
COSMOS

DevOps for Complex Cyber-physical Systems

**Project Number 957254**

# D5.5 Complete framework of test generation and build schedule tooling
# D5.4 Build schedule tool prototype

**Version 1.0**
**30 June 2023**
**Final**

**Public Distribution**

## Delft University of Technology

**Project Partners:** **Aicas, Delft University of Technology, GMV Skysoft, Intelligentia, Q-media, Siemens, Siemens Healthcare, The Open Group, University of Luxembourg, University of Sannio, Unparallel Innovation, Zurich University of Applied Sciences**

# Project Partner Contact Information


| | |
|---|---|
| **Aicas**<br>James Hunt<br>Emmy-Noether-Strasse 9<br>76131 Karlsruhe<br>Germany<br>Tel: +49 721 663 968 0<br>E-mail: jjh@aicas.com | **Delft University of Technology**<br>Annibale Panichella<br>Van Mourik Broekmanweg 6<br>2628 XE Delft<br>Netherlands<br>Tel: +31 15 27 89306<br>E-mail: a.panichella@tudelft.nl |
| **Intelligentia**<br>Davide De Pasquale<br>Via Del Pomerio 7<br>82100 Benevento<br>Italy<br>Tel: +39 0824 1774728<br>E-mail: davide.depasquale@intelligentia.it | **GMV Skysoft**<br>José Neves<br>Alameda dos Oceanos Nº 115<br>1990-392 Lisbon<br>Portugal<br>Tel. +351 21 382 93 66<br>E-mail: jose.neves@gmv.com |
| **Q-media**<br>Petr Novobilsky<br>Pocernicka 272/96<br>108 00 Prague<br>Czech Republic<br>Tel: +420 296 411 480<br>E-mail: pno@qma.cz | **Siemens**<br>Birthe Boehm<br>Guenther-Scharowsky-Strasse 1<br>91058 Erlangen<br>Germany<br>Tel: +49 9131 70<br>E-mail: birthe.boehm@siemens.com |
| **Siemens Healthineers**<br>David Malgiaritta<br>Siemensstrasse 3<br>91301 Forchheim<br>Germany<br>Tel: +49 9191 180<br>E-mail: david.malgiaritta@siemens-healthineers.com | **The Open Group**<br>Scott Hansen<br>Rond Point Schuman 6, 5th Floor<br>1040 Brussels<br>Belgium<br>Tel: +32 2 675 1136<br>E-mail: s.hansen@opengroup.org |
| **University of Sannio**<br>Massimiliano Di Penta<br>Palazzo ex Poste, Via Traiano<br>I-82100 Benevento<br>Italy<br>Tel: +39 0824 305536<br>E-mail: dipenta@unisannio.it | **University of Luxembourg**<br>Domenico Bianculli<br>29 Avenue J. F. Kennedy<br>L-1855 Luxembourg<br>Luxembourg<br>Tel: +352 46 66 44 5328<br>E-mail: domenico.bianculli@uni.lu |
| **Unparallel Innovation**<br>Bruno Almeida<br>Rua das Lendas Algarvias, Lote 123<br>8500-794 Portimão<br>Portugal<br>Tel: +351 282 485052<br>E-mail: bruno.almeida@unparallel.pt | **Zurich University of Applied Sciences**<br>Sebastiano Panichella<br>Gertrudstrasse 15<br>8401 Winterthur<br>Switzerland<br>Tel: +41 58 934 41 56<br>E-mail: panc@zhaw.ch |

# Document Control

| Version | Status | Date |
|---|---|---|
| 0.1 | Outline | 31 May 2023 |
| 0.2 | Background and WP6 overview completed | 5 June 2023 |
| 0.4 | First full draft | 9 June 2023 |
| 0.5 | Initial internal review and changes (within ZHAW) | 9-12 June 2023 |
| 0.6 | Review from other partners of the consortium | 13-24 June 2023 |
| 0.8 | Further editing draft | 25-27 June 2023 |
| 0.9 | Updates addressing final reviewer comments and final draft release | 28 June 2023 |
| 1.0 | Final QA version for EC delivery | 30 June 2023 |

# Table of Contents

# Executive Summary

This report describes the architecture and tools developed as part of the framework for user-oriented test generation and build- and test-scheduler, to improve CPS behavioral states. The test suites for CPSs usually are expensive and time-consuming tasks. COSMOS targeted innovation is to extend traditional DevOps testing pipelines with the ability to prioritize, reducing test duration, and reduce computational costs. More specifically, COSMOS's focus is to select test cases by combining efforts in carving higher-level test cases and high probability failing test cases.

Key technologies for this task are (1) meta-heuristic search for test case generation, (2) Natural Language Processing to link relationships between test cases (WordNet) , and (3) machine learning.

Given such a background, this report begins by describing the current state-of-the-art and identifying gaps in the literature concerning the following relevant research topics:

- Background information on the current state of Test Case Generation specific for CPSs.
- Background information on the current state of testing computer vision tasks within the CPS domain;
- Summary of main general research on regression testing techniques, focused on test suite minimization efforts;
- Background information on the current state of automated program repair in CPSs;
- Focused related studies and tools on simulation-based testing for CPSs.

The deliverable discusses the general architecture of COSMOS Component regarding the build and test scheduler and test generation for CPSs. Then, it discusses the architecture and tools of developed tools:

- Development of Test Case Generation Tools for Rapid DevOps Iterations;
- Adversarial Example Generation for Vision Components within CPSs;
- Program Repair for Self-driving cars with regression testing;
- Tool development regarding the Build and Test Scheduler within a DevOps pipeline.

Finally, a summary of the tools presented within the COSMOS requirements and future work.

# 1 Introduction

## 1.1 Work Package Overview

In recent years, Cyber-physical Systems (CPSs) became of interest across many industries [44] [119]. The increased adoption of CPS increases the urgency to tackle CPS-specific challenges, as these systems became to play critical roles in our daily lives. Besides transportation, we also encounter them as medical devices [28]. Given the severity of unexpected behaviors these systems can have, assuring the quality of software deployed in CPS is a very critical.

Work package 5 concentrates on improving the process of software quality assurance in the development lifecycle of CPS from three different perspectives:

1. Improving code quality to enhance the performance of CPSs (Task T5.1).
2. Reducing the time and resources required to execute tests, written to validate CPS, in DevOps iterations (Task T5.2).
3. (Task T5.3) Enable user-oriented test generation, with the purpose of improving CPS behavioral states, when dealing with humans (e.g., developers or general people interacting with the system).

The first deliverable (D5.1) focused on the first aspect, with the other previous deliverables (D5.2 and D5.3) took the first steps for the second and third points. Starting with the first deliverable, we performed an empirical study to identify CPS-specific (performance) antipatterns. This was followed by the creation of a handbook of detection approaches (including an automatic detection tool) and proposed solutions. Then, we presented a prototype for test decomposition, a new algorithm to test the lane-keeping assistant (REWOSA) to aid automated test generation for self-driving cars, and the first steps regarding test case prioritization.

This document reports the deliverables D5.4 and D5.5. Within this document, we present our Build and Test Scheduler and Test Generation tools.

## 1.2 Task Overview (T5.2 T5.3)

Ensuring software quality in CPSs is done by running software tests with different criteria and granularity levels. The lower-level test cases validate each component in isolation, followed by the integration between each other. These test cases are fast to run and cost little computation effort. The high-level tests focus on validating the functionality of the whole system. These tests are more expensive to run and will require more time. In order to validate recent code changes, they are required to run often (automatically) [55].

In CPS projects, most test suites contain test cases with simulation or the Hardware-in-the-loop, these high-level tests are resource-intensive and time-consuming tasks.

To address these challenges, we aim to work on techniques that can carve the expensive high-level tests in CPSs into low-level (unit or component integration) test cases. Furthermore, we study different techniques to schedule tests according to a list of optimization aspects.

This task includes the following activities:

- developing automated test generation techniques, which are useful in generating low-level tests (e.g., unit or component integration tests) from high-level tests.
- developing improved testing techniques for vision components in CPS, using a Differential Evolution (DE) approach (with single- and multi-objective variants).
- extend regression testing methods for program repair for self-driving cars and the build- and test-scheduler.

## 1.3 Purpose of This Deliverable

This report describes the framework for build- and test scheduling and test-generation tooling, with the purpose of improving quality assurance for CPS. This report begins by describing the current state-of-the-art and identified gaps in the literature concerning the following relevant research topics:

- Background information on the current state of Test Case Generation specific for CPSs (Section 2.1);
- Background information on the current state of testing computer vision tasks within the CPS domain (Section 2.2);
- Summary of main general research on regression testing techniques, focused on test suite minimization efforts (Section 2.3);
- Background information on the current state of automated program repair in CPSs (Section 2.4);
- Focused related studies and tools on simulation-based testing for CPSs (Section 2.5).

The deliverable discusses the general architecture of COSMOS Component regarding the build and test scheduler and test generation for CPSs. Then, it discusses the architecture and tools of developed tools:

- Development of Test Case Generation Tools for Rapid DevOps Iterations (Section 3.2 and Section 4);
- Adversarial Example Generation for Vision Components within CPSs (Section 3.3 and Section 5);
- Program Repair for Self-driving cars with regression testing (Section 3.4);
- Tool development regarding the Build and Test Scheduler within a DevOps pipeline (Section 3.5).

Finally, an overview of the tools presented connected to the COSMOS requirements and a summary of future work.

# 2 Background and Related Work

This section overviews the current state-of-the-art and identified gaps in the literature, providing important background information for better contextualizing the innovations targeted by COSMOS.

## 2.1 Test Case Generation

In the software-enabled world that we live in, reliable and correct software is crucial [87]. As such, software quality assurance has become a critical asset in the software engineer's toolbox. For example, automated testing in the form of unit tests has become an important ingredient to ensure high-quality software [15]. While the importance of testing is generally acknowledged, writing tests is seen as a tedious and time-consuming task [17, 11]. To relieve developers and/or testers of the burden of writing test cases, the research community has invested in developing and evaluating automatic test generation approaches [6, 13, 58, 61].

Today, tools such as EvoSuite [58] and Randoop [122] generate a test suite starting from Java source code using search-based algorithms or random approach to reach higher coverage [60, 127]. Several empirical studies recently focused on practical usage, the challenges automated test generators face in real life, and the quality of the tests generated [59, 153, 7, 128, 129]. Even though automated unit test generation has made significant progress, generated unit tests are less readable than their human-written counterparts [69]. Almasi et al. have conducted an extensive evaluation of automatically generated unit tests in the financial services domain; they have observed that developers (i) find it difficult to follow the scenario depicted in the test case, (ii) find the test data unclear, and (iii) have difficulties with the meaningfulness of generated assertions [7].

### 2.1.1 Readability and Understandability

Readability and understandability are two similar terms, but they have different meanings. *Readability* entails structural and semantic characteristics that allow developers to understand source code, while *understandability* is defined as the ease with which developers are able to extract information from a program [120].

Researchers have also tried to formulate/design readability metrics. Buse and Weimer [24] built the readability metric of the source code, and a predictive model was developed by Daka et al. [39] to assess the readability of unit tests. It was then applied to EvoSuite to produce more readable tests by including readability as a secondary objective after coverage. Fraser and Arcuri [58] used mutation analysis in order to reduce the number of assertions, as a primary factor impacting the developers' effort in validating and understating what the tests assert for. However, understandability is more qualitative than evaluating with the pre-trained models or simply counting the number of assertions. Throughout this deliverable, we used the term understandability to signify this difference.

### 2.1.2 Naming and summarization

Zhang et al. proposed an Natural Language Processing (NLP)-based technique that automatically generates descriptive names for unit tests based on the common structure and names of tests [176]. Daka et al. used coverage criteria to generate unique names for automatically generated unit tests [40]; Nijkamp et al. adapted this approach to fit test amplification [118]. Roy et al. developed DeepTC-Enhancer, which uses deep learning to automatically generate method-level summaries and rename identifiers for the generated test cases [147]. Panichella et al. proposed TestDescriber, which generates test case summaries automatically [131]; starting from EvoSuite-generated tests, TestDescriber generates a description of the intent of the unit tests. Panichella et al. have established that developers working with the test case descriptions are quicker in resolving bugs indicated by failing tests.

### 2.1.3 Realistic Inputs

Afshan et al. have combined a natural language model with a search-based test generation to improve the readability of generated inputs [5]. Through a user study they have observed that participants are faster at evaluating inputs generated with their language model. Knowledge bases such as DBPedia have been used in some studies to generate realistic inputs; Alonso et al. [8] utilized this approach to generate realistic web APIs, and Wanwarang et al. [167] have used it to test mobile applications. It is important to note that these aforementioned works only provide linguistically realistic data. On the other hand, MICROTESTCARVER can be used to generate actual test data in a variety of dimensions; it can generate test data when it contains complex objects like collections, ad hoc objects created by the developer, as well as mock objects by using tracing information.

### 2.1.4 Capturing and Replaying

The purpose of carving unit tests is automatically extracting a collection of unit tests replicating the calls seen during the system test [49]. Also, it is called "record and replay" because the key idea is to record such calls, and replay them later - collectively or selectively [175].

Record-and-replay approaches have been widely used for crash replication, e.g., in ReCrash [12], ADDA [35], Bugnet [114], and jRapture [157]. In addition, [16] and [25] are recent record-replay techniques that are based on monitoring non-deterministic and hard-to-resolve methods (when using symbolic execution) respectively. The recent work on reproducing context-sensitive crashes of Android applications, MoTiF [67], also falls in the first category of record-replay techniques. The aforementioned techniques rely on program run-time data for automated crash replication. Thus, they record the program execution data in order to use it for identifying the

program states and execution path that led to the program failure. However, monitoring program execution may lead to (i) substantial performance overhead due to software/hardware instrumentation [143, 30, 115], and (ii) privacy violations since the collected execution data may contain sensitive information [30].

## 2.2 Testing Vision Components in CPS

Many systems we encounter in daily life include machine learning components that make automated decisions or inferences based on observed data patterns. Ever since so-called deep Convolutional Neural Networks (CNN) outperformed hand-crafted methods in the 2012 ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [148], deep learning models have become mainstream in computer vision. Furthermore, in many other applied machine learning domains, such as natural language processing and music information retrieval.

Recent advancements in CNN-based neural networks, such as dropout [156] and batch normalization [77], have effectively addressed prominent challenges encountered in training neural networks, such as the curse of dimensionality and the vanishing gradient problem. Consequently, convolutional networks have emerged as dominant models for various computer vision tasks, including image classification [148], object detection [66, 180], and semantic image segmentation [29, 149].

Therefore, pre-trained CNN models like VGG16, VGG19, and ResNet50 have been used in the literature for desiging vision components of self-driving cars, such as traffic light detection [62], image segmentation for driving scenarios [149], training self-driving agents [18], and steering angle prediction [80].

Deep Learning (DL) models have been lauded for yielding high-accuracy predictions and, thus, have become attractive candidates for integration in real-life systems that may be safety-critical (e.g. vision components in self-driving cars). At the same time, they have been criticized for making intolerable and sometimes incomprehensible prediction errors, jeopardizing safety. As has been shown in the machine learning world, they are e.g. inherently vulnerable to so-called adversarial attacks, in which perceptually small changes to input data can cause very different, erroneous model predictions [163, 68].

### 2.2.1 Adversarial Example Generations for DNNs

Adversarial examples have been extensively investigated in the literature, where the idea is to introduce subtle changes in the data (e.g., changing the pixels in a target image) that do not change the ground truth but make a DL model predict the incorrect output. Existing approaches to adversarial example generation can be classified into white-box and black-box methods. White-box approaches [164, 103, 135, 86, 70, 178] require access to the model under test (i.e. the model architecture, neuron weight values, and gradients). Black-box strategies [116, 181, 161, 160, 99], instead, only require access to model inputs and outputs. These approaches are considered more realistic as it reflects what external attackers can obtain [99], e.g., in the case of remote API access.

Therefore, we will adopt a black-box testing approach instead, which purely focuses on modifying a system's input (in our case, an image) to trigger undesired changes in the system's output (in our case, the object classification for the input image). We will employ evolutionary strategies for this; beyond images, these have e.g. been proposed on credit scoring models [65] and speech audio [83, 78].

In literature, various black-box attacks on image recognition Deep Neural Networks (DNNs) have been proposed [116, 181, 99]. Nguyen et al. [116] generate random images that are noise to humans, but are misclassified as actual objects by a DNN. In our case, we will seek more adversarial examples, where input is kept as close to the original as possible (and thus human-recognizable).

Zhou et al. [181] propose a hybrid black-box approach that combines Evolutionary Algorithms (EAs) with the bisection method. The images are mutated by injecting full black or white pixels. Instead, Chan and Cheng [27] introduced a black-box approach that adds Gaussian noise to large portions of the images. Besides, their work

targets object detection models rather than image recognition. In contrast, we investigate the adversarial example generation in a multi-objective variant where both (i) the model misclassification, and (ii) the number of changed pixels are taken into account.

Several works explicitly focused on minimizing perturbations, such that fewer modifications to an image would already lead to different system output. One example of this is the work by Suzuki et al. [161], which proposes a Discrete Cosine Transform-based method for modifying images. While such perturbations parametrically are small, they still will affect many pixels at once. A similar consideration holds for the work by Sun et al. [160], focusing on minimum visibility of the modification from a perceptual perspective, but not explicitly constraining the number of pixels to modify.

On the other end of the extreme, one may search for attacks that modify as few pixels as possible (and as such will naturally not stand out, when compared to the total amount of pixels in an image). For example, Su et al.[159] propose a single-pixel adversarial attack using Differential Evolution (DE), executed against the classical CIFAR-10[1][89] and ImageNet object classification datasets. Comparing the results on these two datasets, a high success rate is reported for CIFAR-10, but this success rate is much lower for ImageNet, where single-pixel attacks mainly succeed in situations where the original classification of the image was already quite low. This may have to do with the difference in search space; the test images in CIFAR-10 are much smaller ($32 \times 32 = 1024$ pixels) than those in ImageNet ($224 \times 224 = 50,176$ pixels).

A stronger, yet compact attack is proposed by Lin et al. [99], who combined DE [158] and the Fast Gradient Sign Method [112] for black-box adversarial sample generation. Executing a single-objective attack called `Black-box Momentum Iterative Fast Gradient Sign Method (BMI-FGSM)`, to generate an efficient and effective perturbation that is similar to the benign input. Their approach utilizes double-step size and candidate reuse whilst approximating the gradient direction. An initial gradient sign population is generated using DE. The input is then gradually modified using gradient sign approximation until an adversarial example is created that is visibly the same as the original input, but now classified as something different. Lin et al. [99] showed that `BMI-FGSM` successfully generates adversarial examples for large models, outperforming other state-of-the-art white-box and black-box approaches.

While Lin et al. [99] showed that black-box approaches based on EAs can be very competitive with their white-box alternatives, existing approaches have various drawbacks. First, `BMI-FGSM` requires a large number of iterations (in the order of thousands) and population size (hundreds of individuals). In other words, attackers need to query the model under attack many times, increasing the chances of detection. Second, `BMI-FGSM` combines multiple techniques, making its implementation less trivial and introducing more hyper-parameters to tune. Finally, the generated attacks are not minimal, i.e., the prediction flip is achievable but requires changing all pixels in the original image (or seed).

In this deliverable, we introduce a simpler approach purely based on DE. Furthermore, we introduce both a single- and multi-objective variant of our approach. The former focuses on flipping the model prediction, while the latter considers an additional objective that aims to directly minimize the number of modified pixels. As our results will show, our approach requires a much lower number of model queries and introduces fewer image changes compared to `BMI-FGSM`.

## 2.3 Regression Testing

Regression testing is the process of retesting a software project to evaluate whether changes in the production code have any unintended effects on the unchanged portions [173]. The ideal approach to regression testing involves executing the entire test suite within a DevOps pipeline [166]. However, this strategy may not be feasible for systems that require extensive resources, such as build servers, or for test suites that are very expensive, such as simulation-based tests. To address this issue, researchers in the software engineering community have proposed different techniques to manage the cost of regression testing. These techniques

[1] `https://www.tensorflow.org/datasets/catalog/cifar10`

include removing redundant tests through test suite minimization [145], selecting a subset of tests for execution through test case selection [31], and prioritizing test cases to detect regression faults earlier through test case prioritization [146].

Test case prioritization is particularly challenging because testers are unaware of the fault detection capability prior to test execution. To overcome this challenge, surrogate metrics such as code coverage have been used in the literature to determine the order of test case execution. These surrogates are correlated in some way with the rate of fault detection [173]. Surrogate metrics can be classified into two main categories: white-box metrics and black-box metrics [74].

