# Elements of Unsupervised Testing for Software Systems of Autonomous Vehicles

**Sean Hickey and Nickolas Vlahopoulos**
**University of Michigan**
**Ann Arbor, MI**
**hicsea@umich.edu, nickvl@umich.edu**

**Jonathon M. Smereka**
**US Army DEVCOM GVSC**
**Warren, MI**
**jonathon.m.smereka.civ@army.mil**

**Geng Zhang**
**Michigan Engineering Services**
**Ann Arbor, MI**
**gengz@miengsrv.com**

## ABSTRACT

Robotic vehicles are hardware-software systems. These systems combine hardware capabilities with software driven operations. The significant need for verification and validation of the software used to operate robotic vehicles is well recognized. Due to the labor-intensive procedures, typically half of the entire software development lifecycle cost originates from testing efforts, while retesting software after upgrades accounts for eighty percent of the entire maintenance cost. In this paper an overview of methods for automated software testing based on fuzzing techniques and evolutionary optimization methods are discussed first. Then the main elements of a new unsupervised software testing approach are presented. The term "unsupervised" indicates expecting minimal human effort in defining test cases and expected outcomes. The development is targeting software systems based on the Robot Operating System 2 (ROS2) library. A Genetic Algorithm (GA) is used for determining the most effective artificial changes introduced in the data flow of the software that is being tested in order to generate errors. Errors that account for unexpected termination, inactivity, collisions and excessive run times (associated with slowed operation) are considered. The GA makes selections in order to increase the errors while at the same time investigate a wide range of variations in the data flow. An interceptor code and a code for monitoring the encountered errors are integrated with the software system which is tested. The former intercepts the data flow, alters the data flow based on the instructions provided by the GA and publishes the altered data flow. The monitoring code identifies successful termination, or the type of error that was encountered. This information is used by the GA for determining the next round of artificial alterations. The public domain Autoware Universe (AU) system for operating autonomous vehicles and a lightweight simulator are used for demonstrating how the main elements of the new software testing process operate.

## ABOUT THE AUTHORS

**Sean Hickey** is a PhD student at the University of Michigan. His professional experience includes the development and verification of robotics simulation software, the optimization and design of offshore structures including offshore wind and carbon sequestration platforms, and the implementation of user interfaces for marine robotics applications. His current research interests are centered around the unsupervised verification of ground autonomous vehicles.

**Nickolas Vlahopoulos** has been a Professor at the University of Michigan for 25 years and has also worked in the Industry for 7 years prior to his academic career; he has published over 100 papers and has graduated 24 PhD students.

**Jonathon Smereka** is the Ground Vehicle Robotics (GVR) Senior Technical Expert for research at the U.S. Army DEVCOM Ground Vehicle Systems Center (GVSC, formerly TARDEC) in Warren, Michigan. He received his M.S. and Ph.D. in Computer and Electrical Engineering from Carnegie Mellon University. He has worked in research and development of artificial intelligence, machine learning, image and signal processing, and computer vision with respect to biometrics, automated target recognition (ATR), and ground vehicle autonomy.

**Geng Zhang** is a R&D engineer at Michigan Engineering Services; he has 17 years of experience and has published 30 technical papers.

# Elements of Unsupervised Testing for Software Systems of Autonomous Vehicles

**Sean Hickey and Nickolas Vlahopoulos**
**University of Michigan**
**Ann Arbor, MI**
**hicsea@umich.edu, nickvl@umich.edu**

**Jonathon M. Smereka**
**US Army DEVCOM GVSC**
**Warren, MI**
**jonathon.m.smereka.civ@army.mil**

**Geng Zhang**
**Michigan Engineering Services**
**Ann Arbor, MI**
**gengz@miengsrv.com**

## INTRODUCTION

Army robotic ground vehicles are hardware-software systems. These systems combine hardware propulsion, actuation, sensing and other such capabilities with the decision-making, data storage, and other software driven operations. The Army has developed methods for reliability testing for its hardware (i.e. vehicle) systems [1]. However, the need for verification and validation of the long-term software systems of ground robotic vehicles is identified by DEVCOM GVSC as one of the GVSC Robotic and Autonomy needs [2, 3]. The need for tools and capabilities that can be used for verification and validation of software is well recognized in the literature, particularly since half of the entire software development lifecycle cost originates from testing efforts [4]. In addition retesting software after upgrades accounts for eighty percent of the entire maintenance cost [5].

