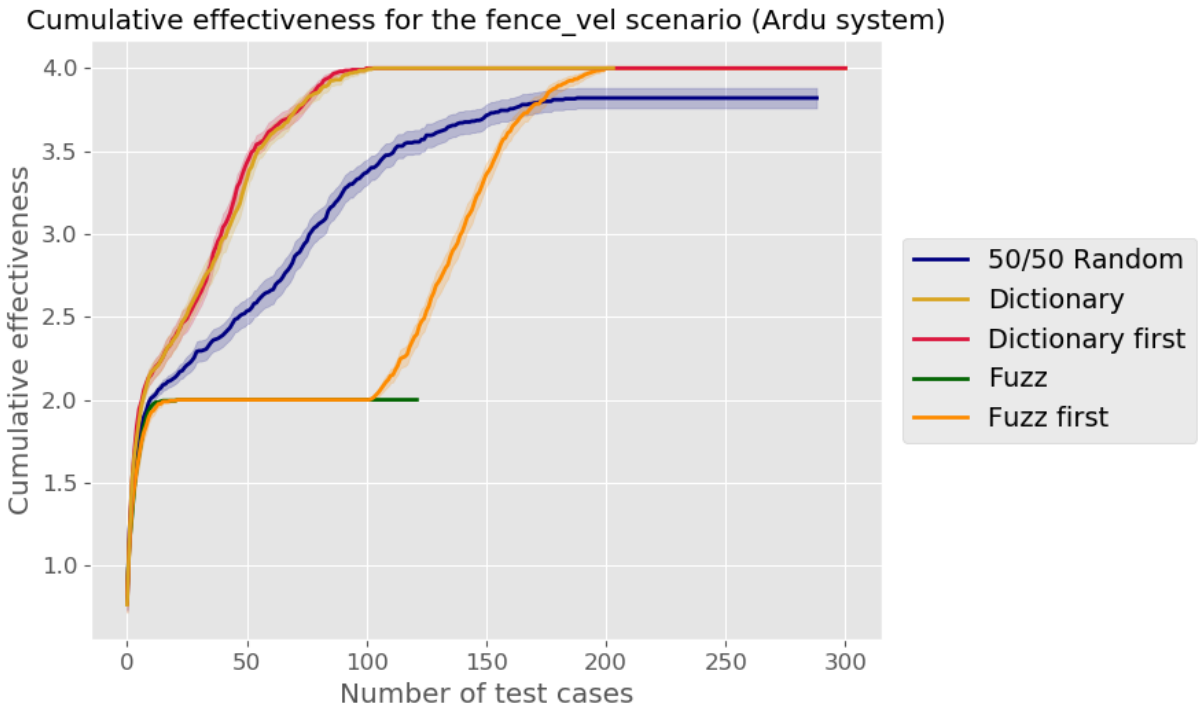

Figure B.3: Comparison of dictionary-first and fuzz-first strategies for the fence_vel scenario (Ardu system)

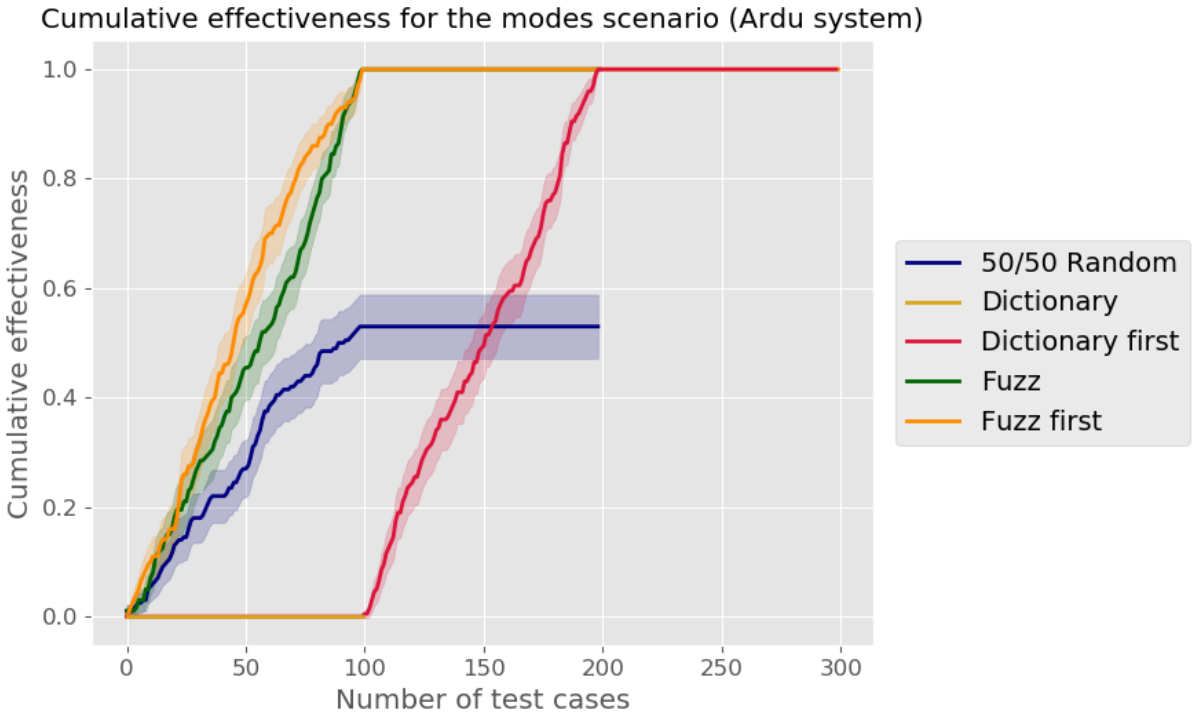

Figure B.4: Comparison of dictionary-first and fuzz-first strategies for the modes scenario (Ardu system)