Among white-box metrics, code coverage —such as branch coverage [144], statement coverage [52], block coverage [46], and function or method coverage [53]— is the most widely used surrogate. Other prioritization criteria have also been explored, such as interactions [22, 37], requirement coverage [155], statement and branch diversity [182, 26], and additional spanning statement and branches [106].

In addition to white-box metrics, black-box metrics have also been investigated. For example, Bryce et al. proposed the "t-wise" approach, which considers the maximum interactions between t-model inputs [21, 23] [136]. Other approaches considered input diversity calculated using Normalized Compression Distance (NCD) [141], Jaccard distance [75, 76], and Levenshtein distance [73, 92] between inputs. Henard et al. also considered the number of killed model mutants [76, 132]. A comparison between white-box and black-box criteria for test case prioritization was conducted by Henard et al. [74], showing that there is little difference between the two categories. In the context of Cyber-Physical systems (CPSs), Birchelr et al. [19] introduced a novel-diversity metric for simulation-based that involve driving scenarios that are specific to the automotive domain. Birchler et al. utilize these road features to quantify the differences between different test cases. By considering the specific road features and utilizing the Euclidean distance, this metric offers insights into the effectiveness of the test suite in capturing different aspects of the driving scenarios. It enables researchers and practitioners to evaluate and optimize simulation-based test suites.

The aforementioned works optimize for a single prioritization criterion and use a greedy algorithm to sort the test cases based on the chosen criterion. Two main greedy strategies can be applied: (i) the "total" strategy, which selects test cases based on the number of code elements they cover, and (ii) the "additional" strategy, which iteratively selects the test case that covers the maximum number of code elements not covered by previously selected test cases [71] [177]. Recently, a hybrid approach proposed by Hao et al. [71] and Zhang et al. [177] combined the "total" and "additional" coverage criteria, demonstrating that their combination can be more effective than the individual components. Greedy algorithms have also been used to combine multiple testing criteria, such as code coverage and cost. For example, Elbaum et al. [50] and Malishevsky et al. [104] considered code coverage and execution cost, customizing the additional greedy algorithm to condense the two objectives into a single function (coverage per unit cost) for maximization. Three-objective greedy algorithms have also been employed to combine statement coverage, historical fault coverage, and execution cost [173, 123].

In addition to greedy algorithms, meta-heuristics have been used as alternative search algorithms for test case prioritization. Li et al. [97] compared the additional greedy algorithm, hill climbing, and genetic algorithms for code coverage-based test case prioritization. They developed fitness functions such as APBC (Average Percentage Block Coverage), APDC (Average Percentage Decision Coverage), or APSC (Average Percentage Statement Coverage) to enable the application of meta-heuristics.

When multiple functions (or objectives) are considered for prioritizing tests, meta-heuristics rely on the concept of Pareto optimality [57]. One test case permutation, $\tau_A$, is considered better than another permutation, $\tau_B$, (and vice versa) if and only if $\tau_A$ outperforms $\tau_B$ in at least one objective while not being worse in all other objectives.

Subsequent works have emphasized that permutation-based genetic algorithms for test case prioritization should consider multiple testing criteria due to the multi-objective nature of the problem. For instance, Li et al. [96] proposed a two-objective permutation-based genetic algorithm to optimize APSC and the execution cost required to achieve maximum statement coverage (cumulative cost). They employed a multi-objective genetic algorithm,

specifically NSGA-II, to find a set of Pareto optimal test case orderings that represent optimal trade-offs between the two corresponding AUC-based criteria.

Islam et al. [79] and Marchetto et al. [105] utilized NSGA-II to discover Pareto optimal test case orderings that represent trade-offs among three different AUC-based criteria: cumulative code coverage, cumulative requirement coverage, and cumulative execution cost. Similarly, Epitropakis et al. [54] compared greedy algorithms, MOEAs (NSGA-II and TAEA), and hybrid algorithms. They considered various AUC-based fault surrogates, including statement coverage (APSC), $\Delta$-coverage (APDC), and past fault coverage (APPFD). The study demonstrated that three-objective MOEAs and hybrid algorithms are capable of producing more effective solutions compared to additional greedy algorithms based on a single AUC metric.

In this deliverable, we consider regression testing methods in (1) program repair for self-driving cars, and (2) for the build and test scheduler.

## 2.4 Automated Program Repair

In the past years, several automated program repair (APR) strategies have been proposed in the literature to fix bugs in individual software programs without any human intervention [168, 85, 138, 63]. Both academia and industry have extensively studied APR techniques. Examples of well-known APR techniques include Genetic Programming (GP) [90] random search [138], and symbolic execution [108].

These techniques require a faulty program and a test suite comprising passing test cases representing the desired program behavior and failing test cases that expose the fault to be addressed. The process of fault repair involves iteratively identifying faulty statements in the code (known as fault localization [171, 81]), automatically modifying the identified statements (patch generation), and checking if the patched code passes all the test cases (patch validation).

Therefore, an APR system consists of three main phases. First is *fault localization*, where off-the-shelf techniques are used to diagnose suspicious code elements (e.g., statements) prior to the repair process. Second is *patch generation*, where repair operations are applied to suspicious locations based on a ranking list. Each modified program version is considered a candidate patch. Lastly, in the *patch validation* phase, each candidate patch is tested against the test suite until a patch that successfully passes all tests is found (known as a plausible patch). Existing APR systems often terminate after finding one plausible patch or reaching the time budget.

The various APR techniques differ on the underline heuristics used to generate candidate patches. Search-based program repair, exemplified by the work of Le Goues et al. [90] in GenProg, focuses on genetic search for repairing C programs. One significant contribution was representing patches as changes (addition, removal, or replacement) to existing statements. Genetic search involves the mutation, creation, and combination of chromosomes, which are the fundamental units of the search. In APR, a chromosome represents a list of such changes rather than the entire program, making the approach lightweight. This approach relies on the Redundancy Assumption [107], which assumes that the required statements for the fix already exist and only need minor adjustments such as using the correct variable or adding a null-check. The validity of this assumption has been confirmed by Martinez et al. [107], who demonstrated its widespread applicability in inspected repositories.

Other APR techniques uses constraint solving [48, 172] (e.g., SMT solvers) to generates patches for conditional expressions, such as condition code with arithmetic and first-order logic operators. Instead, template-based APR [88, 100, 101] involves the design of predefined patterns to guide patch generation. For example, TBar [101] constructs a comprehensive pattern pool by integrating patterns from existing template-based APR systems. Finally, Learning-based APR [32, 95]adopts machine/deep learning techniques to generate patches based on existing code corpora.

In the context of CPS, Abdessalem et al. [4] present a novel automated repair technique designed to tackle the challenges of resolving undesired feature interaction failures in automated driving systems (ADS). Unlike previous approaches that focused on fixing bugs in individual software programs, this research addresses failures

Figure 1: An intersection scenario in CARLA.

that arise at the system-level due to undesired interactions among different components or functions within complex systems such as autonomous cars. The proposed repair strategy encompasses several key aspects, including fault localization across multiple lines of code, simultaneous resolution of multiple interaction failures caused by independent faults, scalability from the unit-level to the system-level, and prioritized resolution based on the severity of failures. This work contributes to the advancement of automated repair techniques for system-level failures in complex autonomous systems, ultimately enhancing their safety and reliability.

Existing research has highlighted the time-consuming nature of patch validation [102] due to two main reasons. Firstly, the generation of a large number of patches adds to the overall time required for validation. Secondly, each patch necessitates the execution of non-trivial time-consuming original tests for validation. In addition to the substantial number of generated patches, the execution of each patch against the original tests incurs significant costs. In fact, the generate-and-validate procedure in APR bears resemblance to regression testing, wherein patches represent modifications to the original flawed program and each patch must be validated using the existing test suite. Therefore, regression testing techniques can help reducing the cost of the patch validation face, as it has the potential to enhance repair efficiency by selectively executing only the tests affected by the patch.

In the context of CPS, simulation-based tests are particular expensive as they require additional resources (e.g., GPU servers) and longer runtime (the time to simulate a test scenario), increasing the potential benefits of regression testing techniques for program repair.

## 2.5 Simulation-based Testing for CPS

Autonomous driving simulators provide a controlled virtual environment for testing autonomous driving systems. These simulators offer a range of capabilities, including the simulation of weather conditions, traffic participants, and sensor behavior. Simulators can be split in roughly three categories: realistic, semi-realistic and abstracted. In this section, we will review several state-of-the-art open-source simulators that are commonly used in research and industry.

### 2.5.1 Realistic Simulators

Realistic simulators, such as CARLA, use digital assets to create a realistic environment, which looks similar to the real world, and as such can be used to test autonomous driving systems in a realistic environment.

**CARLA** [47], developed by the Computer Vision Center at the Autonomous University of Barcelona, is a highly realistic simulator based on Unreal Engine. It is widely used by research groups and companies such as NVIDIA, Intel, and Toyota. CARLA features digital assets to create a realistic environment, and can simulate the behavior of pedestrians, vehicles, and traffic lights. It also allows for the simulation of weather conditions

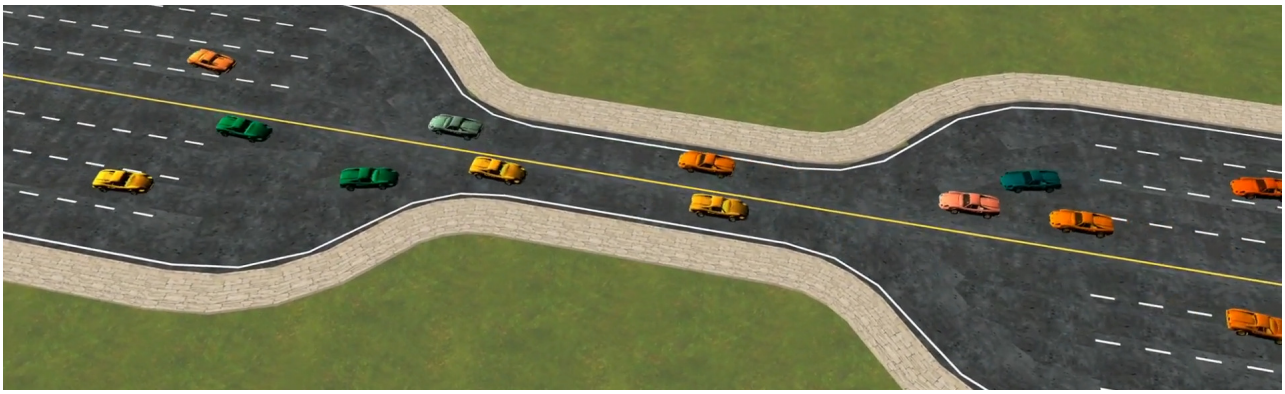


Figure 2: A bottleneck scenario in MetaDrive.

such as rain, fog, and snow, as well as the behavior of sensors used in autonomous driving systems such as cameras, lidar, and radar. Custom scenarios can be created using the built-in map editor, enabling a wide range of traffic scenarios to be tested. See figure 1 for an example of an intersection scenario in CARLA.

**AirSim** [152], developed by Microsoft AI Research, is a multi-purpose simulator for the simulation of different types of vehicles, with a focus on aerial autonomy. It is based on Unreal Engine and Unity and utilizes digital assets to create a realistic environment. Unlike CARLA, it does not include the ability to simulate the behavior of other traffic participants.

**BeamNG.tech** [14], developed by BeamNG, is a simulation framework that supports the testing of autonomous driving systems in the game BeamNG.drive. It is a physics-based driving simulator that allows for the simulation of autonomous driving systems in realistic environments with other simulated traffic participants. It also supports the simulation of sensors such as cameras, lidar, and radar.

**SVL Simulator** [142] is a standalone simulator developed by LG Electronics R&D Lab. Based on Unity, it allows for the simulation of autonomous driving systems in realistic 3D environments. A pre-built environment is provided, but custom environments can be created using assets supported by Unity. It offers the option to simulate random traffic scenarios with various traffic participants such as cars, trucks, buses, and pedestrians, or to define the behavior of traffic participants in a custom scenario. Weather effects such as rain, fog, and snow can also be added to the simulation. A wide range of sensors are supported, including cameras, lidar, radar, GPS, IMU, and odometry.

**AutoDrome** [110] is a simulation framework developed by volunteers that supports the testing of autonomous driving systems in the games Euro Truck Simulator 2 (ETS2) and American Truck Simulator (ATS). These games, which allow players to drive trucks across Europe and the United States, provide large realistic driving environments with a range of weather conditions and traffic participants. AutoDrome, which is focused on reinforcement-learning-based autonomous driving systems, can also make use of the games' built-in map editor to create custom scenarios.

**DeepDrive** [139] is a simulator developed by Craig Quiter, and is based on Unreal Engine. It is designed to be hardware agnostic, enabling the training of autonomous driving systems on a variety of sensors, cars, and environments. DeepDrive uses open-source digital assets to create realistic environments, but does not support the simulation of other traffic participants or weather effects. It includes a Python API for the integration of autonomous driving algorithms and is highly modular, allowing for easy integration with other software and hardware.

### 2.5.2 Semi-Realistic Simulators

Semi-realistic simulators, such as TORCS, use digital assets to create a semi-realistic environment, but lacks the realism to mimic the appearance of the real world.

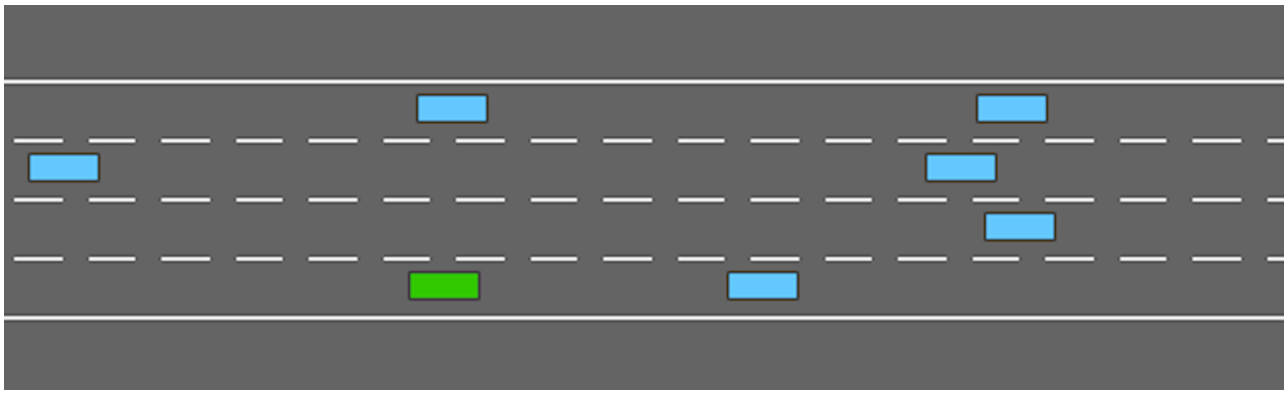


Figure 3: A highway scenario in highway-env.

**MetaDrive** [94], developed by the Centre for Perceptual and Interactive Intelligence, is a simulator that allows for the creation of infinite traffic scenarios through procedural generation. These scenarios include roundabouts, intersections, tollgates, bottlenecks, parking lots, and stretches of road with randomly generated traffic. While it lacks realistic visual assets, simulated pedestrians, traffic lights, and signs, MetaDrive does support a range of sensors such as cameras, lidar, and radar. Additionally, its lightweight design allows it to run scenarios faster than more realistic simulators such as CARLA and AirSim. See Figure 2 for a bottleneck scenario in MetaDrive.

**TORCS** [170], The Open Racing Car Simulator, is a customizable car racing simulator developed by Eric Espié and Christophe Guionneau, based on the OpenGL Utility Toolkit (GLUT). It allows users to create custom tracks and cars, or utilize a selection of pre-built tracks and cars. TORCS was popular in early autonomous driving research, with an autonomous driving championship held at conferences GECCO and SIG, as well as an annual championship hosted by the TORCS team. However, researchers have since moved away from TORCS due to its lack of realism compared to newer simulators such as CARLA.

**Duckietown** [133], created by the Duckietown Foundation, is a simulator for training and developing autonomous driving systems called Duckiebots. It places the Duckiebot in a loop of roads with turns, intersections, obstacles, pedestrians, and other Duckiebots. A variety of scenarios are available, each presenting different challenges for the Duckiebot, such as static obstacles and dynamic obstacles such as pedestrians and other Duckiebots. The simulator is based on a real environment and real Duckiebots, allowing the Duckiebot to be tested in the real world after training in the simulator.

### 2.5.3 Abstracted Simulators

Abstracted simulators, such as highway-env, do not use digital assets to create a realistic environment, instead, the environment is abstracted to a two-dimensional representation and is more focused on testing the decision-making of autonomous driving systems.

**SUMO** [121], The Simulation of Urban Mobility, is a traffic simulation package developed by the Institute of Transportation Systems at the German Aerospace Center. It is primarily used for traffic forecasting in urban environments, but is also utilized by some autonomous driving researchers for testing custom traffic scenarios. The environment is abstracted, with no realistic digital assets, and is represented as a lane-level road network with 2D traffic participant shapes. It includes the simulation of pedestrian, vehicle, and traffic light behavior, but does not support the simulation of weather conditions or sensors.

**Highway-env** [93] is an abstracted autonomous driving simulator. The environment is represented as a lane-level road network with 2D vehicle shapes. Its focus is on the development and testing of autonomous driving system decision-making modules. It offers a range of traffic scenarios, including highways, roundabouts, intersections, and parking lots, with randomly generated traffic participants whose behavior is based on realistic driving models. Custom behavior can also be implemented. Due to its simplified nature and decision-making

focus, highway-env does not support the simulation of weather conditions or sensors. See Figure 3 for a highway scenario in highway-env.

In conclusion, autonomous driving simulators provide a range of capabilities and features, including the simulation of weather conditions, traffic participants, and sensor behavior. The simulators discussed in this chapter, CARLA, AirSim, BeamNG.tech, SVL Simulator, AutoDrome, DeepDrive, MetaDrive, TORCS, Duckietown, SUMO, and highway-env, are, or have been, widely used in research and industry for the testing and development of autonomous driving systems.

# 3 COSMOS Architecture and Tools

## 3.1 Architecture of COSMOS - WP5

This deliverable focuses on Task T5.2, namely the *Development of Test Case Generation Tools for Rapid DevOps Iterations*. To achieve this task, we present four main contributions:

- An unit test carver that generates unit-level test cases starting from system-level tests. The generated unit tests are cheaper and, therefore, can be executed in more time compared to the system-level counterparts. This helps to provide faster feedback for the developers.
- Errors in CPS can also be due to errors in the vision components and the underline machine learning models in particular. We introduce a novel tool that identifies weaknesses (undesired miss-classification) in convolutional neural network (CNN) models that identify objects (e.g., traffic signs) in driving scenarios for CPSs.
- To help CPS developers debug and fix defective code in CPS, we present an automated program repair for self-driving cars. Our approach combines evolutionary algorithms with regression testing techniques to speed up the process of generating patches.
- Throughout the COSMOS projects, we have presented various techniques to generate tests for CSP ad different granularity levels, from uni-level to simulation-based tests. With more tests to run in CI/CD pipelines, scheduling which test to run during the build process becomes more critical. It requires to consider the test inter-dependencies and the resource they require for execution.

The prototype tools are described in detail in the next subsections.

## 3.2 Carving Unit-Level Test Cases

In this Section, we present an approach that carves information from end-to-end (E2E) tests to generate meaningful unit tests. Resting on the assumption that E2E tests are available for the system, during carving we extract the execution trace from a running E2E test, including the order of calls and the inputs. Using that information, we gather scenarios that are meaningful in the domain, and (parameter) values to instantiate objects and pass to method calls. The premise of generating unit tests from E2E tests, is that due to unit tests being more fine-grained, the bug localization becomes easier.