Fuzzing is a technique for automated software testing. The main concept is to use random, invalid and unexpected inputs for detecting failures and crashes (e.g. buffer overflows) [6]. A variety of automated testing fuzzing approaches and tools have been developed, such as CLUSTERFUZZ by Google, or ONEFUZZ by Microsoft. Fuzzing approaches can be distinguished depending on whether they use existing input or not [7]. If the fuzzing reuses existing input, it is called mutation based fuzzer, as it mutates the provided input by making small modifications. In contrast generation based fuzzers generate the input from scratch and therefore do not depend on the existence of inputs. The main goal of the fuzzing technique is to systematically increase code coverage by executing the code under the various inputs that the fuzzer generates. Increasing code coverage allows the discovery of potentially harmful corner cases [6]. The mutation and generation processes that fuzzing is using for generating the test cases is combined with the iterative process of an optimization genetic algorithm (GA) in order to introduce robustness in the fuzzing process [8]. It has been reported that a genetic based search discovers 50% more program failures compared to random fuzzing [9].

Search based software testing driven by evolutionary optimization methods (most commonly by GA) has been applied across the spectrum of test case design methods [10]. Generation of software tests is generally considered as an undecidable problem, primarily due to many possible combinations of a program's input [11]. Genetic algorithms are highly applicable in testing coverage, timings, parameter values, finding calculation tolerances, bottlenecks, problematic input combinations and sequences. The genetic algorithms adapt to a given problem, therefore a genetic algorithm based tester generates several parameter combinations that reveal minor bugs and based on this information it constructs sequences that reveal a much higher amount of bugs than random testing [12]. GA have been used for generating a good test suite efficiently [13]. A good test suite is defined as one that consists of test cases that not only focuses on error causing areas of the code, but provides a balanced representation of all errors. The fitness function that the GA uses for generating a good test suite includes the Euclidian distance between test cases and the novelty of a test case that creates errors against all other error-generating cases [13].

GA algorithms have been used in a variety of autonomous vehicle applications for testing embedded software. The main novelty in each such research effort is associated with the definition of the fitness function that the GA is using for determining the selection of the test cases. When evaluating autonomous vehicle software controllers, a set of

simulated fault scenarios is applied to the controller and a GA searches for significant combinations of faults [14]. In such application, the fitness can be measured as the difference between the controller's actual performance in a scenario against an ideal response. This definition requires a substantial effort for determining the expected ideal response. Instead, a fitness function that searches for "interesting" scenarios is very attractive because it avoids the aforementioned shortfalls. "Interesting" scenarios are those in which minimal fault activity causes a mission failure or a vehicle loss [14].

A GA has been used for testing the Advanced Driver Assistance System (ADAS) associated with the automated emergency breaking system of a vehicle [15]. The main function of the tested system is to use a vision based capability to identify pedestrians in front of the vehicle and to avoid collision. The commercial PreScan simulator [16] was used for determining and executing scenarios capturing various road traffic situations and different pedestrian to vehicle and vehicle-to-vehicle interactions. In this case the fitness function is defined based on the certainty of detection of the pedestrian, the minimum distance between the pedestrian and the field of view of the vehicle and the speed of the vehicle at the time of the collision (if it occurs). The GA is used for creating new test scenarios within a complex and multidimensional input space that identify the critical regions of the ADAS input space. Such characterizations help debug the system. They further help engineers identify environment conditions that likely lead to ADAS failures [16]. The information obtained through the combination of a GA search and the simulator is critical for developing reliable autonomous vehicles. Otherwise unrealistic time would be required for physical testing since the corner cases that reveal the weaknesses of the ADAS rarely appear in physical testing [17]. In addition, such testing is expensive and dangerous. A GA has also been used in self-driving car applications for creating virtual roads for testing the lane keeping capability of autonomous driving [18]. In similar applications of creating virtual driving environments using GA, racing tracks with various turns and speed profiles were used for increasing in virtual racing games the player's enjoyment [19] and for automatically generating challenging tracks based on players' bio-metric feedback [20]. A GA approach has also been used for automatically creating testing environments for a robot obstacle avoidance system and for a vehicle lane keeping assist system [21]. The deviation of the observed vs. the expected behavior and the diversity in test cases were combined for defining the fitness function. The GA based test development process was used to generate virtual environments of different types and complexity and reveal the system's fault early in the design stage [21].

The control software associated with the DARPA Fast Lightweight Autonomy program of an aerial vehicle was tested using a GA approach [22]. The GA approach outperformed the automated test generation methods which are commonly used to assess autonomous vehicle software robustness [23]. The ability of the autonomous aerial vehicle to follow a path and pass through an opening at the end of the path was used for defining the fitness function. A set of initial conditions and the timing of multiple faults inserted during the path comprised the group of parameters that the GA was selecting when generating the test cases.

Based on the outlined current state-of-the-art the main elements of an unsupervised testing process for ROS2 based software systems are identified and presented in this paper. The operation of each element (interception, termination, automation and GA driven testing process) is described. The AU [24] and a lightweight simulator are used for demonstrating the functionality of each element of the new testing process. The novel aspects presented in this paper are the developments that are necessary to make such a process work for a ROS2 system and the integration of a multi-objective GA that optimizes the performance of the software testing process.