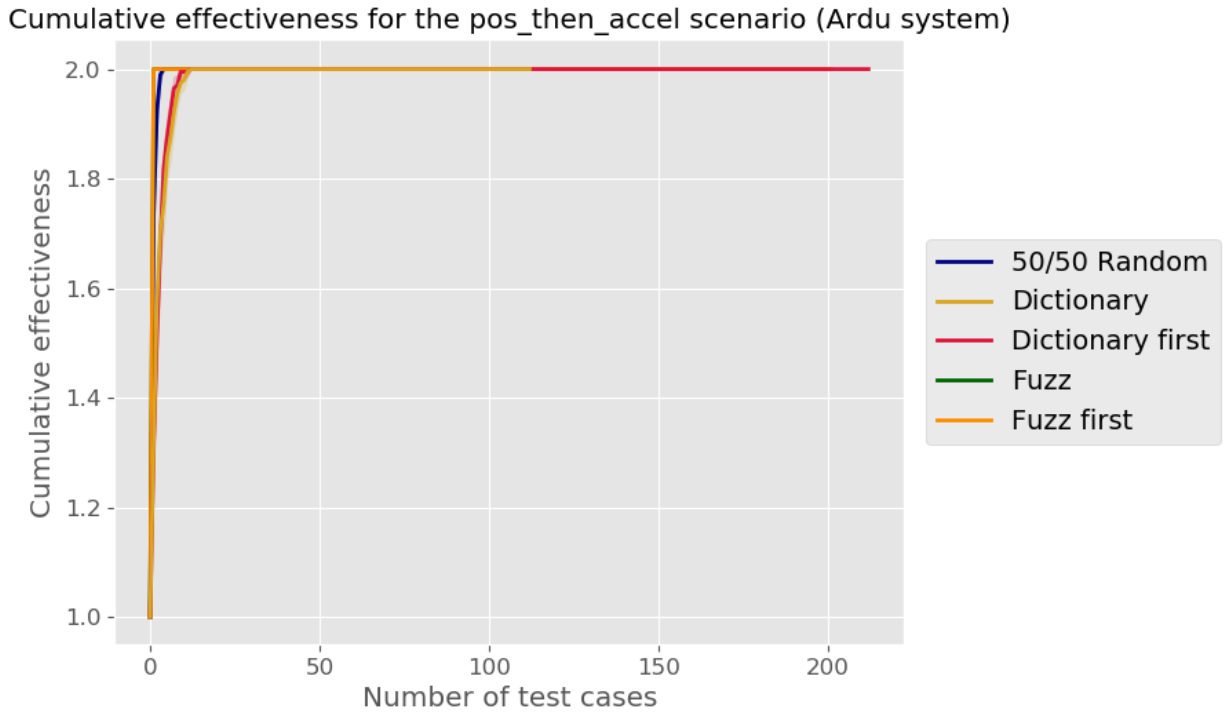

Figure B.5: Comparison of dictionary-first and fuzz-first strategies for the pos_then_accel scenario (Ardu system)

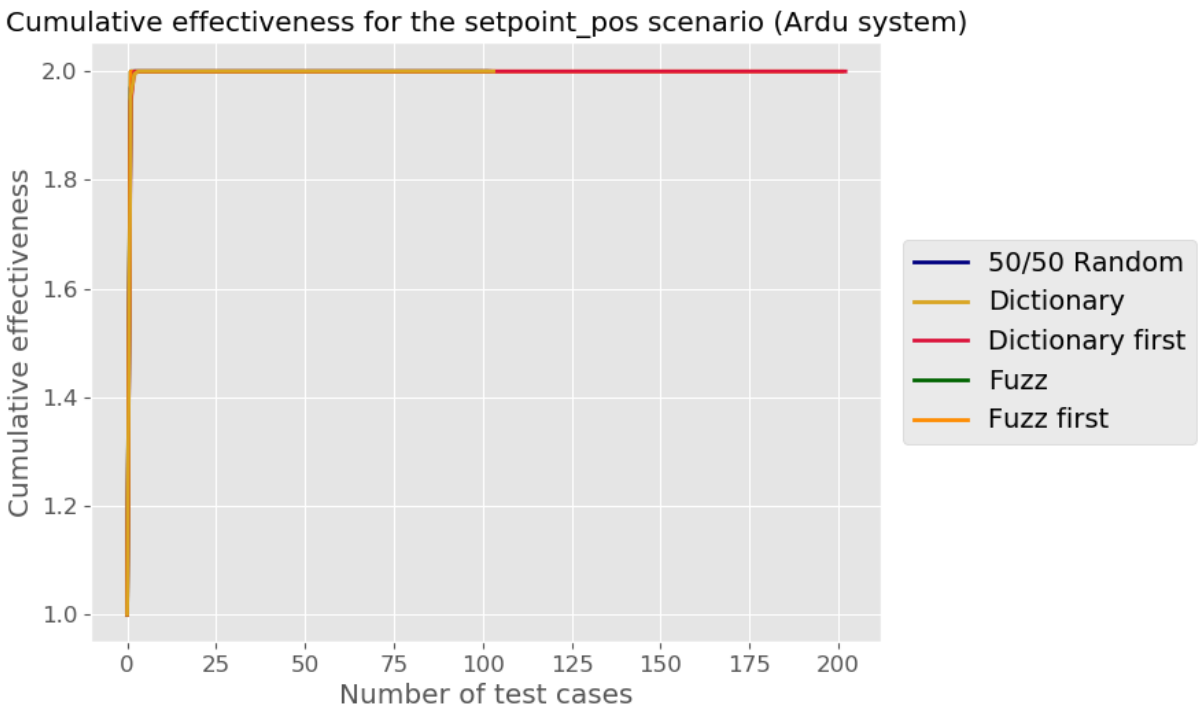

Figure B.6: Comparison of dictionary-first and fuzz-first strategies for the setpoint_pos scenario (Ardu system)