### 3.2.1 Motivating example

Consider Listing 1 and Listing 2 which depict respectively a manually written JUnit test and an EvoSuite-generated JUnit test. When we compare the scenarios of both of these test cases, the manually written one is

```
1| @Test
2| public void shouldCallWeatherService() {
3|     // Arrange
4|     var expectedResponse = new WeatherResponse("raining", "a light drizzle");
5|     /** Mocking Weather Service */
6|     given(restTemplate.getForObject("Weather API", WeatherResponse.class))
7|         .willReturn(expectedResponse);
8|     // Act
9|     var actualResponse = subject.fetchWeather();
10|    // Assert
11|    assertThat(actualResponse, is(Optional.of(expectedResponse)));
12| }
```

Listing 1: An example of a manually written unit test

```
1| @Test(timeout = 4000)
2| public void testEqualsWithNull() throws Throwable {
3|     WeatherResponse weatherResponse0 = new WeatherResponse("S:q$ZHC!0J3", "&_>!@K");
4|     boolean boolean0 = weatherResponse0.equals((Object) null);
5|     assertFalse(boolean0);
6|     assertEquals("WeatherResponse{weather=[Weather{main='S:q$ZHC!0J3',
↪  description='&_>!@K'}]}",  1| weatherResponse0.toString());
7| }
```

Listing 2: An example of a unit test generated by EvoSuite

seemingly easier to understand. For example, when we zoom in on line 4 of Listing 1 and line 3 of Listing 2, we can easily grasp that in the former case we are constructing an object to represent *rainy weather*, while the latter case does not correspond to an actual weather situation (*S:q$ZHC!0J3*). Moreover, in Listing 1 a REST API response is mocked, which checks if the weather that is returned by the mock corresponds to an expected weather situation. In the case of the generated test in Listing 2, the test checks whether the object is null, and checks the result of the toString() method, albeit with constants that do not make sense in the domain.

The overview of the proposed approach is illustrated in Figure 4. The MICROTESTCARVER framework takes an E2E test, which can either be a manual or scripted E2E test, and it generates unit tests in three phases: *instrumenting*, *parsing*, and *generating unit tests*. In the first step, it instruments the E2E tests and records information, such as calls and inputs data; in the second step, it parses this information, and finally, it uses a template-based approach to generate unit tests based on the parsed data.

Our approach "carves scenarios" from the E2E tests to reproduce (smaller elements of) them in the form of unit tests. Our hypothesis is that these higher-level test scenarios embedded in the E2E tests and containing concrete values, can lead to easier-to-understand unit test scenarios.

Our approach is implemented in a Java-based prototype called *MicroTestCarver*. Our tool focuses on generating tests for public methods, which is similar to how a developer would produce unit tests for their production code.

Next, we describe each phase of our approach in detail.

### 3.2.2 E2E tests instrumentation

We utilize BTrace [1] as the basis for our instrumentation tool. We have created a fork of BTrace and developed some additional functionality for our carving approach. In particular, we now collect detailed information of types, and we also collect and serialize information on fields, arguments, and callbacks in a uniform manner. In

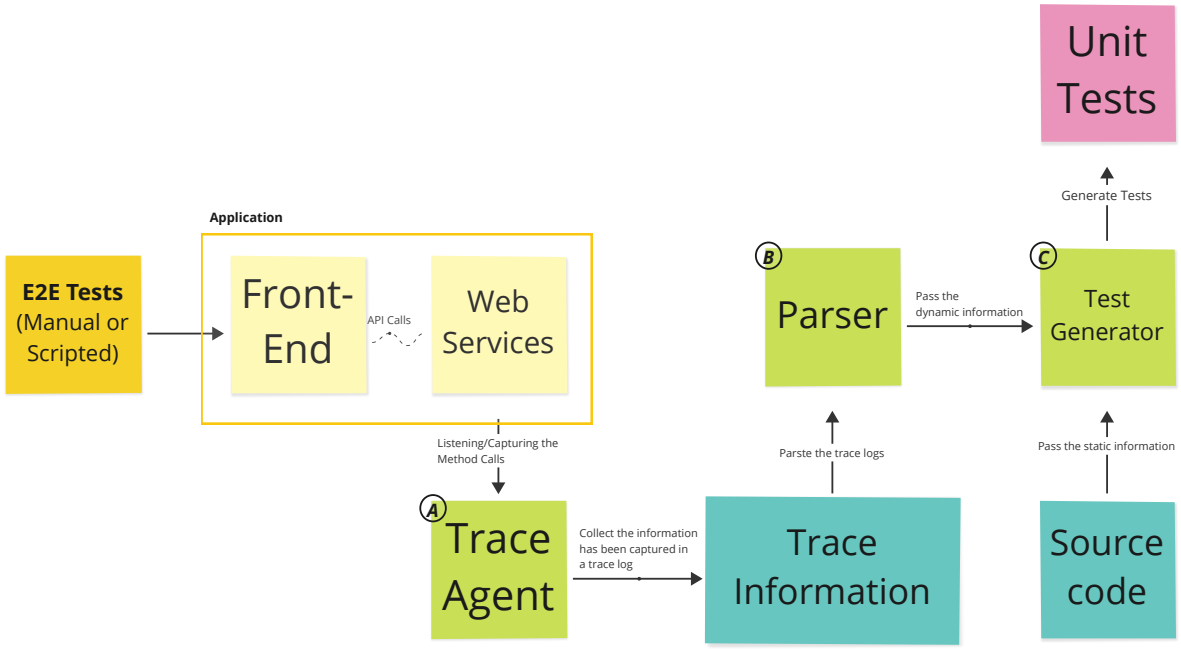



Figure 4: Overview of the carving framework

addition, we chose to use XStream [2] in the modified Btrace because it is an advanced and robust serialization library for Java, and it can handle complex custom objects effectively. The modified version of BTrace is available in the replication package [45].

A high-level overview of our tracing approach is depicted in Algorithm 1.

In this algorithm, the recorded information will be written into a trace log, and all objects observed in lines 11, 12, 21 that are not primitive and can be serialized will be serialized into a serialized object pool. An example of a trace log is shown in Listing 3.

Basically, a trace log is a graph, in which we identify two types of methods: a *NodeMethod*, which is a method that calls other methods of interest in the test generation process, and a *LeafMethod*, which could still call other methods, but those methods are no longer of interest in the test generation process (and could for example be mocked). An example of this graph is shown in Figure 5. We will now explain both concepts.

*LeafMethod.* A LeafMethod (LM) is a called method that does not have a callee that the trace script is watching. An LM refers to a method that is outside the package being watched; it may be a method in a third-party library. Each LM has a *name*, a *type*, a set of *arguments*, a state of the object (i.e., attribute values) before executing the method body, and if the return type is not void, a *return* value. In lines 16–27 of Listing 3, an LM with the name of `RestTemplate` is illustrated: its arguments are a string, class, and an array of objects. In lines 18–27, its callback is shown, which is a container object (Optional) of type `WeatherResponse`; it is serialized in a serialized object pool with the name `1e23ee0e`.

*NodeMethod.* A NodeMethod (NM) is a method that is called within the scope of tracing, and a test will be generated according to this method during the phase of test generation. Each NM, in addition to all the properties of an LM, includes a set of methods called in it; these methods can be a LeafMethod or a NodeMethod.

On line 1 of Listing 3, `fetchWeather` is a NM with no argument and has several fields, such as `city` and `restTemplate`. `RestTemplate.getForObject()`, which is an LeafMethod, is also called in this NM and finally returns a `WeatherResponse` on lines 28–32.

**Algorithm 1:** Tracing Algorithm

```
1  Function Instrumentation:
       Input: AUT: app under test, pName: package name pattern, mName: method name pattern, default
              value is "*"
       Output: t: trace graph
       Init : v ← ∅
2      while AUT is running do
3          foreach m entered in AUT do
4              if m.name ∈ mName and m.clazzName ∈ pName then
5                  v ← createNM(m);
6                  t.add(v);
7  Function createNM (root) :
       Init : nm ← null
8      nm.fields ← captureObjects(root.fields);
9      nm.args ← captureObjects(root.args);
10     nm.callees ← ∅;
11     nm.children ← ∅;
12     foreach m called in root do
13         nm.callees.add(m);
14         if m.clazzName ∈ pName then
15             nm.children.add(createNM (m) );
16     nm.return ← captureObject(root.callback);
17     return nm;
```

The concept of trace log is illustrated in Figure 5, in which three NMs are highlighted that are called in the `ExampleController` class. $NM_2$ is `fetchWeather()` method that is explained in Listing 3, and has `restTemplate` LM. The leaves in a trace log are an LM or a NM without a method call.

An important element to consider is what constitutes a *LeafMethod* and a *NodeMethod*. During the instrumentation phase, we focus on a particular packageName; we watch the classes inside the package and generate tests for them. The classes outside of the package, but are called in methods that belong to classes inside the package, are labeled LeafMethods.

### 3.2.3 Parsing

As we aim to create building blocks for the test generation phase, we parse the trace log and deserialize the serialized objects with the aim of reconstructing the trace data into actual runtime objects. We do so by unifying all elements that were recorded, i.e., the trace log itself and the serialized objects. We create a set of classes that group together NodeMethods based on the classname in the fully qualified path. For example, in Fig. 5, `ExampleController` will be re-initiated based on the NMs whose class name is `ExampleController`. In order to create a class, its arguments, fields, and methods will be assigned based on its NMs, and its constructor method is a NodeMethod with `<init>` name.

### 3.2.4 Generating Unit Tests

We generate test cases based on the classes created in the Parsing phase and also the analysis of the existing source code. We used *JUnit* for the test framework and *Mockito* for mocking, and we utilized their annotations

```
example.weather.WeatherClient.fetchWeatherNodeMethodorange:{
 Args: []
 Fields: [{
   name: city,
   type: java.lang.String,
   object: "Hamburg,de",
   ...
 }, {
   name: restTemplate,
   type: org.springframework.web.client.RestTemplate,
   isPrimitive: false,
   isInterface: true,
   object: org.springframework.web.client.RestTemplate,
   fields: [...],
 }, ...]

 virtual java.lang.Object
org.springframework.web.client.RestTemplate#getForObject
(java.lang.String, java.lang.Class, java.lang.Object[])
[org.springframework.web.client.RestTemplate@4dd4965a]
 Args: [...]
 Callback: {
   ...
     hash: 1e23ee0e,
     type: java.util.Optional,                    LeafMethod
     serialized: true,
     object: Optional[WeatherResponse{weather=
     [Weather{main='Clear', description='clear sky'}
     ]}],
     fields: [class java.lang.Object value=
     example.weather.WeatherResponse@3e1e8fc, ],
 }

 Return: {
   ...
     type: java.util.Optional,
     object: Optional[WeatherResponse{weather=[Weather{main='Clear', description='clear sky'}]}],
     fields: [class java.lang.Object value=example.weather.WeatherResponse@3e1e8fc, ],
 }
}
```

Listing 3: An example of trace log

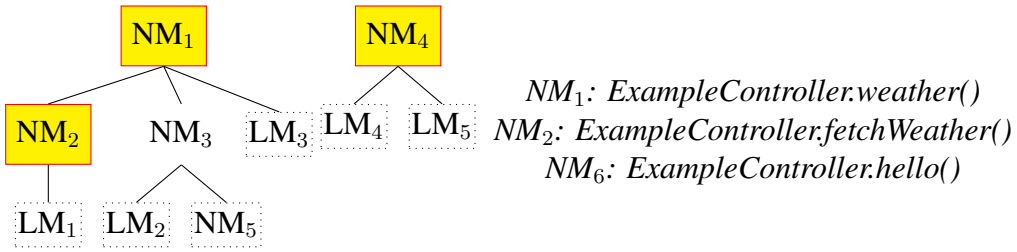



Figure 5: An example of trace log with their classes

to improve legibility. In addition, we used Khorkiov's guidelines [84], which contain a set of best practices and recommendations for writing unit tests, in order to ensure the carved tests have a clear and readable structure. We will first explain how we reproduce objects in a test, and then explain how different components of a test (*fields*, *set-up*, and *test method body*) will be produced.

*Reproducing Objects*. In order to accurately reproduce objects in various parts of a test, such as setting values, invoking methods, and making assertions, besides performing dynamic analysis, it is crucial to perform static analysis. To accomplish this, we combined *Spoon* [134] and Java reflection. As a result of analyzing the source code with Spoon, we are able to identify the appropriate constructor that can recreate the parsed object and set its fields. We have implemented three strategies in order to reproduce the parsed objects: *Unmarshalling*, *ToString*, and *Guessing*. The *Unmarshalling* strategy is used when the object is deserialized, and it will reveal how to recreate the runtime object. We already have implemented *unmarshallers* for various types: Primitive types, String, Collection, Map, Optional, Enum, Date, Locale, and custom objects. The custom object *unmarshaller* is used as a fallback, replicating an object based on setting its fields. This strategy reproduces `WeatherResponse` in lines 18 and 24 of Listing 4. The *guessing* strategy is used when the object is not deserialized, and we are trying to reproduce it like the custom unmarshaller by setting its fields, with this difference that its fields come from the trace log, not a runtime object. The *toString* strategy is used for the assertion section when the object is not deserialized, but it overrides the `toString()` method.

*Fields*. The fields of a test class include the initialization of the CUT (class under test) and setting up a mocked object. As we want to make it very clear what the CUT is, we name that field *subject*. The objects that are annotated as `@Mock` contain methods that are called in the test methods. This is illustrated in Listing 4 where the subject is the instantiation of `WeatherClient`, the class under test, and `restTemplate` is the object that should be mocked since its method, `getForObject`, is called in `fetchWeather` (line 21).

*Set-up*. Every class has a set-up part containing a common initialization that is repeated in all methods; they are also known as test fixtures [84]. The heuristic used for setting the default values of the fields is to select a value that is repeated most often in the NMs of a class. By doing so, a significant amount of duplication can be avoided. For example, in Listing 4, lines 11-12, the subject object and the city field are initiated and set; and in Listing 5, the `id` with the value of `1` is repeated most, helping to reduce the number of lines.

*Test Method*. In order to have a simple and uniform structure, we use the Arrange-Act-Assert (AAA) pattern. Additionally, this pattern makes test cases easier to read and understand. We will discuss the elements of a test method in the following: method name, arrange, action, and assertion.

*Test Name:* When developers navigate among sets of unit tests, the names of the tests aid them in understanding the purpose and scenario of the tests. While there are complex approaches [40, 147] to naming the methods, we used a simple heuristic approach for creating unique test cases based on the inputs and output of a NodeMethod. If there is only one NodeMethod for the MUT (method under test), the test name will be [MUT]Test. If there are multiple NodeMethods, the test name will be a combination of the types and values of the inputs and output. This name is unique since if all conditions were the same, a duplicate test would be recognized. The test name pattern is:

$$([MUT][Where[Inputs]^*][Returning[Output]]?\,Test)$$

```
1  public class WeatherClientTest {
2
3    private WeatherClient subject;
4
5    @Mock                                          } Parameters
6    private RestTemplate restTemplate;
7
8    @BeforeEach
9    public void setUp() throws Exception {
10     MockitoAnnotations.openMocks(this);
11     subject = new WeatherClient();               } Set-Up
12     subject.city = "Hamburg,de";
13   }
14
15   @Test
16   public void fetchWeatherTest(){
17     // Arrange
18     given(restTemplate.getForObject("Weather API", org.springframework.web.client.RestTemplate))
           .willReturn(new WeatherResponse("Clouds", "few clouds"));
19
20     // Act                                         } Test Method
21     Optional fetchWeather = subject.fetchWeather();
22
23     // Assert
24     assertThat(fetchWeather, is(Optional.of(new WeatherResponse("Clouds", "few clouds"))));
25   }
26 }
```

Listing 4: An example of a carved unit test

*Arrange:* The arrange section involves bringing the subject and its dependencies into the desired state as well as mocking any other methods in the MUT that need to be called.

By first determining which objects to mock, which is done in the fields section, we can mock the methods based on the NM's callees (LeafMethods) and their callbacks. As shown in Listing 4, line 18, the behavior of `restTemplate.getForObject` is mocked, which retrieves the weather from the weather API.

Additionally, if the value of the fields in the NodeMethod is different from the one set in the set-up, it will be reset in this section. In Listing 4 the fields are set in the set-up, and Listing 5 shows the name field as a private method and the value in the test method differs from set-up, while the value of `id` is the same between set-up and `getNameTest`.

*Act:* This section contains the method called on the CUT, input values are passed to them, and output values are captured. The NM type will be assigned to the output type. For example, fetchWeather() has been called in Listing 4, and getName() in Listing 5.

*Assert:* This section contains the verification of the result of the return value or the final state of the subject with the expected results. We use the *assertThat* assertion, which compares the output of the MUT and the expected result captured in the NM. Listings 4 and 5 contain assertion statements at lines 24 and 14, respectively.

## 3.3 Adversarial Example Generations for the Vision Components of CPS

Without loss of generality, an image classifier $f$ is a mathematical function/model $f : I \longrightarrow L \times \mathbb{R}^n$, which takes as input an image $i \in I$ and returns a label $l \in L$ and a confidence vector $conf \in \mathbb{R}^n$, which contains the probabilities associated with all labels $l \in L$ in descending order. The first element $conf_1$ is the probability

```
    ...
 1| @BeforeEach
 2| public void setUp() throws Exception {
 3|     ...
 4|     subject.setId(1);
 5|     subject.setName('Basil');
 6| }
 7| @Test
 8| public void getNameTest(){
 9|     // Arrange
10|     subject.setName('radiology');
11|     // Act
12|     String getName = subject.getName();
13|     // Assert
14|     assertThat(getName, is("Mike"));
15| }
```

Listing 5: An example of setting fields of the subject and call the MUT

associated with the most likely (predicted) label $l$, while the remaining entries in $conf_2 \ldots conf_n$ are related to the other possible labels $\in L$. We can now reformulate adversarial attack generation as a search problem:

**Definition 1.** *Let $f : I \longrightarrow L \times \mathbb{R}$ be a trained model that takes as input an image $i \in I$ and returns a predicted label $l \in L$. Let $m : I \longrightarrow I$ be a transformation function that mutates (i.e. applies changes to) an image $i \in I$. The problem is finding a mutated image $m(i)$ such that $f(m(i)) \neq f(i)$, with the constraints that both $i$ and $m(i)$ share the same correct label (same ground truth).*

Attack generation strategies can be *targeted* or *un-targeted*. The former aims to flip the prediction to a specific label or classification outcome, while the latter aims to lead the model toward producing any incorrect outcome. We focus on un-targeted attack generation: for demonstrating the vulnerability of a machine learning model, it is sufficient to generate mutated images $m(i)$ that flip the predicted output to any other label than the ground truth label. Since a classification model returns both the label and the corresponding confidence level, we can use the latter to guide the search toward the flipped prediction. More precisely, given a classification model $f : I \longrightarrow L \times \mathbb{R}$, a seed image $i$, and its mutated variant $m(i)$, we optimize the following objective:

$$\min \quad O_1 = \begin{cases} f(m(i))_1^{conf} - f(m(i))_2^{conf} & \text{if } f(i)_1^l = f(m(i))_1^l \\ -f(m(i))_1^{conf} & \text{if } f(i)_1^l \neq f(m(i))_1^l \end{cases} \tag{1}$$

In other words, this objective aims to reduce the confidence for the most likely prediction/label ($f(m(i))_1^{conf}$), while increasing the confidence for the second-most-likely prediction ($f(m(i))^{conf_2}$). Therefore, the overall goal is to reduce the difference between the top-2 labels until the model $f$ flips the prediction to a different label (condition $f(m(i))_1^{label} \neq f(m(i))_1^{label}$). In general, Equation 1 takes values in [-1,1]. A zero value indicates that the models assign equal confidence scores to the top-2 labels. A negative value indicates that the model $f$ flips the prediction to a different label, whose confidence level corresponds to the absolute value of Equation 1.

We can expand this to a multi-objective problem where both "fooling" the model and reducing the number of perturbations (at the pixel level) are equally important:

**Definition 2.** *The problem is finding a mutated image $m(i)$ such that $f(m(i)) \neq f(i)$ and that minimizes the distance $d(i, m(i))$, with the constraints that both $i$ and $m(i)$ share the same correct label (same ground truth).*

Beyond flipping the prediction outcome by optimizing for $O_1$, we now also need an additional objective to guide the search towards minimizing the difference between the original image $i$ and its mutated counterpart $m(i)$.

To this end, our second objective counts the number of pixels that differ between the seed image $i$ and the mutated image $m(i)$:

$$\min\ O_2 = \pi(m(i[a,b]) \neq i[a,b]) \tag{2}$$

$$= |\{e_a, b \in i : i[a,b] \neq m\left(i[a,b]\right)\}| \tag{3}$$

where $i[a,b]$ and $m\left(i[a,b]\right)$ denote the pixel values in row $a$ and column $b$ for the two images $i$ and $m(i)$, respectively.

These two objectives are *conflicting*. A simple solution for $O_1$ may consist of changing all pixels in the original figure $i$ such that the object is no longer recognizable for the model $f$. However, such a solution would not be optimal for $O_2$. Vice versa, a new image with zero alteration would be optimal for $O_2$, but not flip the prediction as sought by $O_1$.