## UNSUPERVISED SOFTWARE TESTING PROCESS

Figure 1 presents the flow chart of the testing process presented in this paper. Each main element of the process is numbered and labeled in red. The interceptor node is tailored to the module which is tested since it introduces modifications in the data flow entering the module. The interceptor also considers the operation of the ROS2 library which is used as a development platform of the software system. The termination monitoring determines the outcome associated with each set of modifications introduced by the interception in the data flow. Automation of the entire process is necessary in order to run multiple simulations with a different set of inputs each time. Finally, the GA is selecting the changes in the data flow of an entire population at each generation. All data sets of each generation are executed. During this process the diversity of the input data is monitored and the diversity in faults is determined along with the severity of the faults. These metrics are used by the GA for generating the population of the next generation. Information about each element of the unsupervised software testing process is presented in the next four sub-sections.

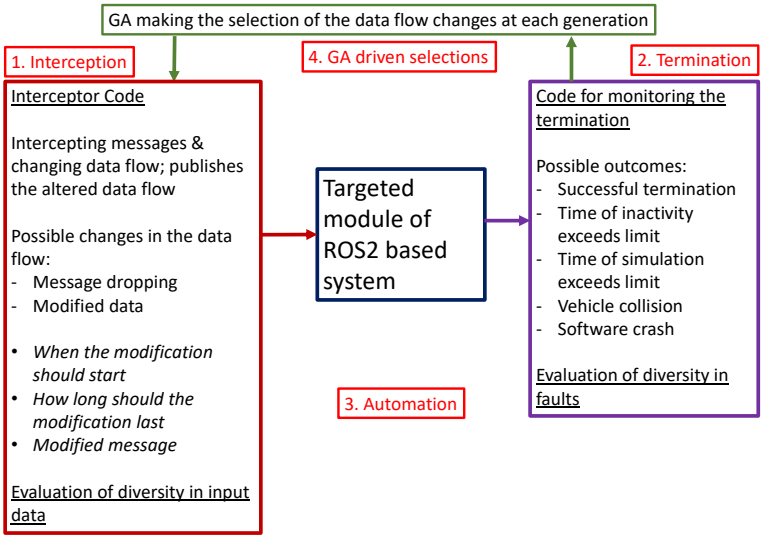


**Figure 1. Flow chart of unsupervised software testing process**

## Interception

Interception in a ROS2 code involves inserting a node in between a publisher node and a corresponding subscriber node in order to drop or modify the data as it is being passed between these two nodes (Figure 2). This idea can be scaled and applied to intercept data at any point in the network of nodes in a ROS2 system. By passing all of the messages from nodes under test through a single interceptor node, control over the flow of data in the system is established to introduce modifications to the data to identify faults.

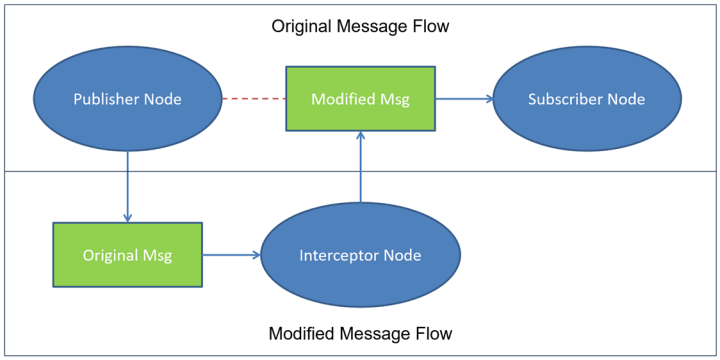


**Figure 2. Message interception concept diagram**

The interception strategy consists of a few steps depicted in Figure 2. First a set of topics are selected to be manipulated. They are intercepted at the publisher side by modifying the node launch files and remapping topics through the interceptor node. A single interceptor node subscribes to all targeted topics, renames them, introduces the desired modifications and then publishes them to the original topics. The interceptor reads a user input file, which specifies the topics to be manipulated, the modifications to be introduced for each topic, and the timing for these modifications. With this information, the interceptor node can modify the data as it is passed through the node. In this implementation messages can be dropped or modified.

As an example of an interceptor, the implementation for the control module of AU is discussed. Eleven control topics that are published by the control module are passed through an interceptor node, which allows for these messages to be modified or dropped at any specified time in a simulation. An example of the “control_cmd” topic being intercepted is shown in Figure 3. If no modifications are specified for a given topic, the message is passed through unmodified to preserve the original functionality of the system. This was tested and verified by implementing the interceptor node and passing each of these topics through without specifying any modifications to

the values, which led to the system functioning the same way before the interceptor node was implemented. This demonstrates that the interceptor node can be introduced without altering the behavior of the system. This also shows that the interceptor node can be scaled to intercept any number of topics in the system independently.