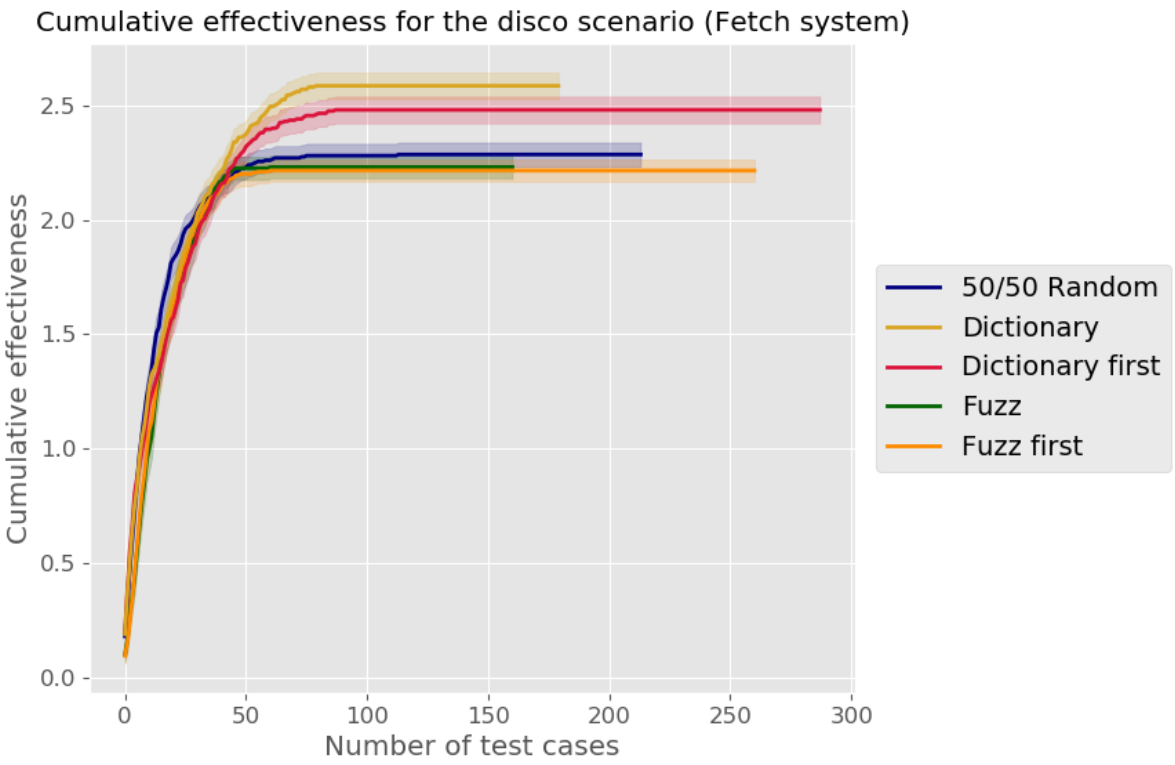

Figure B.7: Comparison of dictionary-first and fuzz-first strategies for the disco scenario (Fetch system)