Given the conflicting nature of our objectives, it is not possible to find one single solution that simultaneously optimizes them all. In other words, the problem is inherently multi-objective where the goal becomes to find the set of optimal trade-offs between $O_1$ and $O_2$. In particular, we aim to find *Pareto efficient* trade-offs based on the concepts of *dominance* and *Pareto optimality*.

### 3.3.1 Single and Multi-objective Differential Evolution

To find adversarial attacks, we rely on differential evolution (DE) only. Hence, compared to `BMI-FGSM`, we do not use any algorithm to approximate the gradient. We consider two different variants of DE: `Pixel-SOO`, a traditional single-objective variant (to optimize $O_1$) and `Pixel-MOO`, a multi-objective variant based on the *non-dominated sorting algorithm* (NSDE) [10] (to optimize $O_1$ and $O_2$).

Both variants iteratively evolve a pool of $N$ randomly generated adversarial attacks, called *population*. In each iteration, $N$ offspring attacks are generated from the population using variation operators. Then, the population for the next iteration is obtained by combining the previous population and the offspring attack, forming a pool $Q$ of $2 \times N$ attacks and selecting the $N$ top individuals. The selection is performed using an *environmental selection* and represents the main difference between `Pixel-SOO` and `Pixel-MOO`.

In `Pixel-SOO`, the *environmental selection* is applied by selecting the best $N$ individuals among the parent and the offspring solutions/attacks according to the main objective $O_1$. This mechanism is *elitist* since the best attacks can survive across the generations until new better solutions are found.

In `Pixel-MOO`, the *environmental selection* is performed by applying the *fast non-dominated sorting algorithm* [43], which ranks the solutions in $Q$ into sub-dominated fronts based on the dominance relation.

In the following, we detail (1) the encoding schema, (2) how we initialize the initial population, (3) the variation operator.

#### 3.3.1.1 Encoding schema.

As mentioned before, an adversarial attack is produced by altering a seed image $i$. Instead of representing/encoding an adversarial attack as a completely new image, we only encode the changes to be applied, also called the *mask*. In particular, given the seed image $i$, we encode a solution/attack in NSDE as a list of pixels to change: $X = [x_1, \ldots, x_k]$. Each entry $x_j$ in $X$ is a tuple $[a, b, value_j]$, where $a$ and $b$ determine the position of the pixel to change (i.e., $a$ is the row index and $b$ is the column index), while $value_j$ indicates the new pixel value in RGB notation.

#### 3.3.1.2 Initialization.

The first step to initializing NSDE involves generating an initial pool of adversarial attacks. To this aim, we create $N$ attacks by creating an empty mask $X = []$ and adding some changes using the *add* operator, one of three alternative variation operators described below.

#### 3.3.1.3 Variation operator.

Given a parent attack $X$, we design three types of operators that *add*, *delete*, or *change* entries in $X$. Each operator is applied with probability 1/3.

The **add** operator randomly inserts one entry in $X$ with probability $\sigma = 1$; a second entry is added with probability $\sigma = 0.5$; the third one with probability $\sigma = 0.25$; and so on until no other element is added. To add a new element/entry $x$ in $X$, this operator randomly selects one pixel from the original seed image $i$ with position $row_j$ and $col_j$ and draws three random (noise) values $\delta(\mu, \lambda)$ from a Gaussian distribution with mean $\mu$ and standard deviation $\lambda$, that will be applied to the respective R, G and B channels. Hence, the new entry $x$ will be equal to $[row_j, col_j, value_j + \delta(\mu, \lambda)]$.

The **delete** operator simply deletes one entry/tuple from the mask $X$. However, this operator is applied only if $X$ contains at least two entries. This operator plays a critical role in our multi-objective formulation as it allows to remove spurious pixel changes that do not contribute to changing the prediction results of the model $f$ under test.

The **change** operator changes the pixel values using the traditional *differential operator*. Given a parent image to mutate $X$ and a donor solution $Y$, a new solution $X'$ is formed by using the following formula:

$$X'[a,b] = \begin{cases} R_1[a,b] + F \cdot (R_2[a,b] - Y[a,b]) & \text{if } r < CR \\ X[a,b] & otherwise \end{cases} \quad (4)$$

where $r \in [0,1]$ is a randomly generated number; $R_1$ and $R_2$ are other solutions within the population. There are various variants of the differential operator that differ in how $X_1$, $X_2$, and the donor solution $Y$ are selected. In this report, we use the standard `DE/rand/1` variant, where `rand` indicates that the donor $Y$ is randomly selected, while `1` indicates there is only one donor solution. Finally, the other solutions $R_1$ and $R_2$ are always randomly selected from the population.

In Equation 4, $F \in [0,2]$ is called *scaling factor* and establishes how far the new solution $X'$ is from the original solution $X$ based on the differentials values of the donor solution $Y$. Hence, $F$ balances both exploration and exploitation. Finally, $CR \in [0,1]$ is the *crossover rate* and determines how many pixels in $X$ will be changed.

We apply a small tweak in our context compared to the traditional differential operator. If the pixel $X[a,b]$ differs from the original seed solution, Equation 4 may remove this change if $R_1[a,b]$, $R_2[a,b]$, and $Y[a,b]$ are identical to the original seed image. To prevent this case, we set the pixel $R_1[a,b] = [0,0,0]$ (in RGB notation) if $R_1[a,b]$ is identical to the pixel of the initial image/seed. The same is done for $R_2[a,b]$ and $Y[a,b]$. Notice that this tweak is applied only if $X[a,b]$ differs from the original seed's pixel in row $a$ and column $b$.

#### 3.3.1.4 Additional Remarks.

`BMI-FGSM` by Lin et al. [99] and our approach share the main goal of flipping the label prediction. However, there are critical differences that are worth highlighting. The first difference regards the main objective or fitness function. The main (single) fitness function used by `BMI-FGSM` aims to "*suppress the probability of the ground-truth label*" [99] until another false label is predicted. Our $O_1$ explicitly maximizes the confidence level for the false label, even after the prediction has been flipped (second case in Equation 1).

The most critical difference is within the variation operator. `BMI-FGSM` combines differential operators and the *iterative fast gradient sign method*. Instead, `Pixel-MOO` and `Pixel-SOO` rely on the differential operators but introduce two novel ways to generate offspring: (1) the *add* and (2) *delete* operators. The latter allows deleting pixel changes that do not contribute to optimizing $O_1$ (for `Pixel-SOO`) or that are not Pareto efficient w.r.t. $O_1$ and $O_2$. (for `Pixel-MOO`). Finally, `Pixel-MOO` explores multi-objective optimization where the mask size is considered as an explicit objective with the goal of generating minimal adversarial attacks. Instead,

`BMI-FGSM` targets only the prediction outcome and does not constrain the number of pixels that can be altered to reach a prediction flip.

## 3.4 Speeding Up Program repair for Self-driving Cars With Regression Testing

In this deliverable, we focus on self-driving cars (or automated driving system) as an instance of Cyber-Physical Systems (CPS) to be repaired using evolutionary algorithms [3]. Specifically, our focus is on the integration component responsible for preventing feature interaction failures. Self-driving cars typically incorporate multiple features [3], such as automated emergency braking (AEB), adaptive cruise control (ACC), and traffic sign recognition (TSR), where each feature individually automates an independent driving function.

Feature interaction failures refer to scenarios or driving situations in which one feature affects the behavior of another feature, leading to violations of safety requirements, such as collision avoidance or speed limit constraints. To mitigate these scenarios, engineers develop the decision logic of the integration component, often in the form of rule sets, ensuring that interactions do not result in failures. This is typically achieved using existing techniques for feature interaction resolution.

In this section, we introduce a novel approach called RESTORE (REgreSsion Testing prOgram REpair) to automatically repair rule-based integration components. RESTORE combines evolutionary algorithms for the patch generation phase and regression testing for the patch validation phase.

Algorithm 2 provides the pseudo-code for RESTORE. The inputs to RESTORE are:

- A faulty integration component $IC = (f_1, \ldots, f_n, \Pi)$, where $f_1, \ldots, f_n$ represent $n$ features and $\Pi$ denotes the integration rule set.
- A test suite $TS$ that verifies the safety requirements $SR = \{r_1, \ldots, r_k\}$.

The output is a repaired integration component, i.e., $(f_1, \ldots, f_n, \Pi^*)$, that satisfies all test cases in $TS$, resulting in a *test-adequate* patch $\Pi^*$.

RESTORE implements the (1+1) Evolutionary Algorithm (EA) with an archive, following recommendations from prior studies [4, 3]. While traditional evolutionary tools, such as GenProg [90], employ population-based algorithms like Genetic Programming, we utilize an evolutionary algorithm with only one candidate patch that evolves throughout the generations.

This is because each candidate patch undergoes evaluation against the complete test suite to determine whether the changes result in an increase or decrease in the number of failing tests. As a result, the overall cost of a single iteration or generation is calculated as $N \times \sum_{tc \in TS} cost(tc)$, where $cost(tc)$ represents the cost associated with the test $tc$, and $N$ represents the population size. In our case, $N = 1$, which reduces the overall evaluation cost to $\sum_{tc \in TS} cost(tc)$.

Population-based algorithms operate under the assumption that the evaluation cost of individual patches is relatively small, allowing test results to be collected within a few seconds and used to provide feedback (i.e., compute the fitness function) for the search. However, this assumption does not hold in our specific context, as running a single simulation-based test case takes several minutes. Consequently, evaluating a pool of patches in each iteration or generation becomes excessively time-consuming, requiring several hours to complete.

### 3.4.1 Algorithm Outline

RESTORE initialize the search by adding the buggy integration rule-set $\Pi$ in the *archive* (line 2 of Algorithm 2. The solution(s) in the archive are evolved through the loop in lines 5-10. In particular, in each iteration RESTORE selects one patch $\Pi_p \in$ *archive* randomly (line 6) and creates one offspring ($\Pi_o$) in line 7 (routine

**Algorithm 2:** RESTORE

**Input:**
$(f_1, \ldots, f_n, \Pi)$: Faulty self-driving system
$TS$: Test suite
**Result:** $\Pi^*$: repaired integration rules satisfying all $tc \in TS$

```
1  begin
2      Archive ⟵ Π
3      TS̄ ⟵ PRIORITIZE(TS)                                  // Regression testing
4      Ω ⟵ RUN-EVALUATE(Π, TS)
5      while not(|Archive|==1 & Archive satisfies all tc ∈ TS) do
6          Π_p ⟵ SELECT-A-PARENT(Archive)                   // Random selection
7          Π_o ⟵ GENERATE-PATCH(Π_p, TS, Ω)
8          S̄ ⟵ SAMPLE(TS̄, K)                               // Sample K tests
9          Ω ⟵ RUN-EVALUATE(Π_o, S̄)
10         Archive ⟵ UPDATE-ARCHIVE(Archive , Π_o, Ω)
11     if |Archive|==1 & Archive satisfies all tc ∈ TS) then
12         Ω ⟵ RUN-EVALUATE(Archive, TS̄)    // Validate against the entire test suite
13     return Archive
```

GENERATE-PATCH) by (1) applying fault localization (see Section 3.4.3) and (2) mutating the rules in $\Pi_p$. The routine GENERATE-PATCH is presented in subsection 3.4.4.

Then, the offspring $\Pi_o$ is evaluated (line 9) by running a subset of the test suite in line 8 of Algorithm 2. The test cases are selected based on the different regression testing strategies discussed in subsection 3.4.2. The offspring $\Pi_o$ is added to the *archive* (line 8 of Algorithm 2) if it decreases the number of failing tests compared to the patches currently stored in the *archive*. The *archive* and its updating routine are described in details in subsection 3.4.5. The search stops when the termination criteria are met (see Section 3.4.6).

### 3.4.2 Regression testing

To minimize the cost of the patch validation phase, RESTORE utilizes regression testing techniques. Regression testing is applied in three key steps of Algorithm 2, specifically in lines 3, 8, and 9.

Firstly, the test cases in the test suite $TS$ are sorted using *SO-SDC-Prioritizer* by Birchler et al. [19], as described in our deliverable D5.3. *SO-SDC-Prioritizer* employs multi-objective evolutionary algorithms to sort simulation-based test scenarios based on their diversity in terms of read features. In the context of self-driving cars, simulation-based tests involve varying road map characteristics. For each test case $tc \in TS$, we extract the road features mentioned in Table 1, which were introduced in our previous work [19]. Subsequently, the tests are sorted using NSGA-II [43] in a manner that prioritizes running more diverse and less expensive test cases first. The diversity between two test cases, $tc_1$ and $tc_2$, is measured using the Euclidean distance between their respective road feature vectors. It is important to note that we normalized the features using *z-score* normalization, which is a well-known method for addressing outliers and rescaling a set of features with different ranges and scales [64]. *Z-score* normalization scales the features using the formula $\frac{x-\mu}{\sigma}$, where $x$ is the feature to be rescaled, $\mu$ is its arithmetic mean, and $\sigma$ is the corresponding standard deviation [64].

The execution cost of each test case $tc \in TS$ is estimated based on the past execution cost gathered from previous test runs, as recommended in the literature [51, 174]. This estimation is accurate for SDC (Self-Driving Cars) since the cost of running simulation-based tests is proportional to the length of the road and the cost of rendering the simulation, which are fixed simulation elements. As demonstrated by Birchler et al. [19], past execution cost provides an accurate estimation, considering that simulation time does not significantly vary across multiple test re-runs.

Table 1: Road Characteristics Features

| ID | Feature | Description | Type | Range |
|---|---|---|---|---|
| F1 | Direct Distance | Euclidean distance between start and finish | float | [0-490] |
| F2 | Road Distance | Total length of the road | float | [56-3,318] |
| F3 | Num. Left Turns | Number of left turns on the test track | int | [0-18] |
| F4 | Num. Right Turns | Number of right turns on the test track | int | [0-17] |
| F5 | Num. Straight | Number of straight segments on the test track | int | [0-11] |
| F6 | Total Angle | Total angle turned in road segments on the test track | int | [105-6,420] |
| F7 | Median Angle | Median of angle turned in road segment on the test track | float | [30-330] |
| F8 | Std Angle | Standard deviation of angled turned in road segment on the test track | int | [0-150] |
| F9 | Max Angle | The maximum angle turned in road segment on the test track | int | [60-345] |
| F10 | Min Angle | The minimum angle turned in road segment on the test track | int | [15-285] |
| F11 | Mean Angle | The average angle turned in road segment turned on the test track | float | [5-47] |
| F12 | Median Pivot Off | Median of radius of road segment on the test track | float | [7-47] |
| F13 | Std Pivot Off | Standard deviation of radius of turned in road segment on the test track | float | [0-23] |
| F14 | Max Pivot Off | The maximum radius of road segment on the test track | int | [7-47] |
| F15 | Min Pivot Off | The minimum radius of road segment on the test track | int | [2-47] |
| F16 | Mean Pivot Off | The average radius of road segment turned on the test track | float | [7-47] |

*SO-SDC-Prioritizer* is applied only once at the initialization phase of RESTORE (line 3 of Algorithm 2); its overall execution cost is negligible to the overall the test execution cost for simulation-based test suites, as also reported by Birchler et al. [19].

The prioritized test suite $\overline{TS}$ plays a critical role in the patch validation phase of RESTORE, specifically in lines 8 and 9 of Algorithm 2. During this phase, each generated patch $\Pi_o$ is validated against a subset of the prioritized/optimized test suite, denoted as $\overline{S} \subset \overline{TS}$. We consider three different heuristics for selecting the test cases to run:

- *Only-failing tests*: With this heuristic, only the failing test cases are selected to validate the generated patch $\Pi_o$. While this solution can significantly reduce the validation cost (in case only a few tests fail), it may not detect malformed patches that can cause other previously passing tests to fail.
- *Top-K most diverse tests*: With this heuristic, the first $K$ test cases are selected from the prioritized test suite $\underline{TS}$. Therefore, the focus is on considering the most diverse and less expensive simulation tests. The value of $K$ is a user-defined parameter that can be chosen based on (1) domain expert preference and (2) the number of failing tests. Ideally, $K$ should be greater than the number of failing tests to provide additional guarantees that the generated patch will not break previously passing tests.
- *Random sampling*: This heuristic randomly samples $K$ test cases from $TS$. It serves as a baseline for comparison and assessment of the previous two heuristics.

In line 9 of Algorithm 2, each patch is validated using $\overline{S} \subset \overline{TS}$. As a result, the fitness of each objective is computed based on a subset of the test suite rather than the whole test suite as typically done in the

literature [90, 4]. The main objective to optimize (minimize in our case) is the number of failing test cases in $\overline{S}$, which is determined using one of the three heuristics discussed earlier.

Given a candidate patch $\Pi_o$, its fitness function is computed as follows:

$$fitness(\Pi_o) = \frac{\sum_{tc \in \overline{S}} \mu_{fail}(tc)}{\mid \overline{S} \mid} \tag{5}$$

Here, $\mu_{fail}(tc) \in 0, 1$ is the counting function that determines whether a test case $tc \in \overline{S}$ fails, and $\mid \overline{S} \mid$ represents the number of selected test cases.

### 3.4.3 Fault localization

To determine which line in the integration rule set should be altered or mutated, RESTORE employs the Tarantula formula [82] for fault localization. , a rank refinement is applied. The formula ranks the integration rules based and determine the most suspicious one that may be causing the failure. With Tarantula, the suspiciousness is calculated by checking which lines of code are run by the selected test cases in combination with the knowledge that the test case passed or failed. The formula for this calculation [82] is as follows:

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}} \tag{6}$$

`suspiciousness(e)` is the final number that indicates the suspiciousness level of each line of code. When the closer the number reaches 1, to higher its suspiciousness level. Notice that $totalfailed$, $totalfailed$, and coverage ($failed(e)$ and $passed(e)$) are computed with regards to subset $\overline{S}$ of the test suite.

### 3.4.4 Patch generation

Patches are generated by mutating the rules in $\Pi$. To this aim, we consider four mutation operators:

- The *modify* operator changes the existing conditions in the rule set by either (1) replacing a conditional operator (e.g., $\leq$) with another one (e.g., $\geq$) or (2) modifying the constants in the conditions.
- The *delete* operator randomly deletes one if statement (for single-clause conditions) or one clause in a multi-clause condition.
- The *shift* operator changes the priorities of the rules in $\Pi$ by shifting them up or down (or shifting the if conditions). This corresponds to the *shift* operator introduced by Abdessalem et al. [4].
- The *add* operator randomly adds one rule by (1) copy-pasting an existing rule and (2) applying the *modify* operator.

### 3.4.5 Archive Updates

The archive stores the best partial fixes discovered during the search process. In the beginning, at line 2 of Algorithm 2, the archive is initialized with the faulty rule set $\Pi$. As new patches $\Pi_o$ are generated and evaluated, they are compared against all the patches stored in the archive. The comparison is performed using the concept of dominance in Pareto optimization, considering the failing tests as the objectives to compare on.

For each generated patch $\Pi_o$, an objective vector $O$ is assigned, where the length of $O$ corresponds to the number of selected test cases, i.e., $\mid O \mid = \mid \overline{S} \mid$. Each entry in the objective vector is assigned a binary value indicating whether the test $tc_i \in \overline{S}$ passes or fails when executed against $\Pi_o$.

A generated patch is added to the archive if it either (1) dominates one of the patches already in the archive, or (2) is non-dominated by any patches in the archive. In the former case, the dominated patches will be removed from the archive. This archiving strategy allows the storage of partial patches that satisfy different patches generated previously.

The archive is updated at the end of each iteration, as indicated in line 10 of Algorithm 2.

### 3.4.6 Termination and Final patch validation

The search process terminates under two conditions: either when the allocated search budget is exhausted or when a test-adequate patch is discovered. A test-adequate patch refers to a patch that satisfies all test cases $tc \in \overline{S}$, indicated by the presence of a single patch $\Pi^*$ in the archive. It is important to note that although the algorithm stores all partial patches, only one final solution (if found) is generated to satisfy all failing tests.

As the patches are validated against a subset of the entire test suite, the final patch undergoes additional validation against the complete test suite $TS$". However, the validation of the complete patch considers the test cases prioritized by *SO-SDC-Prioritizer*. This prioritization is employed to increase the likelihood of detecting any failing tests early in the process, providing prompt feedback in the case of faulty patches.

## 3.5 Prototype of the Build and Test Scheduler

When a team of developers commits a new code change to a version control system (such as git or svn), it triggers the pipeline and initiates the build process. Typically, the build process consists of multiple jobs that run in parallel. Each job consists of various tasks that are generally executed sequentially. Common tasks include Static Code Analysis, Build, and Test. The purpose of continuous integration is to ensure thorough testing is conducted before code submission, preventing build failures and avoiding delays in receiving prompt feedback, which is a key aspect of continuous integration.