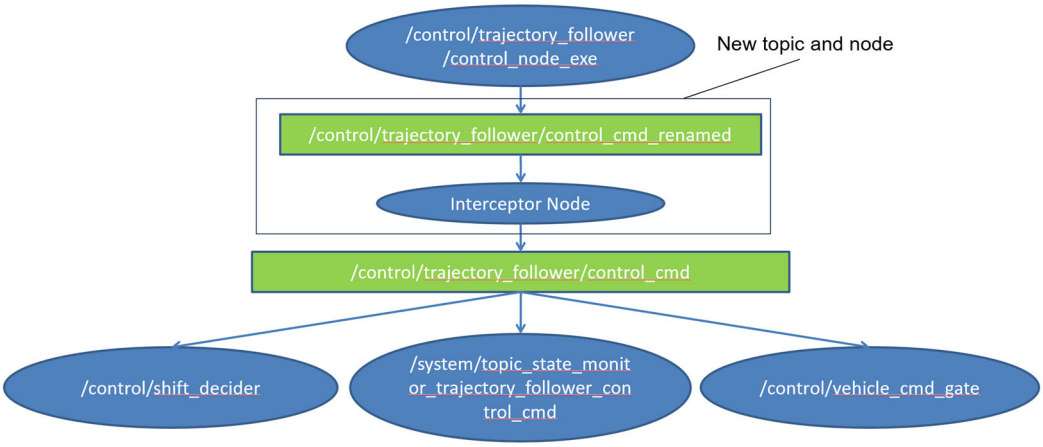


**Figure 3. Message interception example showing the "control_cmd" topic being intercepted**

**Termination**

Monitoring the termination of a simulation is crucial for evaluating the performance and the effectiveness of fault detection for a set of simulation inputs. A bash script is implemented to automatically run the simulations and monitor how they are terminated. The script automatically launches the simulation using a user input file along with ROS2 launch commands and constantly monitors the state of the vehicle along with the processes for each active node.

Five types of termination events are considered. Each has its own termination code that is output to a file along with data from the simulation. This information is used to measure the performance of the run. The termination codes and the types of termination considered in this work are summarized in Table 1.

The first type is a successful run. This occurs when the vehicle correctly follows its path and reaches its target pose from the initial pose. To monitor for a successful run, the script tracks the state of the vehicle and ends the simulation once the target pose is reached.

The second type is a system crash. This occurs when one or more ROS2 nodes unexpectedly stops running, or other errors cause the simulation to terminate. Monitoring for system crashes is done through a script that monitors the active processes when the ROS2 nodes are launched. If any nodes are killed unexpectedly during the simulation, it indicates a system crash, causing the simulation to be unintentionally terminated.

The third type is a timeout fault. This occurs when the vehicle takes longer than a user-specified time limit to reach its goal. To monitor for timeouts, the script starts a timer at the beginning of the simulation. If the vehicle doesn't reach its goal within the preset limit, the simulation is terminated and a timeout fault is recorded.

The fourth type is a hung fault. This happens when the vehicle remains "hung" or stationary with zero velocity for a user-specified time limit. To detect a hung state, the script monitors the vehicle's velocity. If it remains at zero consecutively for the specified time limit, the simulation is terminated and a hung fault is recorded.

The final fault type is a collision. This occurs when the vehicle hits an obstacle or enters an invalid area. To monitor such faults, the script tracks the vehicle's state and intentionally terminates the simulation if a collision is detected or if the vehicle enters an invalid area. A summary of the aforementioned termination logic is presented in Figure 4.

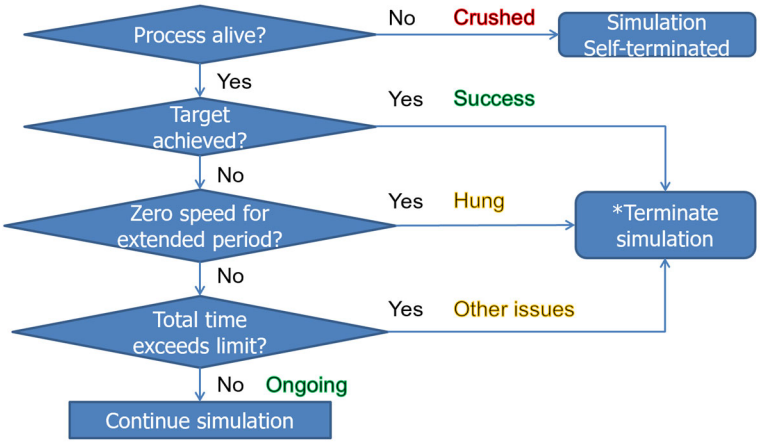


**Figure 4. Flow chart demonstrating termination logic used to automate simulation runs**

**Automation**

Automation is essential for executing multiple simulations with different inputs each time. Automating multiple simulations allows a GA to be applied to the software testing process. The automation process is based on using ROS2 commands in the Linux command line.