Appendix C

Replacement percentage efficiency

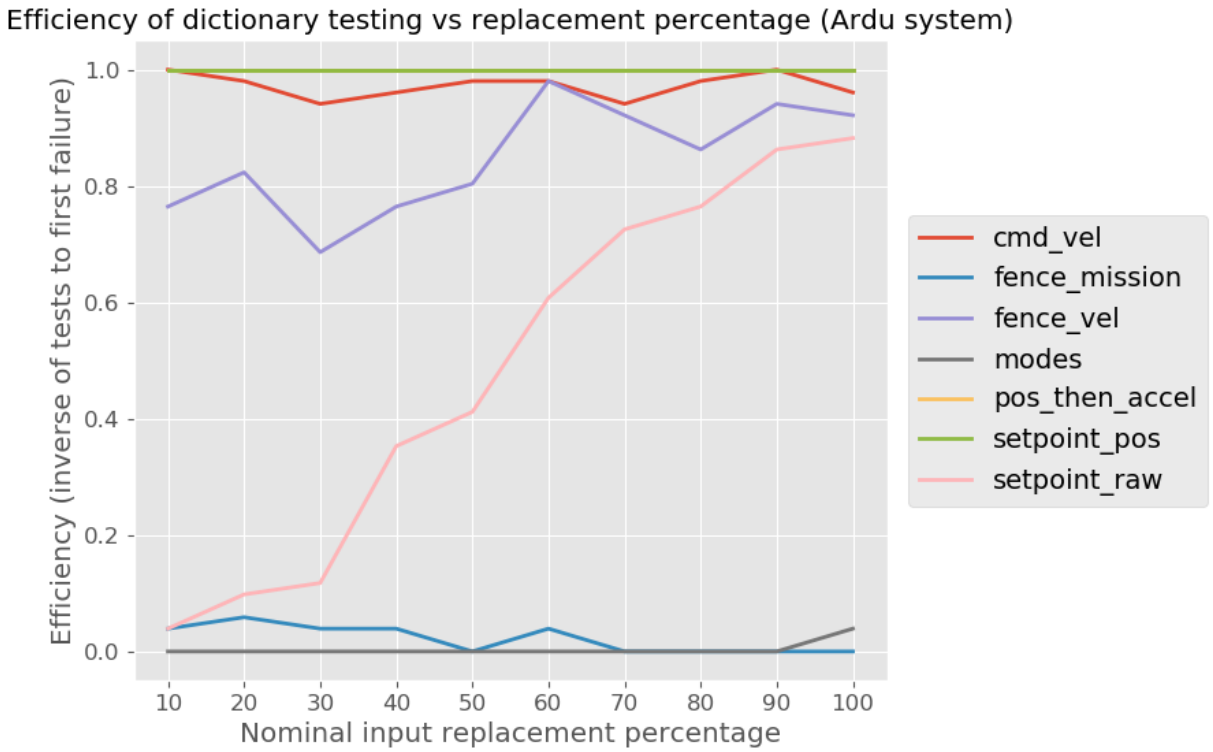

Figure C.1: Efficiency comparison for replacement percentage using dictionary-based testing on the Ardu system.

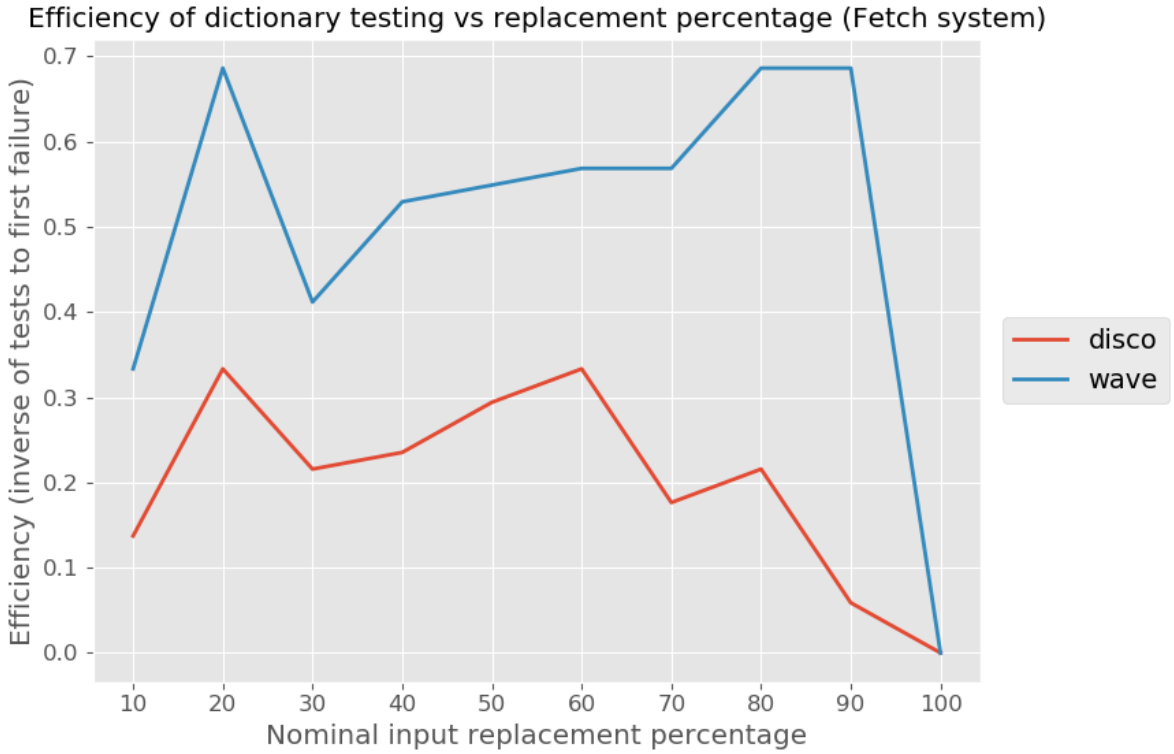

Figure C.2: Efficiency comparison for replacement percentage using dictionary-based testing on the Fetch system.