However, in the context of CPSs, task execution may require the use of simulators and hardware-in-the-loop (HiL) setups, which may have limited availability and be shared among multiple projects. Consequently, the number of jobs and tasks can be reduced, and only those that provide useful information are executed. The challenge lies in optimally allocating simulators and hardware while prioritizing jobs and tasks.

The primary objective is to allocate tasks in a cost-effective manner and ensure sustainable pipeline usage. For instance, the build process can be restructured to concurrently perform tests on different sub-modules of a system. Test cases can focus on specific sections of code, allowing for parallel execution of all tests. Our approach aims to optimize the allocation of the test tasks by taking into account (1) the different levels and types of tests, (2) the cost associated with the resources needed to run the tests (e.g., servers with GPU), and (3) the dependencies between the tests and the other tasks in the build schedule.

The ultimate goal encompasses three key optimization aspects: (i) minimizing the test execution time, (ii) maximizing the effectiveness of testing, and (iii) optimizing the utilization of simulators and hardware devices.

### 3.5.1 Problem Formulation

In the context of this report, we focus on two main types of tasks: (1) build tasks related to compilation, and (2) test tasks. CPSs consist of multiple modules or components that need to be built and tested in a specific order as specified in build scripts or CI/CD configuration files. Hence, a build schedule consists of completing a

set $B = \{\langle C_1, T_1\rangle, \cdots, \langle C_n, T_n\rangle\}$, where $C_i$ represents the compilation task for the component at position $i$ within the schedule, and $T_i$ denotes the corresponding test suite to be executed.

Our goal is to sort or prioritize the test cases within each test suite $T_i$ in such a way that the test cases with a higher likelihood of detecting regression faults are executed earlier. In regression testing, we typically do not know in advance which test case or test suite will fail until the tests are actually run. Therefore, prioritization relies on heuristics that are correlated with fault detection capabilities.

In this report, we focus on black-box heuristics, which do not require access to the source code or the instrumentation of code to parse execution traces. This is particularly crucial in CPS, as different components are often written in various programming languages, necessitating different instrumentation tools for the different languages used. Within black-box heuristics, we specifically concentrate on input diversity, a well-known heuristic that aims to prioritize the execution of the most diverse test cases first. A comprehensive overview of diversity-based black-box heuristics is discussed in Section 2.3.

To illustrate the problem of test scheduling without loss of generality, we can formulate it as follows:

**Definition 3.** *Let $B = \{\langle C_1, T_1\rangle, \cdots, \langle C_n, T_n\rangle\}$ be the list of compilation and test tasks, where $C_i$ represents the compilation task for component $i$, and $T_i = \{t_{i,1}, \cdots, t_{i,m}\}$ is the corresponding test suite associated with component $i$. The problem is to find an optimal order of test cases $\tau$ that (1) maximizes the diversity between subsequent tests to be executed (2) minimizes the execution time (in seconds), (3) minimizes the resource usage (such as GPU and CPU time).*

### 3.5.2 Test Diversity

Several heuristics have been proposed in the literature to measure the diversity among test cases. These heuristics include Jaccard distance [75, 76], Levenshtein distance [73, 92], information retrieval techniques [150], and topic model techniques [9]. However, these techniques make an assumption that the same keywords or identifiers will appear in multiple test cases.

In systems with different granularity levels, such as unit-level and system-level tests, the assumption that the same keywords or identifiers will appear in multiple test cases may not hold. In fact, unit-level tests that target the same classes may share common identifiers or keywords, while system-level tests invoke components and call specific APIs; thus, reducing the number of overlapping keywords and identifiers. In general, system-level tests focus on the interactions between components rather than individual methods or functions within a single class.

To address this issue/limitation, we leverage WordNet [109], a well-established lexical database of English words grouped in sets of synonyms, called *synsets*. Synsets are connect via synonym relationships, i.e., synsets sharing the same meaning are connected in the semantic space. In addition to synonyms, WordNet also provides information about additional relationships between synsets, such as hypernyms (superordinate terms), hyponyms (subordinate terms), meronyms (part-whole relationships), and holonyms (whole-part relationships). These relationships help to establish connections and hierarchy between different concepts.

In the following subsection, we describe how we use WordNet to compute the distance/diversity between test cases and the various pre-processing steps.

### 3.5.3 Pre-processing

Before using WordNet, the test cases undergo a pre-process step aiming to extract words to query on the WordNet and exclude programming language specific keywords that do not contribute to the test semantic. To this aim, we pre-process the test by parsing them as text and applying various information retrieval transformations, namely (i) *tokenization*, *removing stop words*, and (ii) *stemming*. First, *tokenization* aims to extract words in

the text and remove non-relevant characters, such as punctuation marks, special characters, and numbers [125]. We split compound names (i.e., identifiers) into tokens using *camel case* and *snake case* splitting [126]. For example, the method name `get_data` will be split into the two tokens `get` and `data`.

We further applied *stop-word list* and *function* to remove words that do not contribute to the semantic content of the analyzed text [41, 130]. The former is a list of generic words (i.e., prepositions, articles, auxiliary verbs, and adverbs) that are commonly found in any text, thus, not providing any useful information. Our stop-list includes the standard list for the English language [130], plus a list of words that are specific to the programming languages (i.e., reserved keywords like `class` in `Java`). The stop-word function instead removes words that are too short [41], i.e., that contain less than three characters. Finally, we applied *stemming* algorithms to reduce the words to their root form using the Porter stemming algorithm [137].

#### 3.5.3.1 WordNet Distance .

Given two test cases $t_i$ and $t_j$, we measure their semantic distance as the average pairwise distance for the $k$ most frequent keywords in the WordNet taxonomy/database:

$$d_k(t_i, t_j) = \frac{1}{k} \sum_{m=1}^{k} wd(w_{m,i}, w_{m,j}) \tag{7}$$

where $w_{m,i}$ and $w_{m,j}$ are the $m$-th most frequent words (after processing) in the two test cases $t_i$ and $t_j$, respectively; $wd(.)$ represents the WordNet distance.

In this report, we consider different WordNet distances:

1. *Path Distance (PD) similarity*: It measures the distance in the WordNet taxonomy as the shortest path that connects the synsets in the is-a (hypernym/hyponym) taxonomy.

2. *Leacock Chodorow* (LCH) similarity [91]: It enhances the Path-based similarity by taking into account the depth of the taxonomy. It is computed as the negative logarithm of the shortest path ($path$) between two words ($synset_1$ and $synset_2$), divided by twice the total depth of the taxonomy ($D$):

$$LCH(synset_1, synset_2) = -\log\left(\frac{path(synset_1, synset_2)}{2 \times D}\right) \tag{8}$$

3. *Wu & Palmer* (WUP) similarity [169]: It considers the depths of the two synsets in the WordNet taxonomies, along with the depth of the Least Common Subsumer (LCS), also called the most specific ancestor node:

$$WUP(synset_1, synset_2) = 2 \times \frac{depth\left(lcs(synset_1, synset_2)\right)}{depth(synset_1) + depth(synset_2)} \tag{9}$$

4. *Resnik similarity* [140]: It measures the similarity of two synsets based on the Information Content (IC) of the Least Common Subsumer (or most specific ancestor node):

$$RS(synset_1, synset_2) = IC(lcs(synset_1, synset_2)) \tag{10}$$

#### 3.5.3.2 Encoding

Since the solution for test prioritization is an ordered sequence of tests to run during the regression testing phase, we encode solutions using a *permutation encoding*. Our goal is to establish the execution order of N tests; thus, our approach encodes each chromosome as an N-sized array of integers that represent the position of a test in the desired order. For instance, let $\tau = \langle t_1, t_2, t_3 \rangle$ be a chromosome for a test suite containing three test cases. In this case, test case $t_1$ will be executed first, followed by $t_2$ and $t_3$.

### 3.5.4 Multi-objective optimization

In this report, we propose using multi-objective optimization that considers the execution cost and test case diversity as two different objectives to optimize simultaneously. Assume that $\tau = \langle t_1, \ldots, t_n \rangle$ is a solution (i.e., test execution order) generated by the search process. The first goal to optimize is computed using the following equation:

$$\max f_1(\tau) = \sum_{i=2}^{n} \frac{wd(t_i, t_{i-1})}{i} \tag{11}$$

where $wd(t_i, t_{i-1})$ denotes the WordNet distance between a test $t_i$ and its predecessor $t_(i-1)$ in the ordering. The contribution of each test case $t_i$ to the cumulative diversity is divided by its position $i$ in the ordering $\tau$. In other words, this objective promotes solutions where the most diverse test cases are executed earlier.

The second objective measures how steadily the cumulative test execution runtime increases when executing the tests with a given order $\tau$:

$$\min f_2(\tau) = \sum_{i=1}^{n} \frac{time(t_i))}{i} \tag{12}$$

where $time(t_i)$ denotes the execution time of the test case $t_i$ in $\tau$. The contribution of each test case $t_i$ to the cumulative runtime is divided by its position $i$ in the ordering $\tau$, with the goal of promoting solutions where the least expensive test cases are executed earlier. Notice that this objective should be minimized.

Finally, we consider a third objective that considers the cost associated with the resource needed for each test with a given order $\tau$:

$$\min f_3(\tau) = \sum_{i=1}^{n} \frac{cost(t_i))}{i} \tag{13}$$

where $cost(t_i)$ denotes the cost of the test case $t_i$ in $\tau$. The cost is measured considering the average cost per second associated with the hardware resources needed to run the test. The cost can be approximated considering the average rent cost for using local machines or the cost of renting external cloud sources (e.g., Azure service) with dedicated hardware (e.g., GPU).

Finding optimal solutions for problems with multiple criteria requires trade-off analysis. Given the conflicting nature of our two objectives [2], it is not possible to obtain one single solution that optimizes both objectives at the same time [36]. Hence, we are interested in finding the set of solutions that are optimal compromises between the three objectives.

For multi-objective problems, the concept of optimality is based on concepts of *Pareto dominance* and *Pareto optimality*[36]. In particular, a solution $\tau_A$ *dominates* another solution $\tau_B$ ($\tau_A <_p \tau_B$) if and only if at the same level of diversity and run-time, $\tau_A$ has a lower cost than $\tau_B$. Alternatively, $\tau_A$ *dominates* $\tau_B$ if and only if, at the same level of cost a time, $\tau_A$ has a larger diversity than $\tau_B$. Finally, $\tau_A$ *dominates* $\tau_B$ if and only if, at the same level of cost a diversity, $\tau_A$ has a lower runtime execution than $\tau_B$.

Among all possible solutions, we are interested in finding those that are not dominated by any other possible solution (*Pareto optimality*). Pareto optimal solutions form the so-called *Pareto optimal set* while the corresponding objective values form the *Pareto front*.

**3.5.4.1 NSGA-II** To find Pareto optimal solutions, we uses NSGA-II [43]. This genetic algorithm provides well-distributed Pareto fronts and performs best when dealing with two or three search objectives [43]. NSGA-II shares the main loop of the genetic algorithm. Thus, it shares the same encoding schema as well as mutation and crossover operators. However, it differs on how parents are selected for reproduction and how the new

[2] Diverse tests are not necessarily the least expensive to run

population is formed for the next generation. Parents are selected using the *binary tournament selection*, which compares pairs of solutions in tournaments and selects the "fittest" solution from each pair for reproduction. Finally, the population for the next generation is obtained by selecting the "fittest" solutions among parent and offspring solutions (elitism).

In NGSA-II, the "fitness" of the solutions is determined using the *fast non-dominated sorting* algorithm and the concept of *crowding distance* [42]. The former ranks the solutions according to their dominance relations. All non-dominated solutions within a given population are inserted in the first front $F_1$ (rank $r = 1$); the subsequent front $F_2$ (rank $r = 2$) contains all solutions that are dominated only by the solutions in $F_1$; and so on. Hence, solutions in the fronts with lower rank are "fitter" according to the Pareto optimality.

Instead, the crowding distance aims at promoting more diverse (isolated) solutions within each dominance rank. The crowding distance for a given solution is computed as the sum of the distances between such an individual and all the other individuals with the same rank. This heuristics is put in place to avoid selecting individuals that are too similar to each other.

### 3.5.5 Geneti operators

NSGA-II evolves test case orderings by applying crossover and mutation operators. Given the nature of our permutation problem, we use the *Partially-Mapped Crossover* and various *permutation* mutation operators.

**3.5.5.1 Partially-Mapped Crossover (PMX)** In the crossover, an offspring $o$ is formed from two selected parents $p_1$ and $p_2$ , with the size of N, as follows: (i) select a random position $c$ in $p_1$ as the cut point; (ii) the first $c$ elements of $p_1$ are selected as the first $c$ elements of $o$; (iii) extract the $N - c$ elements in $p_2$ that are not in $o$ yet and put them as the last $N - c$ elements of $o$.

**3.5.5.2 Mutation operators** A chromosome $p$ can be mutated one or more times according to the given mutation probability. In each round of mutation, one of the three following mutation operators[151] is selected randomly with an equal chance of $0.33\%$ to perform the mutation:

- **SWAP mutation:** This mutation operator randomly selects two positions in a chromosome $p$ and swaps the index of two genes (test case indexes in the order) to generate a new offspring.
- **INVERT mutation:** This mutation operator randomly selects a segment (with a random size) of the given chromosome $p$. Then, it reverses the selected segment end to end and reattaches it to generate a new offspring.
- **INSERT mutation:** This mutation randomly selects a gene in the chromosome $p$ and moves it to another index in the solution to generate a new offspring.

We consider the three operators above since prior studies [151] showed that using multiple mutation operators for permutation-based optimization problems increases the likelihood of escaping from solutions that are locally optimal under one mutation operator. This procedure used for the mutation is the same in both of the *SDC-Prioritizer* variants introduced in this report.

# 4 Evaluation: Carving Unit-Level Test Cases

## 4.1 Study Setup

In this section, we evaluate our prototype, presented in Section 3.2. Our investigation is steered by the following research questions:

**RQ$_1$** Can MICROTESTCARVER generate unit tests based on information carved from E2E tests? (Feasibility)

**RQ$_2$** How do the tests generated by our approach compare to EvoSuite-generated tests in terms of understandability?

**RQ$_3$** How do the tests generated by our approach compare to manually written tests in terms of understandability?

We have carried out the evaluation on three popular open-source Java web applications. We have used the following criteria for our selection of projects:

- The project is an open-source Java web application.
- The project is popular, active, and mature as measured in terms of forks, stars, and commits.
- The project has a test suite with tests for different layers of the test pyramid, especially unit tests.

Using these criteria, we found nearly two hundred projects on GitHub. Next, we manually search among these projects to select three projects from different domains with varying sizes. The selected projects are: *Spring-Testing*, *PetClinic*, and *Alfio*; detailed characteristics are listed in Table 2. We manually conducted end-to-end tests for these projects, which covered their core functionalities. Spring-Testing and PetClinic are tested in Java 11, while Alfio is tested in Java 17. Manually written unit tests are available for all of these applications, which makes it possible to compare them with carved tests for RQ3.

The first research question investigates the feasibility and analyzes the MICROTESTCARVER approach. For the second and third research questions, we carry out an exploratory case study in which we qualitatively investigate the MICROTESTCARVER tests and compare them to tests that are automatically generated by EvoSuite, and to tests that were manually written and part of the open source projects already. Our focus in this investigation is on the understandability of the generated test cases, and not so much on their effectiveness (e.g., in reaching high code coverage). In the following, we discuss the results of our research questions.

## 4.2 RQ1: Feasibility of the unit test generation based on E2E Tests

Table 3 presents the experimental results of the carved unit tests regarding execution rates. In total, 41 tests are carved for 21 CUTs of the three study subjects; 5 tests for Spring-Testing, 20 for PetClinic, and 16 for Alfio. Of the 41 carved tests, 35 are executable (85%), 37 have an executable body, and 36 have executable test fixtures.

Taking a look at Table 3, rows 1–5 show execution results for Spring-Testing, with four out of five tests being executable and all executable tests that pass. Rows 6–25 show the execution results for PetClinic: 19 out of the 20 tests are executable, and one test fails. Rows 26–39 show the execution results for Alfio, in which 12 out of 16 tests are executable.

Next, we investigate the reasons for generating non-executing tests, and for generating failing tests.

Table 2: Projects used in the evaluation

| Application | Version | #Tests | #Stars | #Forks | #Commits | Scale |
|---|---|---|---|---|---|---|
| Alfio | 2.0.5 | 189 | 1.5K | 2.5K | 3.6K | Large |
| Petclinic | 2.7.3 | 23 | 5k | 15K | 829 | Mid |
| Spring-Testing | 0.0.1 | 13 | 0.8K | 0.4K | 130 | Small |

```
1| @OneToMany
2| private Set<Visit> visits = new LinkedHashSet<>();
3| public Collection<Visit> getVisits() {
4|     return this.visits;
5| }
```

Listing 6: Type conversion failure example

### 4.2.1 Execution Failure Analysis

We have identified five causes for the failure of six tests. We annotated them R1 to R5 and analyze them.

***R1:*** *Passing a class as an argument.* in the method calls of fetchWeatherTest (row 5) and emptyTest (row 32) a class is passed statically as an argument to invoke a method (e.g., for mocking, or initializing a CUT). BTrace only has access to the runtime objects during carving and cannot determine which object is being passed statically.

***R2:*** *Type conversion error.* `getVisitsTest` (row 11) fails to execute because of failing type conversion. More specifically, in this case, hibernate acts as a proxy and changes the object type at runtime. As shown in Listing 6, a return type defined for the MUT (`getVisits`) is a collection that returns a `LinkedHashSet`. However, at runtime, hibernate wraps the object in `PersistentSet` because of the `@OneToMany` annotation. This leads to a type inconsistency at runtime.

***R3:*** *Unable to access fields during trace.* Java 17+ only accesses an object's information under certain conditions; otherwise it requires adding JVM arguments. In our case study with Alfio, we did not add these JVM arguments, which leads to the higher execution failure rate compared to the two other case studies. Examples are `getProviderTest` (row 27) and `getPathLevelTest` (row 33).

***R4:*** *Unable to reproduce an object.* In `emptyTest` (row 32), MICROTESTCARVER failed to reproduce `AlfioMetadata` object with the strategies for unmarshalling. Here, XStream failed to serialize this object, which does not implement `toString()`, and it was not possible to generate that with the guessing approach.

***R5:*** *Private method.* If the method is private, it is not possible to instantiate from this class, and technically, it is out of the project scope to generate tests for private methods. `getPathLevelTest` (row 34) is a private method, and this method cannot be invoked in the class.

### 4.2.2 Test Failure Analysis

We use the Hamcrest matcher (`assertThat` and `is`) for assertions, which internally invokes the `equals` method to compare two objects. In order for the test to pass, the `equals` method needs to be overridden for the CUT, otherwise, it relies on the memory address comparison implementation of `equals` in the `Object` class. `getVisitsTest` (row 10) failed because `equals` was not implemented for `PetType` class, but it passed after implementation.

Examining the coverage of the study subjects gives us useful information even though MICROTESTCARVER is not designed to optimize test coverage (it is not search-based). Figure 6 compares instruction coverage for each application using MICROTESTCARVER , existing manually written tests, EvoSuite-generated tests, and their combinations for each application. As Alfio is quite a large project, we exclude certain classes, such as configuration classes, to measure coverage with a better representation of the core functionality.