Typically, running a simulation in the AU Graphical User Interface window involves three steps: defining an initial pose, defining a target pose, and pressing the engage button after the route is planned. These steps can be automated using ROS2 commands in the Linux command line to replicate a single simulation without manually interacting with the Graphical User Interface window. Instead of clicking buttons to set the initial and target poses, a pose message can be published to the corresponding topics through the command line. Once these poses are established and the route is planned, the engage service can be called via the command line to run the simulation.

By combining these steps with the termination monitoring described previously, a single AU simulation can be generated from start to finish using only the Linux command line. Using shell scripts, we can loop over many simulations with different inputs, automating multiple runs in a row. Such automation steps were implemented for automatically driving multiple runs of AU simulations, as an example of automatically running a ROS2 based system.

The automation approach is flexible and scalable. Any number of simulations can be specified in a single, well-formatted user input file that contains an array of simulation inputs. The output for one of these automated runs is also a well-formatted output file that contains an array of information where the data at each index is the output of a single run, including the termination code and the data from the simulation. This allows any number of simulations to be processed all at the same time by a GA. Pseudocode for using this automated procedure inside a GA is shown in Figure 5.

**1 Performance Monitoring Script**

```
Start with initial input file with k simulations
for n generations do
    for k simulations do
        while monitoring termination behavior do
            Launch ROS2 simulation k from input file
            Log data and termination output to file
        end while
    end for
    Calculate error metric from output file
    Genetic algorithm uses error metric to generate new input file
end for
```

**Figure 5. Pseudocode outlining the process for automating many simulations inside a genetic algorithm**

**Genetic Algorithm and Lightweight Simulator**

A lightweight simulator developed using ROS2, that emulates the control module of AU, is integrated with an automated GA driven process in order to demonstrate the operation of the unsupervised software testing approach. Utilizing a lightweight simulator makes it possible to ensure that the multiple performance metrics encourage the genetic algorithm to perform as expected. This lightweight simulator, which we call TankSim, is adapted from an available architecture created for ROS2 tutorials [25]. A PID controller with obstacle avoidance was utilized to move a vehicle from an initial location to a target location, just like in AU.

In addition to engaging the aforementioned procedures for intercepting data, monitoring simulation termination, and automating multiple runs, these functionalities were wrapped with the NSGA-II genetic algorithm. The NSGA-II optimizes multiple objectives simultaneously and comprises a well-established multi-objective GA [26]. In this development the three objectives are to maximize fault severity, maximize fault diversity, and maximize input diversity. Since the NSGA-II minimizes the objective functions by default, a numerical value for each one of the three objectives is calculated for every simulation. The sign is reversed for each numerical value so that the objectives are maximized by the algorithm.

The first objective is to maximize the severity of the faults observed. Severity scores were assigned to each fault, as shown in Table 1. The objective function is simply the negative value of the "Fault Severity Score" shown in Table 1. As the type of termination is identified (Figure 4), the appropriate score is assigned. The severity scores comprise user defined information.

**Table 1. Fault Severity Scores**

| Termination Type | Termination Code | Fault Severity Score |
|---|---|---|
| **Success: Reached target** | 0 | 0.0 |
| **System Crash** | 1 | 1.0 |
| **Timeout** | 2 | 0.6 |
| **Hung** | 3 | 0.8 |
| **Collision** | 4 | 0.9 |

The second objective is to maximize the diversity of the faults identified by the population in each generation. This objective was adapted from [27]. This metric represents a desired proportion of the count of each fault relative to the total number of faults encountered in each generation. Since there are four types of faults, a desired ratio of 0.25 is used for each proportion of fault type count to total fault count. Each simulation in a population is exponentially penalized for higher magnitudes in the difference between the observed proportion and the desired proportion. Equation (1) shows the calculation of this objective value. This objective function returns a value of 1 when the proportion is equal to 0.25, and returns a value that gets exponentially closer to zero the farther away the encountered proportion is from 0.25.

$$\text{Fault Diversity} = \left(1 - \sqrt{\left|\frac{c}{t} - 0.25\right|}\right)^{10} \quad \text{if } current\,code \neq 0, \text{ else } 0 \tag{1}$$

Notation:

c – fault code count associated with the termination code of the current simulation
t – total fault code count of the current generation
current code – the termination code of the current simulation