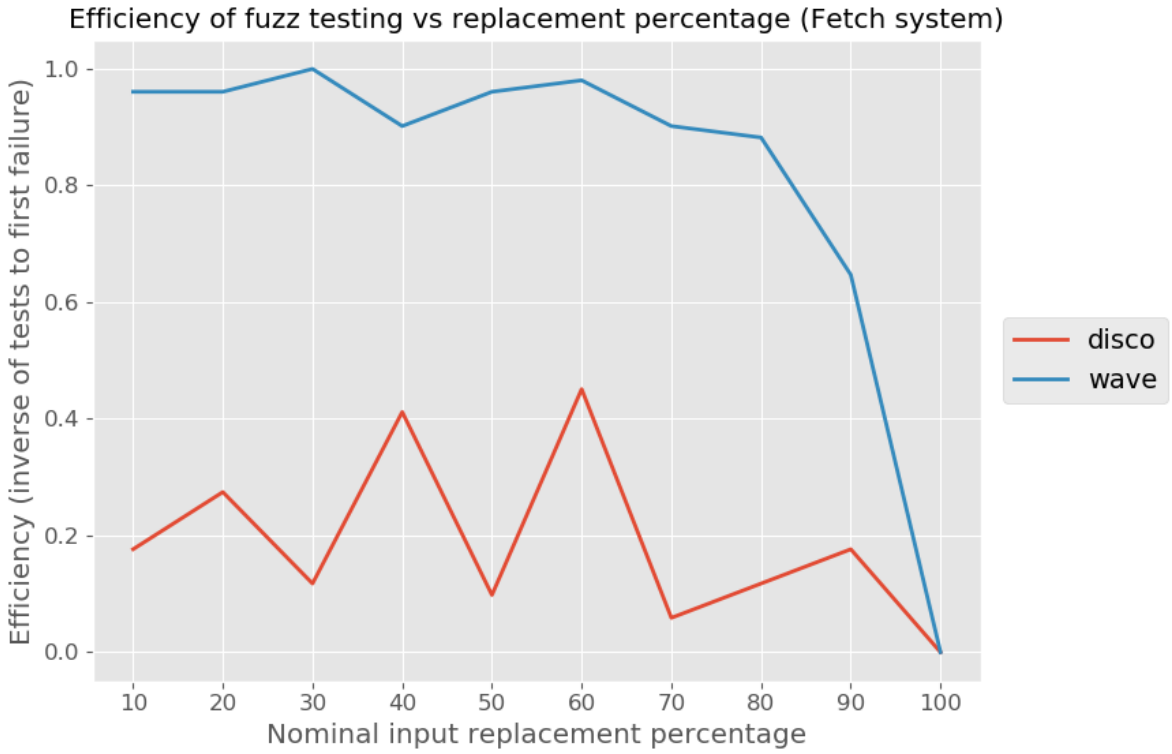

Figure C.3: Efficiency comparison for replacement percentage using fuzz-based testing on the Fetch system.

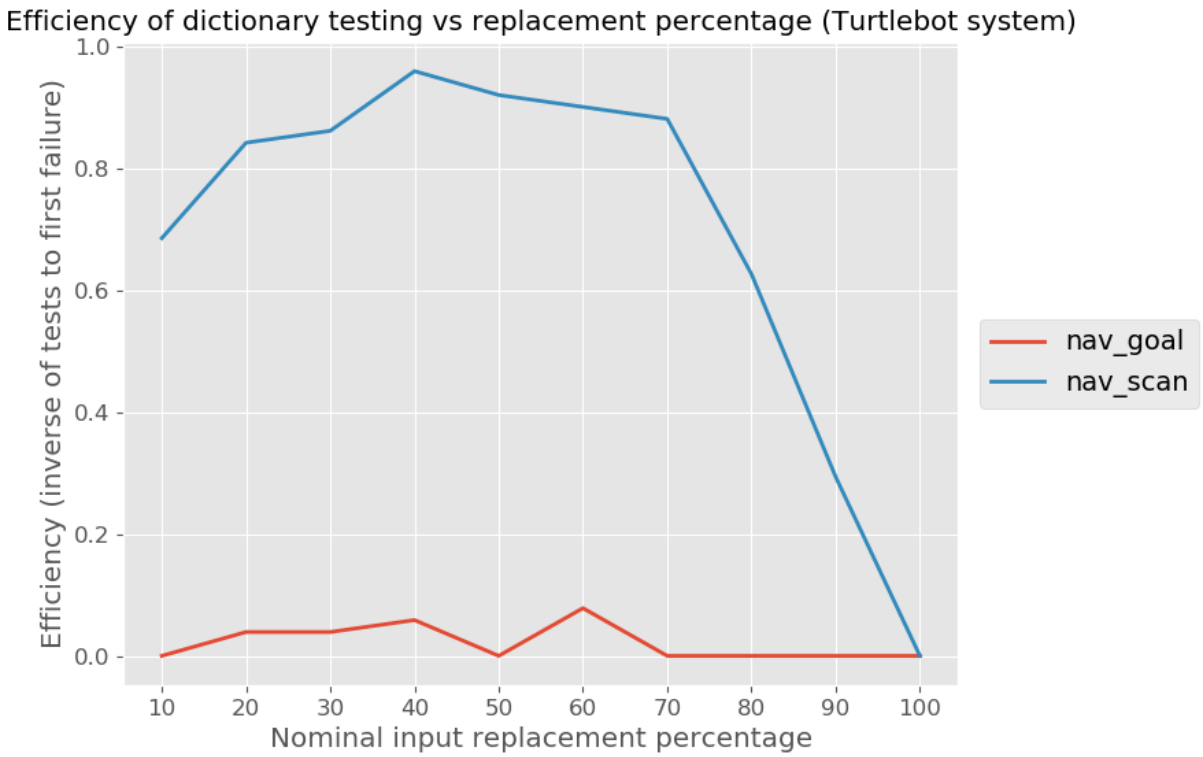

Figure C.4: Efficiency comparison for replacement percentage using dictionary-based testing on the Turtlebot system.