**Summary RQ1**. *Our results indicate that 85% of the unit tests that* MICROTESTCARVER *generate from carved information from E2E tests are executable. We have also collected a number of reasons why generated tests might fail to execute.*

Table 3: Experimental results of the generated unit tests on the study subjects using MICROTESTCARVER

| # | Test Method Name | CUT | Executable Fixture | Executable Body | Pass/Fail | Executable Status |
|---|---|---|---|---|---|---|
| 1 | helloWorldTest | ExampleController | True | True | Pass | Executable |
| 2 | helloTest | ExampleController | True | True | Pass | Executable |
| 3 | helloWherePersonTest | ExampleController | True | True | Pass | Executable |
| 4 | weatherTest | ExampleController | True | True | Pass | Executable |
| 5 | fetchWeatherTest | WeatherClient | False/R1 | False/R1 | - | Refinement-Needed |
| | **Spring-Testing Total**: 5 Tests | 2 CUTs | 4/5 (80%) | 4/5 (80%) | 4/4 (100%) | 4/5 (80%) |
| 6 | toStringTest | NamedEntity | True | True | Pass | Executable |
| 7 | getNameTest | NamedEntity | True | True | Pass | Executable |
| 8 | getFirstNameTest | Person | True | True | Pass | Executable |
| 9 | getLastNameTest | Person | True | True | Pass | Executable |
| 10 | getTypeTest | Pet | True | True | Fail | Executable |
| 11 | getVisitsTest | Pet | True | False/R2 | - | Refinement-Needed |
| 12 | getBirthDateTest | Pet | True | True | Pass | Executable |
| 13 | printTest | PetTypeFormatter | True | True | Pass | Executable |
| 14 | printWhereCatTest | PetTypeFormatter | True | True | Pass | Executable |
| 15 | parseWhereBirdTest | PetTypeFormatter | True | True | Pass | Executable |
| 16 | parseWhereCatTest | PetTypeFormatter | True | True | Pass | Executable |
| 17 | parseWhereDogTest | PetTypeFormatter | True | True | Pass | Executable |
| 18 | parseWhereSnakeTest | PetTypeFormatter | True | True | Pass | Executable |
| 19 | parseWhereHamsterTest | PetTypeFormatter | True | True | Pass | Executable |
| 20 | getDateTest | Visit | True | True | Pass | Executable |
| 21 | getNameTest | PetType | True | True | Pass | Executable |
| 22 | toStringTest | PetType | True | True | Pass | Executable |
| 23 | getNameTest | Specialty | True | True | Pass | Executable |
| 24 | toStringTest | Specialty | True | True | Pass | Executable |
| 25 | getVetListTest | Vets | True | True | Pass | Executable |
| | **PetClinic Total**: 20 Tests | 8 CUTs | 20/20 (100%) | 19/20 (95%) | 18/19 (94%) | 19/20 (95%) |
| 26 | getRoleTest | Authority | True | True | Pass | Executable |
| 27 | getValueTest | ConfigurationKeyValuePathLevel | True | True | Pass | Executable |
| 28 | getConfigurationKeyTest | ConfigurationKeyValuePathLevel | True | True | Pass | Executable |
| 29 | getProviderTest | ProviderAndKeys | False/R3 | True | - | Refinement-Needed |
| 30 | getDescriptionTest | EventDescriptionTest | True | True | Pass | Executable |
| 31 | getLocaleTest | EventDescriptionTest | True | True | Pass | Executable |
| 32 | getLocaleTest | Language | True | True | Pass | Executable |
| 33 | getDisplayLanguageTest | Language | True | True | Pass | Executable |
| 34 | emptyTest | AlfioMetadata | False/R3 | False/R4 | - | Refinement-Needed |
| 35 | getEmailAddressTest | ConfirmationEmailConfiguration | False/R3 | True | - | Refinement-Needed |
| 36 | getPathLevelTest | SystemLevel | False/R5 | True | - | Refinement-Needed |
| 37 | getDescriptionTest | LocaleDescription | True | True | Pass | Executable |
| 38 | getLocaleTest | LocaleDescription | True | True | Pass | Executable |
| 39 | getEmailTest | OrganizationContact | True | True | Pass | Executable |
| 40 | getNameTest | OrganizationContact | True | True | Pass | Executable |
| 41 | getStatusTest | TicketReservationStatus... | True | True | Pass | Executable |
| | **Alfio Total**: 16 Tests | 11 CUTs | 12/16 (75%) | 15/16 (93.7%) | 12/12 (100%) | 12/16 (75%) |
| | **Total**: 41 Tests | 21 CUTs | 36/41 (87.8%) | 38/41 (92.6%) | 34/35 (97%) | 35/41 (85.3%) |

## 4.3 RQ2: Understandability of the carved tests vs EvoSuite tests

In order to answer RQ2, we applied EvoSuite to the three case study subjects to compare these EvoSuite-generated tests with the carved tests, amongst others in terms of understandability. We have first established that EvoSuite fails to generate some of the carved tests, likely because it does not have access to dynamic analysis information. When we further investigate the tests generated by EvoSuite, we observe that EvoSuite typically generates test data with either random data, empty strings or fields, null, or mock objects. For example, for *String* and *Integer* types, EvoSuite generates random values, and for a type like Date, it mocks a current date/time. More specifically, in Table 4 we compare EvoSuite-generated tests with carved tests that test the same methods. For Spring-Testing, we observe that EvoSuite does not generate any tests for the classes under test that MICROTESTCARVER generates test for (`ExampleController` and `WeatherController`); we think this is due to the difficulty to mock complex objects, and we do observe EvoSuite-generated tests for CUTs `Person` and `Weather`.

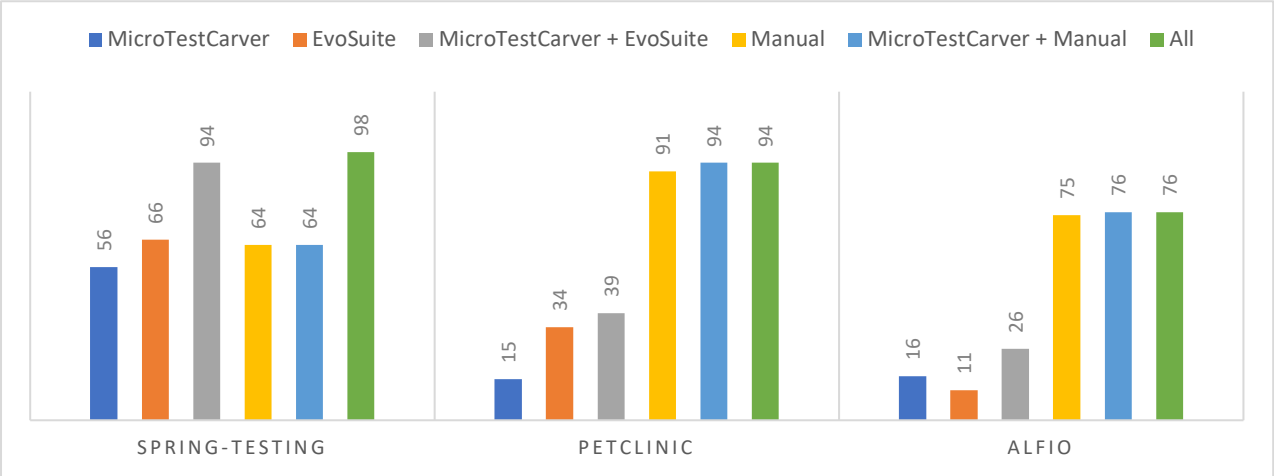



Figure 6: Instruction coverage for each study subject

```
A | new WeatherResponse("Jy(+XI9N", "Jy(+XI9N")
-------------------------------------------
B | new WeatherResponse("Clouds", "few clouds")
```

Listing 7: WeatherResponse object generated by EvoSuite (A) in comparison to MICROTESTCARVER (B)

Table 4: Results from EvoSuite tests corresponded to those from Carved tests

| Application | #Carved Tests | #Generated By EvoSuite | Test Data | | |
|---|---|---|---|---|---|
| | | | Random | Null | Empty |
| Spring-Testing | 4 | 0 | - | - | - |
| Petclinic | 19 | 10 | 4 | 3 | 2 |
| Alfio | 12 | 2 | 2 | 0 | 0 |

Subsequently, for the projects where tests could be generated, we note the different construction methods for test data, in particular, how many times random, null, or empty values are used by EvoSuite. Creating objects with random fields is not meaningful for a class like `WeatherResponse`. Listing 7 illustrates an example of an object EvoSuite and MICROTESTCARVER generate for the `WeatherResponse` class.

In PetClinic, EvoSuite could generate 10 out of 19 carved tests. For producing test data, EvoSuite generated four tests with random data, three used null assertion (null), and two were created without setting their fields (empty).

In order to illustrate the difference between the carved tests and EvoSuite tests, we randomly selected one test that we show in Listing 8: the first test is generated by EvoSuite, and the second one is generated by MICROTESTCARVER . EvoSuite used an `assertNull` on a `petType` that was not set. MICROTESTCARVER on the other hand created a *"cat"* as a petType and asserts on that.

In order to apply EvoSuite on *Alfio*, we had to downgrade the version of Alfio from 2.0-5 to 2.0-M4-2204, because EvoSuite does not support Java 17. We executed EvoSuite four times to aggregate the generated tests. EvoSuite had difficulties with mocking tests, and we think that is why their coverage in this particular project is low (11%). We discovered two similar tests among the carved tests and EvoSuite-generated tests, and in both cases, EvoSuite utilized random data. As an example, in the test case that tests the functionality of language, the carved test used "English" as test data; in contrast, EvoSuite used "alfio.controller.api.v2.model.Language", which is a string without meaning in the context.

```
@Test
public void testCreatesPetType()  throws Throwable {
    PetType petType0 = new PetType();
    assertNull(petType0.getName());
}
-------------------------------------------------------
public void setUp() throws Exception {
    subject = new PetType();
    subject.setName("cat");
    subject.setId(1);
}
@Test
public void getTypeTest() throws Exception {
    PetType getType = subject.getType();

    PetType petType = new PetType();
    petType.setId(1);
    petType.setName("cat");

    assertThat(getType, is(petType));
}
```

Listing 8: First test is a test generated by EvoSuite, and the second one is generated by MICROTESTCARVER .

Table 5: Results from manual tests corresponded to those from Carved tests

| Application | #Carved Tests | #Manual Test | Understandability | | |
|---|---|---|---|---|---|
| | | | Better | Similar | Poorer |
| Spring-Testing | 4 | 4 | 0 | 4 | 0 |
| Petclinic | 19 | 8 | 0 | 4 | 4 |
| Alfio | 12 | 2 | 1 | 1 | 0 |

When comparing the EvoSuite-generated tests with the MICROTESTCARVER tests, we found that test data plays an essential role in understanding a test case. In particular, with random, null, empty, or mocked inputs it is harder to understand the logic and purpose of a test. Furthermore, search-based test generators like EvoSuite fail to mock some methods because they do not have runtime information access. Nevertheless, search-based approaches are capable of generating tests for corner cases and are good at increasing coverage. Also, the tests that they generate are shorter, because they use minimization of the test case as a secondary search-objective.

**Summary RQ2**. *When we compare carved tests with EvoSuite-generated tests, we observe that the use of actual test data which is derived from E2E in carved tests makes the test easier to understand and more meaningful. Search-based approaches are good at generating short test cases; the entire test suite typically has high coverage.*

## 4.4 RQ3: Understandability of the carved tests vs manual tests

We re-used the methodology that we used for RQ2 to compare the carved tests with existing manually written tests in terms of understandability. We collected the manual tests that match the carved tests, i.e., they test the same method. We then compare them in terms of test data, naming variables and tests, and line numbers; the results are illustrated in Table 5.

```
@Test
void shouldParse() throws ParseException {
  given(this.pets.findPetTypes()).willReturn(makePetTypes());
  PetType petType = petTypeFormatter.parse("Bird", Locale.ENGLISH);
  assertThat(petType.getName()).isEqualTo("Bird");
}
private List<PetType> makePetTypes() {
  List<PetType> petTypes = new ArrayList<>();
  petTypes.add(new PetType() {
    { setName("Dog"); }
  });
  petTypes.add(new PetType() {
    { setName("Bird"); }
  });
  return petTypes;
}
```

```
public void parseWhereBirdTest() throws Exception {
  PetType PetType = new PetType();
  PetType.setId(5);
  PetType.setName("bird");
  ArrayList<PetType> petTypes = new ArrayList<>();
  petTypes.add(PetType);

  given(owners.findPetTypes()).willReturn(petTypes);

  PetType parse = subject.parse("bird", Locale.ENGLISH);

  PetType PetType_1 = new PetType();
  PetType_1.setId(5);
  PetType_1.setName("bird");

  assertThat(parse, is(PetType));
}
```

Listing 9: Comparison of a manual test (A) with a carved test (B)

In Spring-Testing, the understandability of the manual and the carved tests are equal. Both use meaningful test data and assertions. However, the manual tests covered a broader range of scenarios, such as when an error was encountered in a method. In PetClinic, three tests are understandable for both carved ones and manual tests. In four tests, manual tests are more understandable because of using private factory methods and better naming for variables and methods. An example of this comparison in which the manual test is more understandable than the carved one is shown in Listing 9. Although both tests use meaningful test data, and the logic of the test is understandable, the manually written test is more structured and has less duplication in comparison to the carved one; the manual test for mocking `petType` list uses the `makePetTypes()` factory method [113], making it more readable since it does not need to examine the factory method internally and is also reusable when repeated.

Overall, in Alfio we got a mixed image of the understandability of tests. More specifically, we have found two pairs of matching tests. When comparing these pairs, we observe that the manually written tests are overly long and try to test too much (so-called *eager tests* [128]). When looking beyond these two tests, we have observed that parameterized tests are used in several manually written tests, which makes them shorter and easier to understand.

**Summary RQ3**. *Comparisons between carved and manually written tests indicate that using data derived from E2E tests will bring the carved test closer to the manually written test in terms of understandability. However, we observe that some manually written tests use some best practices, e.g., parameterized tests or a factory method, which enhances the readability of manually written tests in some situations.*

## 4.5 Threats to validity

Threats to construct validity concern how we make our observations. As we have performed a largely exploratory analysis using a manual and subjective evaluation, we acknowledge this threat to validity. In future work, we will apply a more structured evaluation in the context of a user study to determine the understandability of tests.

Threats to external validity are related to whether we can generalize our findings. While we examined test cases from three different subject systems that vary in size and domain, the fact that we only examined 41 test cases in our exploratory study means that we cannot claim generalizability. In future work, we will extend our investigation to more subject systems and more test cases.

# 5 Evaluation: Adversarial Example Generation for the Vision Components of Cyber-Physical Systems

We introduce two variants of DE: (1) a single-objective variant (`Pixel-SOO`) that steers for pixel-based input changes to cause an output prediction to change gradually; (2) a multi-objective variant (`Pixel-MOO`) that additionally seeks to minimize the number of pixel modifications made to the original input image. Then, we conduct a preliminary study focused on answering the following research questions:

**RQ1** : *How do `Pixel-SOO` and `Pixel-MOO` perform compared to the state-of-the-art `BMI-FGSM` in generating adversarial attacks?*

**RQ2** : *What are the strengths and weaknesses of `Pixel-SOO` and `Pixel-MOO`?*

To answer our RQs, we run our approach with 5 different, well-known deep computer vision models from the `Keras` python library. We chose `VGG16` and `VGG19` which are both classified as "very deep" convolutional neural networks for large-scale image recognition[3] [154], that got a canonical status due to their strong performance in the `ImageNet` benchmark challenges[4]. VGG16 was one of the best-performing models in the 2014 ILSVRC challenge and achieves 92.7% top-5 test accuracy on the ImageNet dataset. VGG16 and VGG19 both consist of 3x3 convolutional layers stacked on top of each other in increasing depth, with VGG16 having 16 convolutional layers, and VGG19 being 'deeper' with 19 convolutional layers. Furthermore, `ResNet50`, `ResNet101` and `ResNet152` all are based on deep residual learning for image recognition[5] [72].

In our experiments, we use the ImageNet pre-trained weights released by the original authors after training on the ILSVRC2012 training set, as released through Keras [33]. Note that despite these details, we consider all models as a black-box, given the fact that our approach (and the baseline) does not need access to the model internals.

[3] `https://keras.io/api/applications/vgg/`
[4] `https://image-net.org/challenges/LSVRC/index.php`
[5] `https://keras.io/api/applications/resnet`

## 5.1 Dataset

For our experiments, we sample 50 images from the ImageNet ILSVRC2012 Validation data set [148]. We chose the validation set, due to a lack of ground-truth data availability for the test set. First, we draw an initial pool of images for the input by selecting 1000 random images from the ImageNet validation data folder. After pre-processing the images to have a $224 \times 224$ input size, we then choose whether to drop or retain the image based on the prediction confidence by the VGG19 model and class uniqueness. As for the prediction confidence, if the correct ground truth label is predicted with confidence between $0.8$ and $0.9$, we consider the image to be a valid image for our experiments: as the ground truth label is recognized with high confidence, we can be confident it is visually distinguishable and not ambiguous w.r.t. other classes; at the same time, for too high confidence, it may be too obviously one particular image class, and flipping may as a consequence be hard. By retaining only one image per object class, we also ensure a reasonably diverse set of images.

## 5.2 Implementation and Parameter settings

We have implemented the different DE-based approaches in `Pymoo` v0.5.0 [20], using Python 3.10 and `Keras` 2.2.2. Pymoo [20] is an open-source framework that allows us to easily adapt the simple DE and the NSDE for generating adversarial attacks. Runs are evaluated inside a Docker container on an AMD EPYC 7713 64-Core Processor running at 2.6 Ghz and with 256 available CPUs. We had 3 Nvidia A40 GPUs each with 48 GB GDDR6 running CUDA version 11.6 available to us. The Dockerfile in our implementation can be rebuilt on any system, easily modifying the CUDA container for a different system. Our implementation is available on Zenodo:

*https://doi.org/10.5281/zenodo.7741267*

Due to issues with running the code from `BMI-FGSM`, we were not able to execute it using CUDA[117] (GPU). This meant we could only run the baseline on CPU, which has likely resulted in a slower execution time compared to our multi-objective results which were executed on GPU.

### 5.2.1 Parameters setting

We set both the multi-, single-objective DE, and `BMI-FGSM` to evolve a population size of 20 over a maximum of 400 generations. When a test image has been wrongly predicted, we kill the test and allow it to run for five more generations, so better fronts may still be found. For the parameter settings, we have chosen the same values as suggested in the literature [111, 43].

We use the variation operators with a crossover rate $CR = 0.9$ and scaling factor $F = 0.8$, which are the recommended values in the literature [111]. For both algorithms, solutions/attacks are selected for reproduction using the binary-tournament selection [43]. For `Pixel-SOO`, the binary selection is based on the single-objective value to optimize (i.e., $O_1$). Instead, in `Pixel-MOO`, the selection relies on dominance to decide which solution wins each tournament round. Finally, we opted for a relatively small population size $p = 20$ (smaller than $p$=100 used in other studies [99, 111]) as suggested in the literature from problems with expensive objective computation [34].

## 5.3 Study Design

To answer our first research question, we compare our multi-objective approach `Pixel-MOO` to our single-objective approach `Pixel-SOO` (only optimizing for $O_1$); next to this, we compare our single-objective approach to the `BMI-FGSM` method by Lin et al.[99]. In this, we try to stay as close to the implementation by the authors as possible; as a consequence, beyond the parameters setting population size and generations, we do not modify the authors' codebase[6]. The `BMI-FGSM` codebase only supports VGG16, ResNet50 and ResNet101;

[6] `https://github.com/jylink/BMI-FGSM`

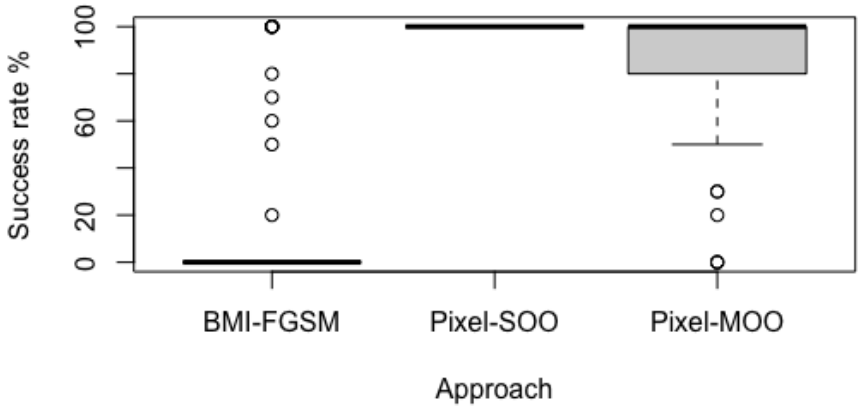



Figure 7: Success rate of `BMI-FGSM`, `Pixel-SOO` and `Pixel-MOO` in flipping the predictions for VGG16.

here, we started our experiments with VGG16. We execute `BMI-FGSM`, `Pixel-SOO`, and `Pixel-MOO` against each image in our dataset to generate adversarial examples. For each image, we run each algorithm 10 times, to account for their random nature. As a consequence, with 50 images, the end result is a total of 1500 test runs (500 for each method). For each of the `Pixel-SOO` and `Pixel-MOO` executions, a new random seed was generated and stored for future replications, together with the results of the generated attacks. For `BMI-FGSM` a slight modification was made to the codebase to output the current prediction data. This data is stored for each run, along with the adversarial image.