The third objective is to maximize the diversity of the inputs generated by the genetic algorithm, representing a diverse set of test cases. Higher diversity indicates a greater exploration of the input space. Input diversity objectives are targeted by maintaining a "fossil record" of all previous simulations from each generation [28]. To measure the diversity of a set of inputs for a simulation, each input parameter is compared to the rest of the current generation and every fossil record seen so far from previous generations. The objective is calculated using equation (2).

$$\text{Input Diversity} = \frac{1}{|\text{Population+Fossils}|}\sum_{\text{individual}\in\text{Population+Fossils}}\sqrt{\sum_{j=1}^{params}\left(\frac{x_j-\text{individual}_j}{x_{j,\max}}\right)^2} \quad (2)$$

Notation:

Population – the set of all simulations (population members) in the current generation
Fossils – the set of all simulations (population members) from past generations
params – parameters that make up the inputs of a simulation, with j indicating the index of the parameter
$x_j$ – the j-th parameter value of the current simulation
$x_{j,\max}$ – the upper bound value of the j-th parameter
$\text{individual}_j$ – the j-th parameter value of the individual being compared to the current simulation

## DEMONSTRATION OF UNSUPERVISED SOFTWARE TESTING

In order to demonstrate how the GA driven automated software testing process operates, the TankSim simulator is used as the software system where faults will be pursued by the GA by dropping messages and modifying data (Figure 1). Through this example it is demonstrated that the elements of the process (interception, termination and automation) operate properly. Additionally, by comparing the faults of the first generation to the faults of the last generation it is demonstrated that the GA selections achieve the objectives of creating faults with high severity, that the generated faults are balanced and that there is input diversity in the test cases considered throughout all generations.

The interceptor node (Figure 3) accesses the 23 different parameters that are summarized in Table 2. The GA determines the values assigned to these parameters (Figure 5) for the simulations that comprise each generation. The selections made by the GA attempt to increase the severity of the faults, generate diverse faults and utilize diverse selections for the parameters in Table 2 as they try to generate the faults.

**Table 2. Parameters determined by the GA for each population member of each generation**

| |
|---|
| **Starting x and y coordinates and starting direction (3 parameters)** |
| **x and y coordinates of target location (2 parameters)** |
| **Maximum allowed velocity (1 parameter)** |
| **x and y coordinates for the center of an obstacle and diameter of obstacle (3 parameters)** |
| **Starting point for dropping velocity information and number of records to be dropped (2 parameters)** |
| **Starting point for dropping location information and number of records to be dropped (2 parameters)** |
| **Starting point for modifying velocity commandsand number of records to be modified (2 parameters)** |
| **Values for modified velocity command messages(3 parameters)** |
| **Starting point for modifying pose information and number of records to be modified (2 parameters)** |
| **Values for modified pose messages (3 parameters)** |

Representative results are presented in Figures 6 – 9. The GA makes selection of twenty simulation cases per generation for fifteen generations. Each selection is comprised by a different set of values for the 23 parameters summarized in Table 2. The progression of the severity error score is summarized over all generations in Figure 6. The error severity is an objective that competes with the fault diversity metric which is presented in Figure 7. The error severity prefers faults that exhibit high scores (Table 1) while the fault diversity score prefers an even distribution for the type of faults that are encountered. The results demonstrate how this balance is achieved towards later generations where the successful terminations with a zero severity score vanish and the fault diversity score becomes more balanced and even among each generation.