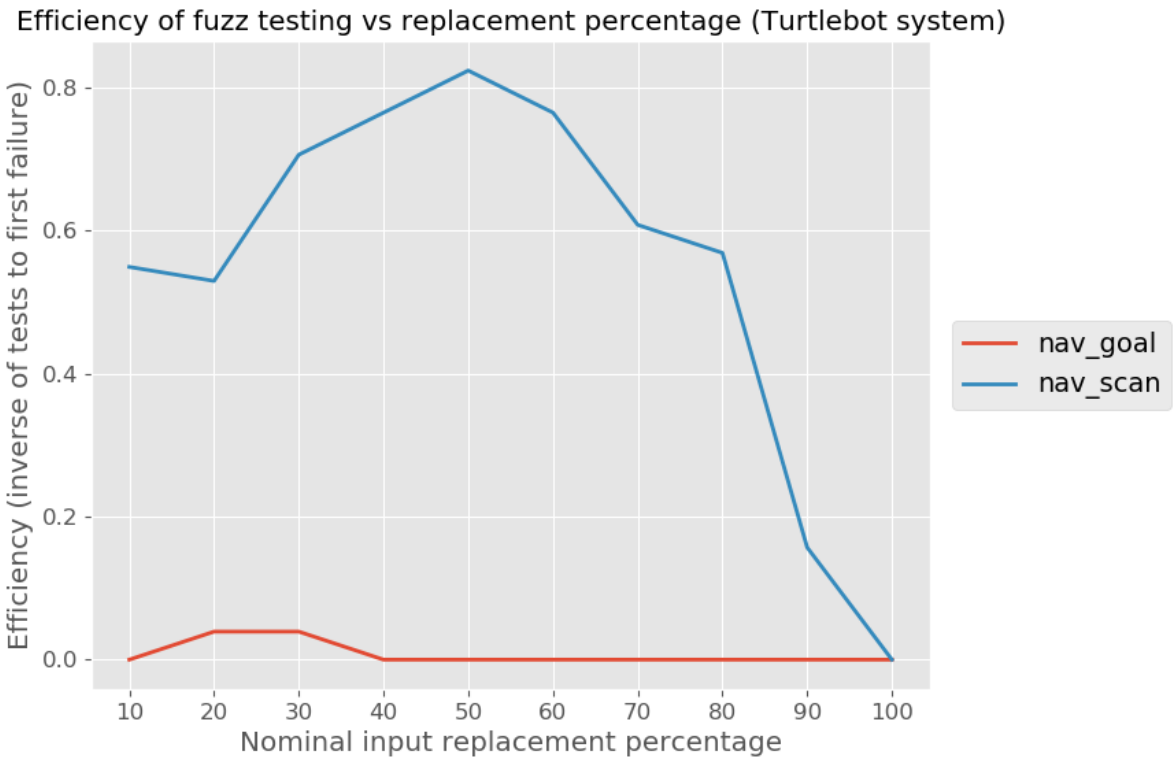

Figure C.5: Efficiency comparison for replacement percentage using fuzz-based testing on the Turtlebot system.

Bibliography

[1] american fuzzy lop [Online]. http://lcamtuf.coredump.cx/afl//. Accessed: 2019-11-18. 2.3.1

[2] Open source drone software. Trusted, versatile, open. ArduPilot [online]. http://ardupilot.org. Accessed: 2019-09-13. 3.5.2, 3.6

[3] Docker: Enterprise container platform [Online]. http://docker.com. Accessed: 2019-09-30. 2.

[4] Web archive: erlerobotics.com [Online]. http://web.archive.org/web/20161101000000*/http://erlerobotics.com. Defunct. Accessed via web.archive.org on 2019-09-13. 3.5.4

[5] Autonomous mobile robots that improve productivity — Fetch Robotics [Online]. http://fetchrobotics.com. Accessed: 2019-09-13. 3.5.1, 3.6

[6] Gazebo [Online]. http://gazebosim.org/. Accessed: 2019-09-13. 3.4.2

[7] Innok Heros [Online]. http://innok-robotics.de/en/products/heros. Accessed: 2019-09-13. 3.5.4

[8] Introduction - MAVLink developer guide [Online]. http://mavlink.io. Accessed: 2019-09-13. 3.5.2

[9] ROBOTIS – OP3 [Online]. http://robotis.us. Accessed: 2019-09-13. 3.5.4

[10] GitHub: google/oss-fuzz [Online]. http://github.com/google/oss-fuzz.

Accessed: 2019-11-18. 2.3.1

[11] Peach fuzzer: Discover unknown vulnerabilities [Online]. http://peach.tech/. Accessed: 2019-11-18. 2.3.1

[12] ROS.org [Online]. http://ros.org. Accessed: 2019-09-13. 3.4.2

[13] TurtleBot [Online]. http://turtlebot.com. Accessed: 2019-09-13. 3.5.3, 3.6

[14] Shipra Agrawal and Navin Goyal. Analysis of Thompson sampling for the multi-armed bandit problem. In *Conference on Learning Theory*, pages 39–1, 2012. 7.7.2

[15] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016. 1, 2.1

[16] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013. 2.6

[17] Ardupilot. ROS - dev documentation [Online]. http://ardupilot.org/dev/docs/ros.html. Accessed: 2019-09-13. 3.5.2

[18] Narayanaswamy Balakrishnan and Valery B Nevzorov. *A primer on statistical distributions*. John Wiley & Sons, 2004. 4.3

[19] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5): 507–525, 2014. 2.4

[20] Victor R Basili and Richard W Selby. Comparing the effectiveness of software testing strategies. *IEEE transactions on software engineering*, (12):1278–1296, 1987. 2.6

[21] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995. 2.1

[22] George A Bekey. *Autonomous robots: from biological inspiration to implementation and control*. MIT press, 2005. 2.10

[23] Donald A Berry and Bert Fristedt. Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability). *London: Chapman and Hall*, 5:71–87, 1985. 7.7

[24] Antonia Bertolino. Software testing research and practice. In *International Workshop on Abstract State Machines*, pages 1–21. Springer, 2003. 2.5

[25] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007. 2.1

[26] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 122–131. IEEE Press, 2013. 2.3.1