For evaluation, we consider two performance metrics: (1) the success rate, indicating the percentage out of the 10 runs for which `Pixel-SOO` and `Pixel-MOO` were capable of causing a change in prediction output, and (2) how many pixels needed modification (i.e., how many tuples are in mask $X$) in the best solution. For the comparison, we considered the best solution/attack in the final population (last generation) of `Pixel-SOO`. Instead, `Pixel-MOO` provides a set of non-dominated solutions (front) rather than one single solution. For our analysis, among all solutions/attacks that lead to flipping the prediction (i.e., those with negative values for the first objective $O_1$), we have chosen the one with the lowest number of changed pixels (second objective $O_2$).

To compare `BMI-FGSM` and `Pixel-MOO`, we analyze the success rate in flipping the prediction and the median number of changes required to flip the model's prediction over the 10 runs. We also apply statistical analysis to further assess whether the observed differences are significant or not. We use Fisher's exact test [56] for the success rate, considering the results of each run (for each algorithm) as a binary/dichotomy outcome (i.e., the prediction was flipped or not). To assess the significance of the differences among `Pixel-SOO` and `Pixel-MOO` w.r.t. the number of altered pixels, we use the Wilcoxon rank sum test [38]. For both statistical tests, we use a confidence level $\alpha = 0.95$. Furthermore, we complement the test for significance with the Vargha-Delaney statistic ($\hat{A}_{12}$) to measure the effect size of the results [165].

## 5.4 Results

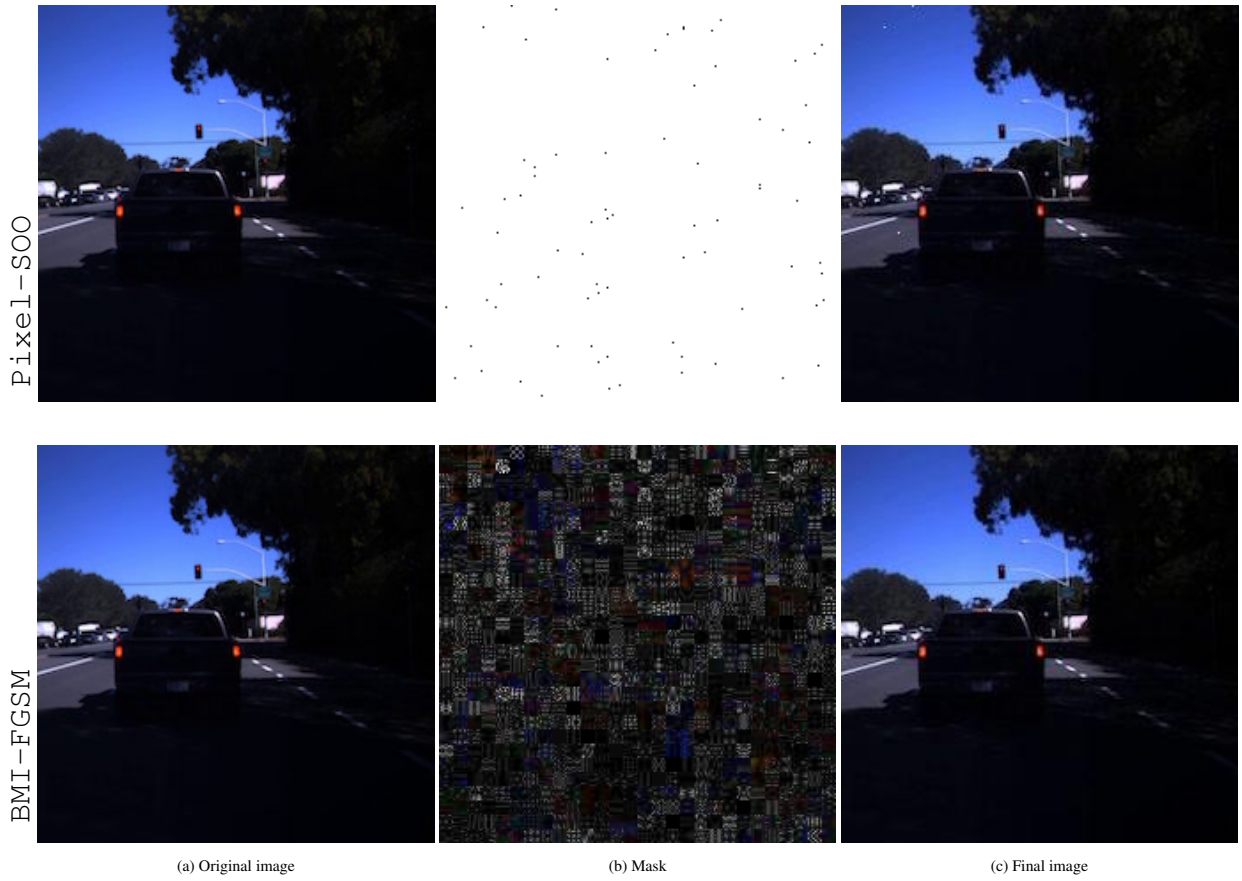
### 5.4.1 Results on VGG16

A full breakdown of the VGG16 results can be seen in Figure 7, which depicts the success rate of `BMI-FGSM`, `Pixel-SOO` and `Pixel-MOO` in creating adversarial examples for the 50 images in our experiment. As we can observe, `BMI-FGSM` was rarely able to flip the predictions for VGG16 (median, second, and third quartiles being equal to zero), while our approaches based on pure DE can do so for all images. By comparing `Pixel-SOO` and `Pixel-MOO`, we observe that the former always achieves a 100% success rate while the latter achieves a lower success rate in 20% of the images. However, both our DE-based approaches can generate an adversarial attack in at least one of the ten repetitions.

| Seed | Perturbations | | | # Generations | | | Runtime (seconds) | | |
|---|---|---|---|---|---|---|---|---|---|
| Image | BMI | P-Soo | P-Moo | BMI | P-Soo | P-Moo | BMI | P-Soo | P-Moo |
| 00323 | 4325 | 25 | 14 | 240 | 10 | 18 | 2850 | 73 | 98 |
| 00344 | - | 16 | 13 | - | 6 | 12 | - | 57 | 77 |
| 00624 | - | 55 | 51 | - | 21 | 42 | - | 126 | 237 |
| 01204 | - | 39 | 29 | - | 13 | 22 | - | 87 | 124 |
| 02431 | 10620 | 19 | 18 | 20 | 10 | 24 | 227 | 74 | 123 |
| 05053 | - | 155 | 129 | - | 65 | 180 | - | 336 | 847 |
| 05412 | - | 65 | 36 | - | 34 | 128 | - | 188 | 631 |
| 05929 | - | 76 | 83 | - | 30 | 71 | - | 168 | 371 |
| 06213 | - | 159 | 92 | - | 76 | 184 | - | 393 | 1897 |
| 07160 | - | 70 | 76 | - | 27 | 70 | - | 153 | 336 |
| 08024 | - | 35 | 39 | - | 19 | 44 | - | 117 | 247 |
| 09335 | - | 17 | 11 | - | 9 | 11 | - | 69 | 72 |
| 09485 | - | 137 | 96 | - | 70 | 297 | - | 363 | 1749 |
| 09654 | - | 65 | 46 | - | 31 | 111 | - | 175 | 532 |
| 09931 | - | 36 | 22 | - | 11 | 22 | - | 83 | 104 |
| 11100 | - | 24 | 11 | - | 9 | 17 | - | 73 | 88 |
| 11177 | - | 69 | 55 | - | 26 | 68 | - | 150 | 326 |
| 11189 | - | 30 | 19 | - | 9 | 15 | - | 72 | 88 |
| 12820 | - | 114 | 100 | - | 43 | 95 | - | 234 | 473 |
| 14504 | - | 79 | 72 | - | 33 | 139 | - | 184 | 710 |
| 15116 | 7391 | 30 | 31 | 240 | 17 | 55 | 2852 | 105 | 292 |
| 15739 | - | 57 | 33 | - | 25 | 62 | - | 144 | 324 |
| 16615 | - | 39 | 32 | - | 12 | 24 | - | 83 | 116 |
| 20018 | NA | 43 | 39 | NA | 25 | 82 | NA | 147 | 398 |
| 20889 | - | 112 | 102 | - | 65 | 237 | - | 335 | 1138 |
| 23090 | - | 162 | 134 | - | 70 | 217 | - | 360 | 1053 |
| 23203 | 10246 | 151 | 140 | 320 | 64 | 286 | 3763 | 334 | 1381 |
| 23729 | - | 82 | 56 | - | 39 | 123 | - | 214 | 609 |
| 24130 | - | 41 | 24 | - | 27 | 77 | - | 151 | 406 |
| 25633 | - | 127 | 121 | - | 47 | 97 | - | 248 | 472 |
| 26738 | - | 67 | 61 | - | 22 | 41 | - | 132 | 219 |
| 27242 | 4885 | 12 | 6 | 320 | 4 | 8 | 3797 | 48 | 48 |
| 29251 | 6593 | 26 | 17 | 320 | 12 | 23 | 3790 | 84 | 124 |
| 30019 | - | 64 | 51 | - | 23 | 57 | - | 136 | 285 |
| 30959 | 4774 | 38 | 32 | 120 | 17 | 30 | 1449 | 106 | 160 |
| 32263 | 12507 | 23 | 15 | 80 | 7 | 12 | 966 | 61 | 67 |
| 32576 | - | 81 | 72 | - | 38 | 101 | - | 205 | 513 |
| 34966 | - | 45 | 39 | - | 17 | 34 | - | 109 | 182 |
| 35091 | - | 34 | 24 | - | 11 | 21 | - | 80 | 105 |
| 35614 | - | 83 | 59 | - | 45 | 155 | - | 235 | 956 |
| 36183 | 15992 | 26 | 22 | 0 | 10 | 16 | 44 | 76 | 104 |
| 38221 | - | 37 | 20 | - | 60 | 93 | - | 307 | 1891 |
| 41173 | - | 101 | 91 | - | 94 | 241 | - | 463 | 1523 |
| 41842 | - | 173 | 129 | - | 81 | 377 | - | 408 | 1903 |
| 43541 | - | 46 | 36 | - | 19 | 32 | - | 120 | 180 |
| 44582 | - | 113 | 119 | - | 51 | 201 | - | 267 | 1005 |
| 44788 | - | 80 | 24 | - | 61 | 61 | - | 313 | 1919 |
| 46372 | 11463 | 35 | 28 | 40 | 14 | 30 | 505 | 95 | 156 |
| 46979 | - | 49 | 39 | - | 18 | 31 | - | 111 | 158 |
| 48219 | 4244 | 24 | 16 | 320 | 8 | 12 | 1180 | 66 | 71 |
| Mean | 8872 | 67 | 54 | 178 | 32 | 90 | 1857 | 176 | 547 |

Table 6: Median number of pixel changes, generations, and running time required by all approaches to generate adversarial attacks on VGG16.

To better understand the time needed to converge and how many pixels have been changed by the different algorithms, we report in Table 6 the detailed results for each image in our benchmark. In particular, we report




(a) Original image (b) Mask (c) Final image

Figure 8: Pixel-SOO vs. BMI-FGSM results for the image of a traffic light

the running time, the number of changes (altered pixels), and the generations required for flipping our image prediction. In Table 6, "–" entries indicate that the corresponding algorithm could not generate an adversarial attack within the search budget. We can observe that BMI-FGSM successfully generated adversarial attacks (at least in one out of 10 runs) for 11 images out of 50. To do so, it changed thousands of pixels (8872 on average) and up to 15992 pixels for the image 'ILSVRC2012_val_00036183.JPEG' (in short, image id = 36183 in Table 6). The image 'ILSVRC2012_val_00020018.JPEG' is an interesting case since BMI-FGSM threw an error when loading the image (highlighted with $NA$ values in Table 6).

Instead, both Pixel-SOO and Pixel-MOO were able to generate adversarial attacks for all the images in at least 1 of the ten individual runs. Indeed, there is no "-" entry in Table 6 for these two algorithms. W.r.t. the number of introduced changes, we can observe that both algorithms required to change fewer pixels than the state-of-the-art BMI-FGSM. Pixel-SOO changed on average 67 pixels while Pixel-MOO generated successful attacks by changing even fewer pixels (54 pixels on average).

To better understand the type of attacks generated by Pixel-MOO and the baselines BMI-FGSM, Fig. 8 shows the results on detecting traffice lights. This is one of the few images for which BMI-FGSM could successfully generate an adversarial attack. The original image was ground-truthed and classified as a 'traffic lights' (920). When initially running the image through BMI-FGSM, the classification was shown as 'limousine'. This is an incorrect classification and may be due to the pre-processing BMI-FGSM applies. However, it was then able to perform a prediction flip to 'traffic lights' (920). while Pixel-SOO successfully flipped it from the correct class ('traffic light') to a 'limousine' (58). Fig. 8 depicts (1) the original image, (2) the 'perturbation mask', and (3) the resulting adversarial attacks.

We can observe that, for BMI-FGSM, the median amount of changed pixels is 4325 and for Pixel-SOO is 25, which is a remarkable difference. From Fig. 8, we can barely see any modification in the change matrix, and this is because BMI-FGSM subtly changes many pixels. Our method, however, stands out for the smallest

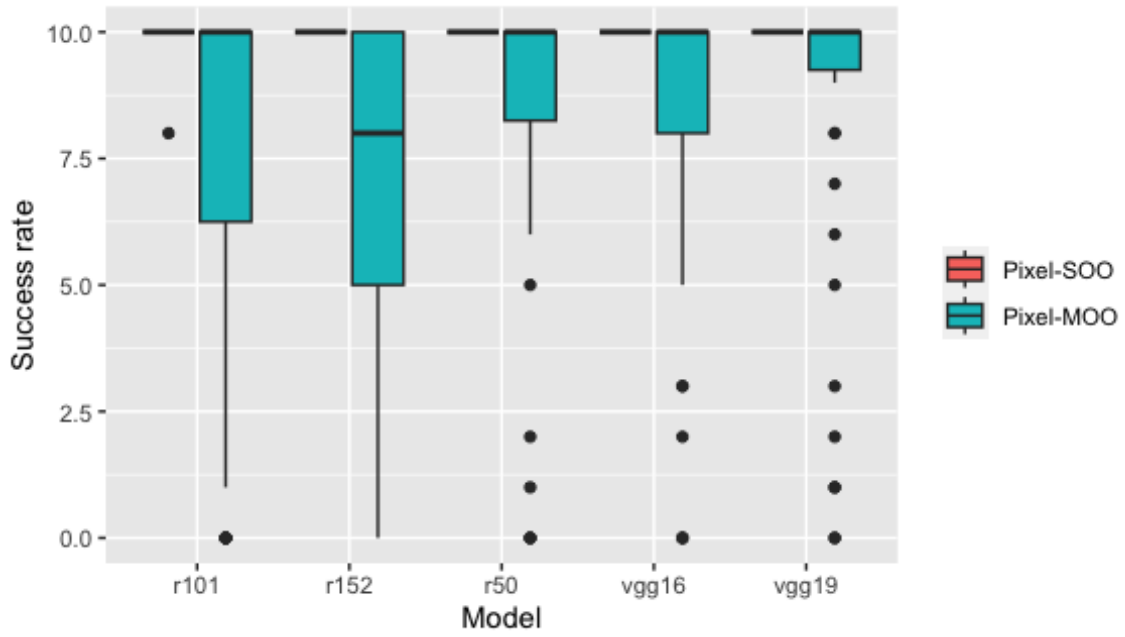



Figure 9: Success rate of both approaches on each model

amount of perturbations which involves making larger color differences to the pixel. However, as we can see by the matrix and the final image, the changes applied are still very subtle.

On top of having considerably fewer perturbations and a better success rate, we observe from Table 6 that the number of generations required by `Pixel-SOO` and `Pixel-MOO` is also much lower than the generations needed by `BMI-FGSM` for converging. `BMI-FGSM` took on average 240 generations to reach a prediction flip, while our `Pixel-SOO` method took only 10, once again being considerably quicker.

### 5.4.2 Results on other models

The original `BMI-FGSM` implementation (which we have reused) was not compatible with the other models except for the ResNet models. However, in our experiment, `BMI-FGSM` took a considerable amount of time to complete just a single run (often over 7 hours) and often could not generate any adversarial example within the search budget. In the following, we therefore only report the results for the two DE-based approaches proposed in this report, namely `Pixel-SOO` and `Pixel-MOO`.

Figure 9 depicts boxplots for the success rate over the different runs of `Pixel-MOO` and `Pixel-SOO` for the five DNN models in our study. As we can observe, `Pixel-SOO` is successful almost 100% of the time, with some outliers only for ResNet101. `Pixel-MOO` still achieves a 100% success rate for more than 50% of the images.

These differences are also confirmed by Fisher's exact test, of which the results are reported in Table 7. More specifically, `Pixel-MOO` and `Pixel-SOO` are statistically equivalent in terms of success rate for the large majority of the images (success rate over ten runs) and across all models. For around 20% of images, the single-objective variants statistically outperform the multi-objective variant. For ResNet101 in particular, `Pixel-SOO` frequently has a higher success rate compared to its multi-objective counterpart. As `Pixel-MOO` searches for trade-offs between prediction flip ($O1$) whilst preserving as many original pixel values as possible ($O2$), we hypothesize it may be slower to reach optima in comparison to `Pixel-SOO`, and may perform better when a larger search budget (i.e., more generations) would be used.

Finally, we compare the attacks generated by `Pixel-MOO` and `Pixel-SOO` w.r.t. the number of pixel perturbations injected in the seed images. Table 8 reports the results of the Wilcoxon rank sum test and the

Table 7: Number of times one approach outperforms the other according to the Fisher's test ($p$-value$<$0.05) w.r.t. the Success Rate

| Model | `Pixel-MOO` wins | Equal | `Pixel-SOO` wins |
|---|---|---|---|
| r50 | - | 44 | 6 |
| r101 | - | 33 | 17 |
| r152 | - | 42 | 8 |
| vgg16 | - | 39 | 11 |
| vgg19 | - | 43 | 7 |

Table 8: Number of times one approach outperforms the other according to the Wilcoxon test ($p$-value$<$0.05) and the $\hat{A}_{12}$ statistics w.r.t. the number of changed pixels.

| Model | `Pixel-MOO` wins | | | Equal | `Pixel-SOO` win | | |
|---|---|---|---|---|---|---|---|
| | Large | Medium | Small | | Small | Medium | Large |
| r50 | 28 | 7 | - | 15 | - | - | - |
| r101 | 33 | 3 | - | 13 | - | 1 | - |
| r152 | 28 | 3 | - | 19 | - | - | - |
| vgg16 | 26 | 6 | - | 18 | - | - | - |
| vgg19 | 29 | 7 | - | 14 | - | - | - |

$\hat{A}_{12}$ statistics. The results indicate that the adversarial attacks generated by `Pixel-MOO` contain fewer pixel alterations than those produced by `Pixel-SOO`, and the results are statistically significant in more than 60% of the comparison, with an effect size being *large* in the large majority of the significant cases.

Therefore, we can draw some general conclusion: `Pixel-MOO` and `Pixel-SOO` provide two different trade-offs concerning the speed and magnitude of the image perturbation. `Pixel-SOO` is faster by producing attacks with less subtle changes. Instead, `Pixel-MOO` produced more subtle attacks but requires more time to converge.

## 5.5 Threats to validity

For the work presented in this report, several threats to validity can be identified.

**Construct validity.** While we assume our adversarial examples have perceptually visible changes, but these changes are small enough to not change the object of focus (i.e. the ground truth) for an image, we do not formally validate this in human experiments. Thus, it is possible that found adversarial examples may be degraded such, that a change in ground truth label would be needed and justifiable. However, in our current experiments, we already see many prediction flips happening when changing less than 200 (so less than 0.3%) out of 50,176 pixels, making it unlikely that the object of focus would be completely obscured by the mutations. Next to this, we validated the model's initial ground truth label against the data supplied from the ImageNet website. Despite this, the model may have an erroneous initial prediction, and even if it would be correct in comparison to the ground truth, plausible labels beyond the indicated ground truth may exist if the visual scene is complex or semantically ambiguous [98]. Thus, while we frame our technique as one creating adversarial attacks, it cannot be guaranteed that we necessarily push the model towards being wrong. Still, the aspect about adversarial attacks that keeps holding is that subtle changes lead to different prediction outcomes; as such, a flip in output prediction still is an indicator of the model not giving robust predictions.

**Internal validity.** While our image selection procedure yielded a random draw of images from 50 unique classes, the ILSVRC2012 classes semantically are not uniformly distributed (e.g. having multiple classes with sub-species of dogs). Future sampling strategies could seek to more explicitly mitigate for this.