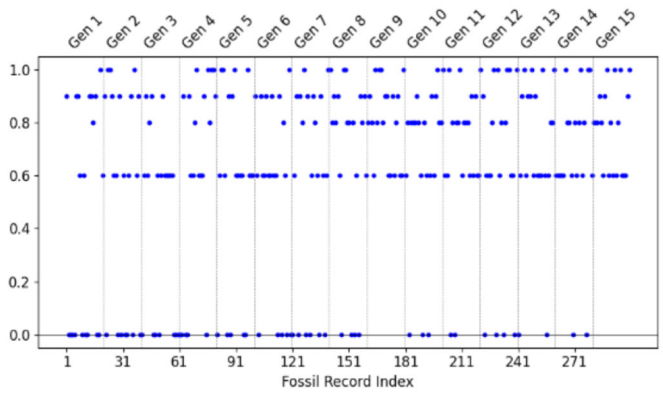


**Figure 6. Error severity score over 15 generations used in the demonstration**

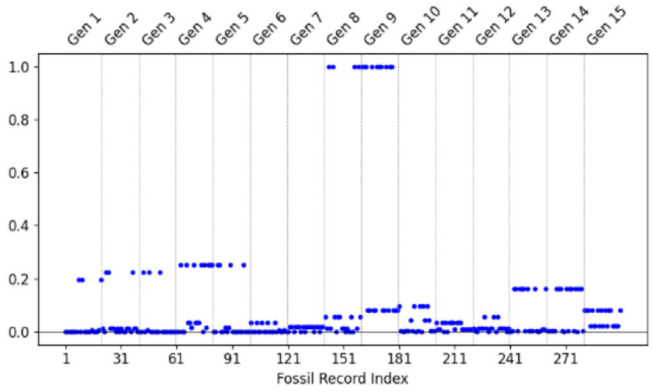


**Figure 7. Fault diversity score over 15 generations used in the demonstration**

Figure 8 presents the input diversity score. Unlike the other two metrics this is a cumulative score evaluated over all encountered generations. It gradually increases and plateaus at later generations. This is expected since more recently selected simulations cannot differ significantly from the increasing number of all previously encountered simulations. Finally Figure 9 presents the terminations encountered in the first and in the last generations of the GA. The selection of the simulations for the first generation is random, therefore the faults are representative of the ones that would be generated by a fuzzing technique. There is a large number of successful outcomes regardless of the artificial message dropping and changes in the data. In the last generation there are no successful outcomes and there is a comparable spread among the types of faults which are generated.

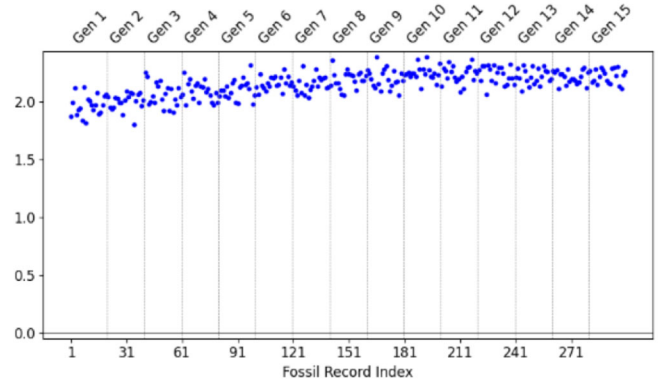


**Figure 8. Input diversity score over 15 generations used in the demonstration**

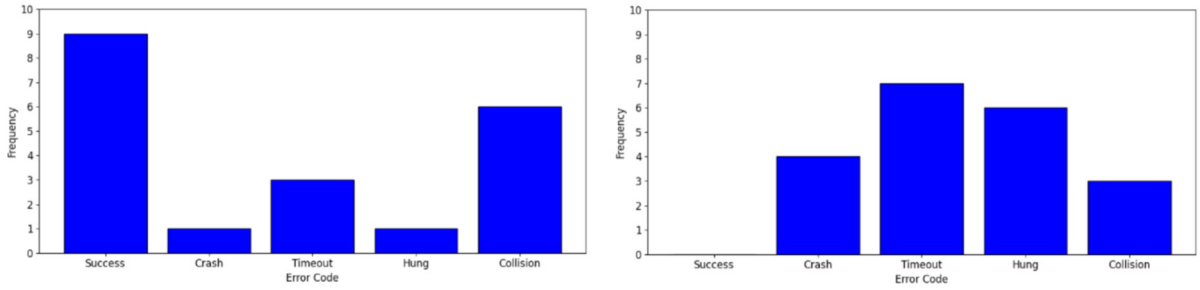


**Figure 9. First (left) and last (right) generation outcome distribution**

Overall the results obtained by this unsupervised testing demonstrate that the process operates as expected. It successfully increases the number of encountered faults, it balances the types of faults that are created and it diversifies the message dropping and data changes used for creating the simulations. In the future the unsupervised testing capability will be used for tracing the implementation or algorithmic improvements in the software that can avoid the faults in the presence of the artificial changes introduced by the GA.

## SUMMARY

The main elements of a new unsupervised software testing process for software systems based on the ROS2 library are presented. The process is based on automating the following functionalities: intercepting and modifying the data that enter the module that is tested; detecting the type of faults created by the artificial modification of the data; evaluating metrics associated with the diversity of the generated faults, the number and severity of the faults and the diversity of the artificial modifications; having a GA guide the selection of the data modifications based on the afore mentioned metrics. In the future work will be performed for tracing back into the tested software the sources of the faults created by the artificial modifications of the data in order to correct them. Additionally, known errors will be interjected in the software which is tested in order to ensure that the unsupervised testing process can discover them.

## ACKNOWLEDGEMENTS

The authors would like to acknowledge the technical and financial support of the Automotive Research Center (ARC) in accordance with Cooperative Agreement W56HZV-24-2-0001 U.S. Army DEVCOM Ground Vehicle Systems Center (GVSC) Warren, MI.