[27] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, 1993. 2.10.1

[28] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008. 2.5

[29] CAPEC. Capec-94: Man in the middle attack [Online]. `http://capec.mitre.org/data/definitions/94.html`. Accessed: 2019-11-30. 3.1

[30] National Robotics Engineering Center. Stress Testing for Autonomous Systems [Online]. `http://www.cmu.edu/nrec/solutions/defense/stress-testing-autonomous-systems.html`. Accessed: 2019-09-13. 3.4.2

[31] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015. 2.3.5

[32] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009. 2.4

[33] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 197–208, 2013. 2.8, 4.1.2

[34] Hana Chockler, Orna Kupferman, and Moshe Y Vardi. Coverage metrics for formal verification. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 111–125. Springer, 2003. 2.5

[35] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys & Tutorials*, 18(3):2027–2051, 2016. 2.3.4

[36] Domenico Cotroneo, Domenico Di Leo, Roberto Natella, and Roberto Pietrantuono. A case study on state-based robustness testing of an operating system for the avionic domain. In *International Conference on Computer Safety, Reliability, and Security*, pages 213–227. Springer, 2011. 2.9

[37] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998. 2.3.1

[38] Siddhartha R Dalal, Ashish Jain, Nachimuthu Karunanithi, JM Leaton, Christopher M Lott, Gardner C Patton, and Bruce M Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, pages 285–294. IEEE, 1999. 2.3.1

[39] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. Soft-

ware engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013. 2.10.2

[40] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 31–36. ACM, 2007. 2.3.1

[41] Joe W Duran and Simeon C Ntafos. An evaluation of random testing. *IEEE transactions on software engineering*, (4):438–444, 1984. 2.3.5

[42] Stuart I Feldman and Channing B Brown. Igor: A system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 112–123, 1988. 2.8

[43] Phyllis G Frankl and Stewart N Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19 (8):774–787, 1993. 2.5

[44] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988. 2.5

[45] Jerry Gao, H-SJ Tsao, and Ye Wu. *Testing and quality assurance for component-based software*. Artech House, 2003. 2.2

[46] Willow Garage. PR2 overview — Willow Garage. `http://www.willowgarage.com/pages/pr2/overview`. Accessed: 2019-09-13. 3.5.4

[47] Sylvain Gelly, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, and Yann Kalemkarian. On the parallelization of Monte-Carlo planning. In *ICINCO*, 2008. 8.1.1.3

[48] Anup K Ghosh, Matthew Schmid, and Viren Shah. Testing the robustness of Windows NT software. In *Proceedings Ninth International Symposium on Software Reliability En-*

gineering (Cat. No. 98TB100257), pages 231–235. IEEE, 1998. 2.3.5

[49] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215, 2008. 2.6

[50] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008. 2.3.1

[51] Amrit L Goel. Software reliability models: Assumptions, limitations, and applicability. *Software Engineering, IEEE Transactions on*, (12):1411–1423, 1985. 2.6.1

[52] Mats Grindal, Jeff Offutt, and Sten F Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005. 2.3.3, 2.5, 8.1

[53] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 2002. 2.6

[54] Philipp Helle, Wladimir Schamai, and Carsten Strobel. Testing of autonomous systems–challenges and current state-of-the-art. In *INCOSE International Symposium*, volume 26, pages 571–584. Wiley Online Library, 2016. 1.1.1, 2.10.2

[55] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN notices*, 39 (12):92–106, 2004. 2.1

[56] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997. 2.3.4

[57] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press, 1994. 2.5

[58] Casidhe Hutchison. Lightweight formalizations of system safety using runtime monitoring of safety proof assumptions. Master's thesis, Carnegie Mellon University Pittsburgh, PA, 2016. 2.10.1

[59] Casidhe Hutchison, Milda Zizyte, Patrick E Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. Robustness testing of autonomy software. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 276–285. ACM, 2018. (document), 2.10.2, 3.1, 3.1, 3.7.1, 5.3.3

[60] IEEE Computer Society. IEEE standard classification for software anomalies. *IEEE Std*, 1044, 1993. 2.3

[61] Andréas Johansson, Neeraj Suri, and Brendan Murphy. On the selection of error model(s) for OS robustness evaluation. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 502–511. IEEE, 2007. 2.6

[62] Aaron Kane. *Runtime monitoring for safety-critical embedded systems*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2015. 2.10.1

[63] Cem Kaner, Jack Falk, and Hung Q Nguyen. *Testing computer software*. John Wiley & Sons, 1999. 2.1

[64] Deborah Katz, Zizyte Milda, Casidhe Hutchison, David Guttendorf, Patrick Lanigan, Eric Sample, Philip Koopman, Michael Wagner, and Claire Le Goues. Robustness Inside-Out Testing (RIOT) Project. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'20)*, 2020. Conference paper accepted and to be presented. 8.1, 9.3

[65] Mohd Ehmer Khan, Farmeena Khan, et al. A comparative study of white box, black box and grey box testing techniques. *Int. J. Adv. Comput. Sci. Appl*, 3(6), 2012. 2.2

[66] Taghi M Khoshgoftaar and Timothy G Woodcock. Software reliability model selection: a cast study. In *Proceedings. 1991 International Symposium on Software Reliability Engineering*, pages 183–191. IEEE, 1991. 2.6.1

[67] Philip Koopman. *Better embedded system software*. Drumnadrochit Education, 2010. 8.2

[68] Philip Koopman and Michael Wagner. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety*, 4(1):15–24, 2016. 3.1

[69] Philip Koopman and Michael Wagner. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017. 2.10.1