## REFERENCES

1. Army Regulation 702-19, Reliability, Availability, and Maintainability, Headquarters, Department of the Army, Washington, DC.
2. J. Smereka, “GVSC Robotic and Autonomy Needs,” US Army Combat Capabilities Development Command – Ground Vehicle Systems Center, April 2022.
3. J. Smereka, “U.S. ARMY COMBAT CAPABILITIES DEVELOPMENT COMMAND – GROUND VEHICLE SYSTEMS CENTER; Robotic Challenge Problems,” October 2022.
4. Yoo, S.; Harman, M. “Regression testing minimization, selection and prioritization: a survey,” Softw. Testing Verif. Reliab. 2012, 22, 67–120.
5. Li, Z.; Harman, M.; Hierons, R.M. “Search algorithms for regression test case prioritization,” IEEE Trans. Softw. Eng. 2007, 33, 225–237.
6. D. Kaufmann, A. Biere, “Fuzzing and Delta Debugging And-Inverter Graph Verification Tools,” Springer Nature Switzerland AG 2022, L. Kovacs, K. Mainke (Eds): TAP 2022, LNCS 13361, pp. 69-88, 2022.
7. A. Zeller, R. Gopinath, M. Bohme, G. Fraser C. Holler, “The Fuzzing Book,” CISPA Helmholtz Center for Information Security, 2021.
8. X. Cao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, “Fuzz Testing based Data Augmentation to Improve Robustness of Deep Neural Networks,” 2020 IEEE/ACM 42nd International Conference on Software Engineering, May 23-29, 2020, Seoul, Republic of Korea.

9. C. Michael, G. McGraw, M. Schatz, C. Walton, “Genetic Algorithms for Dynamic Test Data Generation,” IEEE 0-8186-7961-1/97, IEEE 1997.
10. W. Afzal, R. Torkar, R. Feldt, “A Systematic Review of Search-Based Testing for Non-Functional System Properties,” Information and Software Technology, 51 (2009), pp. 957-976.
11. P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification and Reliability 14 (2) (2004) 105–156.
12. T. Mantere, J. Alander, “Evolutionary Software Engineering, a Review,” Applied Soft Computing, 5 (2005), pp. 315-331.
13. J. McCart, D. Berndt, A. Watkins, “Using Genetic Algorithms for Software Testing: Performance Improvement Techniques,” (2007), AMCIS 2007 Proceedings, 222.
14. A. Schultz, J. Grefenstette, K. De Jong, “Test and Evaluation by Genetic Algorithms,” 0885/9000/93/1000-0009, October 1993, IEEE.
15. R. Abdessalem, S. Nejati, L. Briand, T. Stifter, “Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms,” 2018 ACM/IEEE 40th International Conference on Software Engineering, May 27 – June 3, 2018, Sweden.
16. TASS International. 2017. PreScan simulation of ADAS and active safety. (Aug. 2017).
17. F. Luck, M. Zimmermann, F. Wotawa, M. Nica, “Genetic Algorithm-based Test Parameter Optimization for ADAS System Testing,” 978-1-7281-3927-2/19, 2019 IEEE.
18. A. Gambi, M. Mueller, G. Fraser, “Automatically Testing Self-Driving Cars with Search-Based Procedural Content Generation,” ISSTA 19, July 15-19, 2019, Beijing, China.
19. Kalra, N., and Paddock, S. M. “Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?” Transportation Research Part A: Policy and Practice 94 (Dec. 2016), 182–193.
20. Georgiou, T., and Demiris, Y. Personalised track design in car racing games. In 2016 IEEE Conference on Computational Intelligence and Games (CIG) (Sept. 2016), pp. 1–8.
21. D. Humeniuk, F. Khomh, and G. Antoniol, “A Search-Based Framework for Automatic Generation of Testing Environments for Cyber-Physical Systems,” arXiv:2203.12138v1 [cs.NE] 23 Mar 2022.
22. K. Betts, M. Petty, “Automated Search-Based Robustness Testing for Autonomous Vehicle Software,” Modeling and Simulation in Engineering, Vol. 2016, Article ID 5309348.
23. Z. Saigol, F. Py, K. Rajan, C. McGann, J. Wyatt, and R. Dearden, “Randomized testing for robotic plan execution for autonomous systems,” in Proceedings of the IEEE/OES Autonomous Underwater Vehicles (AUV ’10), pp. 1–9, IEEE, Monterey, Calif, USA, September 2010.
24. https://github.com/autowarefoundation/autoware.universe
25. https://github.com/sukha-cn/turtlesim-ros2
26. K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," in IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182-197, April 2002, doi: 10.1109/4235.996017.
27. K. Aebischer, L. Johnson, A. Watkins, J. Fisher and D. Berndt, "Breeding Software Test Cases for Complex Systems," in 2014 47th Hawaii International Conference on System Sciences, Big Island, Hawaii, 2004.
28. J. McCart, D. Berndt, A. Watkins, “Using Genetic Algorithms for Software Testing: Performance Improvement Techniques,” (2007), AMCIS 2007 Proceedings.