[70] Philip Koopman, Kobey DeVale, and John DeVale. Interface robustness testing: Experience and lessons learned from the Ballista project. *Dependability Benchmarking for Computer Systems*, 72:201, 2008. 2.3.2, 3.3.3, 4.2

[71] Dimitris E Koulouriotis and A Xanthopoulos. Reinforcement learning and evolutionary algorithms for non-stationary multi-armed bandit problems. *Applied Mathematics and Computation*, 196(2):913–922, 2008. 7.7

[72] Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. Automated robustness testing of off-the-shelf software components. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224)*, pages 230–239. IEEE, 1998. 1, 2.3.2, 2.4

[73] Volodymyr Kuleshov and Doina Precup. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014. 7.7.1, 7.7.2

[74] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006. 2.7

[75] PS Loo and WK Tsai. Random testing revisited. *Information and Software Technology*, 30(7):402–417, 1988. 2.6

[76] Robyn R Lutz. Software engineering for safety: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 213–226. ACM, 2000. 2.10.1

[77] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of*

Software Engineering, pages 224–234. ACM, 2013. 2.3.1

[78] Alexis C Madrigal. Inside Waymo's secret world for training self-driving cars. *The Atlantic*, 23, 2017. 3.1

[79] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of Android applications on the cloud. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 22–28. IEEE, 2012. 2.3.1

[80] Yashwant K Malaiya. Antirandom testing: Getting the most out of black-box testing. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 86–95. IEEE, 1995. 2.3.3

[81] Brian Marick et al. How to misuse code coverage. In *Proceedings of the 16th International Conference on Testing Computer Software*, pages 16–18, 1999. 2.5

[82] Johannes Mayer and Christoph Schneckenburger. An empirical analysis and comparison of random testing techniques. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 105–114. ACM, 2006. 2.6

[83] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004. 2.3.5

[84] Tim Menzies and Andrian Marcus. Automated severity assessment of software defect reports. In *2008 IEEE International Conference on Software Maintenance*, pages 346–355. IEEE, 2008. 2.8

[85] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, December 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL `http://doi.acm.org/10.1145/96267.96279`. 2.3.1, 3.3.2

[86] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy,

Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, Technical Report CS-TR-1995-1268, University of Wisconsin, 1995. 2.3.1

[87] Barton P Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of MacOS applications using random testing. In *Proceedings of the 1st international workshop on Random testing*, pages 46–54. ACM, 2006. 2.3.1

[88] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. Technical report, 2007. 2.3.5

[89] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12, 2008. 8.2

[90] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011. 1, 2.1

[91] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2): 29–50, 2012. 2.2

[92] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005. 2.5

[93] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988. 2.3.5

[94] Brian S Pak. *Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2012. 2.9

[95] Yuanli Pei, Arpit Christi, Xiaoli Fern, Alex Groce, and Weng-Keen Wong. Taming a fuzzer using delta debugging trails. In *2014 IEEE International Conference on Data Mining Workshop*, pages 840–843. IEEE, 2014. 2.8

[96] Jane Radatz, Anne Geraci, and Freny Katki. IEEE standard glossary of software engineer-

ing terminology. *IEEE Std*, 610121990(121990):3, 1990. 1

[97] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017. 2.6

[98] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings Fourth International Software Metrics Symposium*, pages 64–73. IEEE, 1997. 2.3.5

[99] ROS.org. mavros - ROS wiki [Online]. `http://http://wiki.ros.org/mavros`. Accessed: 2019-09-13. 3.5.2

[100] Fares Saad-Khorchef, Antoine Rollet, and Richard Castanet. A framework and a tool for robustness testing of communicating software. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1461–1466. ACM, 2007. 2.3.1

[101] Robert G Sargent. Verification and validation of simulation models. In *Proceedings of the 2010 Winter Simulation Conference*, pages 166–183. IEEE, 2010. 3.1

[102] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007. 2.3.1, 2.3.5

[103] Christopher Steven Timperley, Afsoon Afzal, Deborah S Katz, Jam Marcos Hernandez, and Claire Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 331–342. IEEE, 2018. 2.10.2

[104] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. Secfuzz: Fuzz-testing security protocols. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 1–7. IEEE, 2012. 2.3.4

[105] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

2.3.1

[106] Paul Vernaza, David Guttendorf, Michael Wagner, and Philip Koopman. Learning product set models of fault triggers in high-dimensional software interfaces. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3506–3511. IEEE, 2015. 2.7, 2.

[107] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE transactions on software engineering*, 17(7):703–711, 1991. 2.3.5

[108] James A Whittaker and Michael G Thomason. A Markov chain model for statistical software testing. *IEEE Transactions on Software engineering*, 20(10):812–824, 1994. 2.3.5

[109] Marco Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12:3, 2012. 7.7

[110] Stefan Winter, Constantin Sârbu, Neeraj Suri, and Brendan Murphy. The impact of fault models on software robustness evaluations. In *2011 33rd international conference on software engineering (ICSE)*, pages 51–60. IEEE, 2011. 2.6

[111] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522. ACM, 2013. 2.3.5, 7.7.2

[112] Alan Wood. Predicting software reliability. *Computer*, 29(11):69–77, 1996. 2.6.1

[113] Yuen Tak Yu and Man Fai Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *Journal of Systems and Software*, 79(5):577–590, 2006. 2.5

[114] Andreas Zeller. Automated debugging: Are we close? *Computer*, 34(11):26–31, 2001. 2.8

[115] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input.

IEEE Transactions on Software Engineering, 28(2):183–200, 2002. 2.7, 1.

[116] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process*, 28(3): 150–176, 2016. 2.8