Our test validates both SOO and MOO using the same minimise function. The only variation for each example is the problem, with one using the Genetic Algorithm and the other `NSGA-II`. We evaluate both on the same fitness function but also filter our fitness on the matrix size within `NSGA-II`. Changes to the fitness evaluation

could create threats to internal validity and future work could see a deeper connection between the two evaluation methods to ensure changes are applied to both from one singular method.

**External validity.** Currently, our approach was only was tested against (the state-of-the-art) `BMI-FGSM`. It will be worthwhile to also test it against further attack approaches, such as the one-pixel attack [159]. Furthermore, beyond our current set of DNN models, more canonical models exist that can be studied, such as Inception-v3 [162].

Threats to external validity could come from the `Pymoo` methods we employ for our testing. Breaking changes applied to the minimize, genetic algorithm and, or `NSGA-II` could have detrimental effects to our data. Beyond that, any image we choose to test has the potential to be a threat to external validity. If the image is far too complex and requires a significant mutation to disrupt the `DL` model then it no longer falls under our test case pass of minimal changes.

**Conclusion validity.** In some runs, `Pixel-MOO` fails to find an adversarial example; further optimizations w.r.t. population and generation size may be needed.

# 6 COSMOS Requirements, Integration Status & Summary of Future Work

## 6.1 Status of Integration & Requirements Coverage of Tools in each Use case and & Next Steps

This section details (with tables), for each innovation area and tool, how each tool covers the COSMOS requirements as well as the status of integration steps. At the end of that section, we also summarize the project's next steps.

### 6.1.1 Refactoring Framework

Table 9: COSMOS Requirement Coverage Overview

**7.1 CI/CD Pipelines**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U1 | COSMOS can be executed in one or more Docker container(s) | YES | • PyRock: The repository contains a Dockerfile and instructions on how to run it with Docker.<br>• AP-Spotter: The repository contains a Dockerfile and instructions on how to run it with Docker. |
| U2 | COSMOS provides outputs and tools results in a human-readable format | YES | • PyRock: The tool outputs a .md file for analysis.<br>• AP-Spotter: The tool first outputs the results in a .csv file, after which the tool generates a .md file. The .md file is written in a human-readable format for the developers to use. |
| U3 | COSMOS provides outputs and tools results in a machine-readable format for further processing | YES | • PyRock: Each separate output state of the tool, can be analyzed by other tools.<br>• AP-Spotter: The tool first outputs the results in a .csv file, after which the tool generates a .md file. The .csv file is made for the possibility to have the results parsed by different tools. |
| U4 | COSMOS prevents application components that are not released by a gatekeeper in change management from being available in later pipeline stages | NOT APPLICABLE | |
| U5 | COSMOS used in change management is able to evaluate the impact of a changed software component regarding number of affected CPS, scope and which stakeholders/roles to inform about the change | NOT APPLICABLE | |
| U6 | COSMOS is able to support the rapid deployment of new adaptations | NOT APPLICABLE | |
| U7 | COSMOS provide results in a comparable way between adaptations (e.g. history) | YES | • PyRock: The tool is versioned with Git.<br>• AP-Spotter: The tool is versioned with Git. |
| U8 | COSMOS is able to support testing based on data models | NOT APPLICABLE | |
| U9 | COSMOS is able to devise a test strategy for system upgrades | NOT APPLICABLE | |
| U10 | COSMOS tools are able to work within the existing inhouse CI/CD pipeline and test infrastructure | YES | • PyRock: Supported by the use of Docker.<br>• AP-Spotter: Supported by the use of Docker. |

| | | | |
|---|---|---|---|
| U11 | COSMOS supports standalone execution of tools (outside development flow not linked to code change) | YES | • PyRock: The tool can be run by itself.<br>• AP-Spotter: The tool can be run by itself. |
| U12 | COSMOS can act as a gate keeper in the Bundle Pipeline by checking test results against software quality requirements | NOT APPLICABLE | |
| U13 | COSMOS provides a test management infrastructure to balance test executions over testing infrastructures (e.g. scaling and model type testing) | NOT APPLICABLE | |
| U14 | COSMOS provides integration with GitLab | NO | • PyRock: The tool does not provide integration with GitLab.<br>• AP-Spotter: The tool does not provide integration with GitLab. |
| U15 | COSMOS is able to check for the proper signing of software components (e.g. OSGi bundles) | NOT APPLICABLE | |

**7.2 Bad Practices Detection and Anti-Patterns**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U16 | COSMOS provides tools for CI/CD best practices and anti-patterns | NOT APPLICABLE | |
| U17 | COSMOS is able to track best practices and antipatterns | YES | • PyRock: Not applicable.<br>• AP-Spotter: The tool detects Software Performance Antipatterns, specific for CPS. |
| U18 | COSMOS provides detectors for configuration, code and test smells (e.g. from static analysis, heuristics, etc.) | YES | • PyRock: Not applicable.<br>• AP-Spotter: The tool uses static analysis to detect Software Performance Antipatterns, specific for CPS, in the code under analysis. |
| U19 | Processing in COSMOS is sufficiently automated to avoid selective automation and user interventions | PARTIALLLY | • PyRock: After selecting which modules and projects the tool needs to analyze, there are a few steps that need to be taken before the final result is presented. This tool returns information about interesting self-admitted commits.<br>• AP-Spotter: After selecting which modules the tool needs to analyze, there are no manual tasks further required. |

**7.3 Simulators and HiL**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U20 | COSMOS is able to automatically generate tests cases that effectively explore the viable inputs for a SIL environment | NOT APPLICABLE | |
| U21 | COSMOS is able to automatically generate simulation scenarios for HIL testing | NOT APPLICABLE | |
| U22 | COSMOS supports Simulators and HIL for unit tests | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U23 | COSMOS supports Simulators and HIL for unit integration tests | NOT APPLICABLE | |
| U24 | COSMOS supports simulation tools executing on Linux PPC or x86 platform | NOT APPLICABLE | |
| U25 | COSMOS is able to be configured to use the available API to control test executions on testing infrastructure | NOT APPLICABLE | |
| U26 | COSMOS is able to run software on emulated XMEGA | NOT APPLICABLE | |
| U27 | COSMOS supports interaction with the testing infrastructure over a local network | NOT APPLICABLE | |
| U28 | COSMOS is able to automatically generate simulation scenarios (e.g. in BeamNG.research, etc.) to provide CAN signals for HIL testing | NOT APPLICABLE | |

**7.4 Automated Testing**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U29 | COSMOS is able to generate test reports for groups of CPS or aggregate reports of multiple CPS | NOT APPLICABLE | |
| U30 | COSMOS is able to compare different test results and to provide developer feedback pointing out major differences | NOT APPLICABLE | |
| U31 | COSMOS is able to work with test storage and management facilities to allow assignability to a specific version of CPS or software component | NOT APPLICABLE | |
| U32 | COSMOS provides configurability to select individual criteria for each tested software component | YES | |
| U33 | COSMOS test results contain meta data related to the used hardware/device/CPS | NOT APPLICABLE | |
| U34 | COSMOS provides facilities for monitoring information from the system | NOT APPLICABLE | |
| U35 | COSMOS provides embedded tests cases that can be compiled in C/C++ | NOT APPLICABLE | |
| U36 | COSMOS provides facilities for comparing the results of test case with Simulator in the Loop and test cases with HIL | NOT APPLICABLE | |
| U37 | COSMOS provides a facility to verify whether the target system *hardware* meets the requirements of the upgrade software package (i.e. pre-check before testing) | NOT APPLICABLE | |
| U38 | COSMOS suggests which tests must be run in which test phase to minimize efforts and redundancy while maintaining the same level of overall fault revealing power as before | NOT APPLICABLE | |
| U39 | COSMOS supports endurance testing of new software releases | NOT APPLICABLE | |
| U40 | COSMOS supports the testing if new production code are updatable | NOT APPLICABLE | |
| U41 | COSMOS supports the specification of test configurations and associated APIs for testing infrastructures | NOT APPLICABLE | |
| U42 | COSMOS testing supports variance of signals and to track/analyse corresponding output over time | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U43 | COSMOS supports testing using a predefined set of inputs | NOT APPLICABLE | |
| U44 | COSMOS can limit automatic testing to a user defined duration | NOT APPLICABLE | |
| U45 | COSMOS can stop automatic testing upon receiving an according request from the user | NOT APPLICABLE | |
| U46 | COSMOS is able to perform black box testing of OSGi bundles | NOT APPLICABLE | |

**7.5 Test Case Generation**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U47 | COSMOS provides support for defining test oracles (functionality decides if system under test passes) | NOT APPLICABLE | |
| U48 | COSMOS provides automated generation of test oracles | NOT APPLICABLE | |
| U49 | COSMOS generates tests based on a diverse set of testing objectives | NOT APPLICABLE | |
| U50 | COSMOS is able to generate test cases based on API specification (REST/SOAP) | NOT APPLICABLE | |
| U51 | COSMOS is able to generate test cases based on sensor data from MQTT messages with JSON payload | NOT APPLICABLE | |
| U52 | COSMOS uses API / Sensor data gathered from released software to generate tests for software in development / staging phases | NOT APPLICABLE | |
| U53 | COSMOS is able to test CANbus API in development / staging based on recordings of API usage from released software | NOT APPLICABLE | |
| U54 | COSMOS uses specification of valid input data (ranges, types, ..) to generate a wide range of valid inputs for API testing | NOT APPLICABLE | |
| U55 | COSMOS is able to generate test inputs based on known parameters and simulator models | NOT APPLICABLE | |
| U56 | COSMOS supports test case generation for embedded C/C++ components | NOT APPLICABLE | |

**7.6 Extracting Test Scenarios from User Interactions**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U57 | COSMOS can derive/adapt test oracles based on user feedback | NOT APPLICABLE | |
| U58 | COSMOS provides facilities for evaluating user reactions to test sequences | NOT APPLICABLE | |
| U59 | COSMOS is able to automatically create and analyse simulation scenarios in BeamNG.research against a defined "fitness" | NOT APPLICABLE | |

**7.7 Run-time Verification and Monitoring**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U60 | COSMOS supports Simulators and HIL for performance verification | NOT APPLICABLE | |
| U61 | COSMOS is able to evaluate assertions on run-time attributes (i.e. CPU Usage, Memory, Timings) | NOT APPLICABLE | |
| U62 | COSMOS is able to evaluate assertions on the occurrence of system events and misbehaviours (e.g. deploy-install-start-stopdelete) | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U63 | COSMOS provides automated diagnostics of failures (e.g. root cause analysis) detected during operation from the runtime monitoring framework of the application | NOT APPLICABLE | |
| U64 | COSMOS is able to analyse the reason of a particular failure and pin down its approximate location, being software, configuration, or hardware related | NOT APPLICABLE | |
| U65 | COSMOS provides runtime verification facilities for hardware-software integration | NOT APPLICABLE | |
| U66 | COSMOS supports the runtime verification of signal based properties | NOT APPLICABLE | |

**7.8 Security Assessment**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U67 | COSMOS provides security testing facilities to support security assurance processes | NOT APPLICABLE | |
| U68 | COSMOS provides cyclical evaluation of security improvements | NOT APPLICABLE | |
| U69 | COSMOS is able to support the identification and execution of security tests for remote system upgrades | NOT APPLICABLE | |
| U70 | COSMOS provides mechanisms for software vulnerabilities detection for deployed components including interactions with environment | NOT APPLICABLE | |
| U71 | COSMOS provides mechanisms for software vulnerabilities detection prior to deployment | NOT APPLICABLE | |

**7.9 Change Analysis**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U72 | COSMOS considers security requirements as part of the change analysis | NOT APPLICABLE | |
| U73 | COSMOS provides an estimation of the correction time of the identified revision (i.e. based on historical analysis/prediction) | NOT APPLICABLE | |
| U74 | COSMOS supports patch facilities with configurable file outputs | NOT APPLICABLE | |
| U75 | COSMOS is able to steer test selection and test prioritisation based on analysing software code changes in a code commit | NOT APPLICABLE | |

**7.10 Quality Assessment**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U76 | COSMOS allows comparisons between test data against a set of evaluation criteria | NOT APPLICABLE | |
| U77 | COSMOS provides a facility for monitoring activities as a basis for evaluating the quality of a product / patch release | NOT APPLICABLE | |
| U78 | COSMOS provides guidance for error resolution based on an automated approach for failure analysis and fault localisation | NOT APPLICABLE | |
| U79 | COSMOS is able to assess the execution of individual software components | NOT APPLICABLE | |

**7.11 Context Detection and Assessment**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U80 | COSMOS provides an assessment of the impact of component changes on other systems | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U81 | COSMOS is able to handle requests for patch management of other products | NOT APPLICABLE | |

**7.12 Interfaces**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U82 | COSMOS is able to track version information from OSGi bundles | NOT APPLICABLE | |
| U83 | COSMOS is aware of OSGi application life-cycle | NOT APPLICABLE | |
| U84 | COSMOS provides interfaces where baseline test data and criteria can be inputted into the system | NOT APPLICABLE | |
| U85 | COSMOS interfaces with the GitLab tools | NOT APPLICABLE | |
| U86 | COSMOS provides interfaces for controlling simulator executions | NOT APPLICABLE | |
| U87 | COSMOS provides an API to allow external control, receive feedback, and test executions | NOT APPLICABLE | |
| U88 | COSMOS is able to generate testing reports in machine readable format | NOT APPLICABLE | |
| U89 | COSMOS provides at least one input Interface to receive the subjects under test | NOT APPLICABLE | |
| U90 | COSMOS provides at least one output Interface for determining the subjects under test pass/fail status | NOT APPLICABLE | |

**7.13 DevOps Performance Indicators**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U91 | COSMOS provides a KPI framework containing relevant product quality and DevOps maturity indicators as well as indicators characterising business goals | NOT APPLICABLE | |
| U92 | COSMOS KPI framework is able to collect data along the DevOps pipeline, including Ops data for instances in the field | NOT APPLICABLE | |
| U93 | The COSMOS KPI framework includes lagging (backward-looking) and leading (forward-looking) KPIs | NOT APPLICABLE | |
| U94 | COSMOS collects and calculates the KPIs of the KPI framework and stores them for further analysis | NOT APPLICABLE | |
| U95 | COSMOS provides targeted dashboards for visualisation of KPIs for different stakeholders (e.g., testers, developers, managers (incl. CEO)) | NOT APPLICABLE | |
| U96 | COSMOS recommends measures based on the KPIs to improve business and development goals | PARTIALLLY | AP-SPOTTER provides warnings about performance antipatterns while refactoring book provides guidelines on how to address them. |
| U97 | COSMOS suggests dynamic adjustments of the KPI framework and the collected metrics based on changes in the DevOps process (meta level) | NOT APPLICABLE | |
| U98 | COSMOS enriches the KPI framework with aggregated KPIs to provide additional and targeted results for R&D steering (i.e. predictive) | NOT APPLICABLE | |

**7.14 General**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U99 | COSMOS supports software under test that is written in Java | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U100 | COSMOS supports software under test that provided as JAR files | NOT APPLICABLE | |
| U101 | COSMOS supports software under test provided as OSGi bundles | NOT APPLICABLE | |
| U102 | COSMOS is able to support black box testing | NOT APPLICABLE | |
| U103 | COSMOS supports software under test that is written in C/C++ | NOT APPLICABLE | |
| U104 | COSMOS ensures tests respect the limitations of the embedded system (e.g. number of cores, etc.) | NOT APPLICABLE | |
| U105 | Facilities are provided to support secure access from external tools to COSMOS | NOT APPLICABLE | |
| U106 | COSMOS is able to support existing security access facilities for pipeline infrastructure | NOT APPLICABLE | |
| U107 | COSMOS will not break the signature of a correctly signed software components (e.g. OSGi bundles) | NOT APPLICABLE | |
| U108 | COSMOS provides support for JSON (e.g. for test specifications) | NOT APPLICABLE | |
| U109 | COSMOS provides support for HTTP(S) and MQTT | NOT APPLICABLE | |
| U110 | COSMOS supports VNEXT Pipelines (i.e. no YAML Pipelines) with Azure DevOps Server on Premise (not in cloud) | NOT APPLICABLE | |
| U111 | COSMOS supports the version control systems TFVC (Microsoft TFS Version Control System) and GIT | NOT APPLICABLE | |
| U112 | COSMOS supports Microsoft C# programming languages for test and product code | NOT APPLICABLE | |
| U113 | COSMOS supports *Microsoft C++* programming languages for test and product code | NOT APPLICABLE | |
| U114 | COSMOS supports the frontend technology WPF and HTML5 (Angular) used in end-to-end-testing | NOT APPLICABLE | |
| U115 | COSMOS supports the Infrastructure Tooling in .NET Core / C# and PowerShell Core | NOT APPLICABLE | |
| U116 | COSMOS supports the requirements management tool Microsoft TFS Work Items | NOT APPLICABLE | |
| U117 | COSMOS supports TFS WorkItems for test management | NOT APPLICABLE | |
| U118 | COSMOS provides a facility to configure and manage test flows (i.e. test sequences and dependencies) | NOT APPLICABLE | |
| U119 | COSMOS checks for the presence and completeness of formal documents such as licenses and documentation | NOT APPLICABLE | |
| U120 | COSMOS has facilities to integrate with Jenkins | NOT APPLICABLE | |
| U121 | There is at least one User Interface to verify the functionalities of COSMOS are operational | NOT APPLICABLE | |

### 6.1.2 Test Decomposition and Test Generation

Table 10: COSMOS Requirement Coverage Overview

**7.1 CI/CD Pipelines**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U1 | COSMOS can be executed in one or more Docker container(s) | PARTIALLLY | • MICROTESTCARVER: Has not yet been implemented with Docker. This is intended to be completed in the near future.<br>• FAULTSPOTTER: Runs with Docker.<br>• TEST SCHEDULER: Runs with Docker. |
| U2 | COSMOS provides outputs and tools results in a human-readable format | YES | • MICROTESTCARVER: The tool returns unit-level test cases, these test cases can be reviewed and adjusted by the developers.<br>• FAULTSPOTTER: Returns the information in the form of a table to the developer.<br>• TEST SCHEDULER: provides an ordered list of test case to run in CI/CD pipelines. |
| U3 | COSMOS provides outputs and tools results in a machine-readable format for further processing | YES | • MICROTESTCARVER: The tool returns unit-level test cases, these test cases can be directly used as part of the test suite.<br>• FAULTSPOTTER: The information is stored in a .csv formatted file.<br>• TEST SCHEDULER: it provides a JSON with the ordered list of test cases to run. |
| U4 | COSMOS prevents application components that are not released by a gatekeeper in change management from being available in later pipeline stages | NOT APPLICABLE | |
| U5 | COSMOS used in change management is able to evaluate the impact of a changed software component regarding number of affected CPS, scope and which stakeholder-s/roles to inform about the change | NOT APPLICABLE | |
| U6 | COSMOS is able to support the rapid deployment of new adaptations | NOT APPLICABLE | |
| U7 | COSMOS provide results in a comparable way between adaptations (e.g. history) | YES | • MICROTESTCARVER: The tool is versioned with Git.<br>• FAULTSPOTTER: The tool is versioned with Git.<br>• TEST SCHEDULER: The tool is versioned with Git. |
| U8 | COSMOS is able to support testing based on data models | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U9 | COSMOS is able to devise a test strategy for system upgrades | PARTIALLLY | • The TEST SCHEDULER can be used to aid in devising a test strategy for system upgrades. |
| U10 | COSMOS tools are able to work within the existing inhouse CI/CD pipeline and test infrastructure | PARTIALLLY | • MICROTESTCARVER: it requires the compilation of the test with dedicated instrumentation.<br>• FAULTSPOTTER: it requires access to the Git history of the project.<br>• TEST SCHEDULER: it requires access to the source code but without compilation, building, or running the test (test as text). |
| U11 | COSMOS supports standalone execution of tools (outside development flow not linked to code change) | PARTIALLLY | • MICROTESTCARVER: The tool can be run by itself after instrumentation has been applied.<br>• FAULTSPOTTER: The tool can be run by itself after instrumentation has been applied.<br>• TEST SCHEDULER: The tool can be run by itself. |
| U12 | COSMOS can act as a gate keeper in the Bundle Pipeline by checking test results against software quality requirements | NOT APPLICABLE | |
| U13 | COSMOS provides a test management infrastructure to balance test executions over testing infrastructures (e.g. scaling and model type testing) | NOT APPLICABLE | |
| U14 | COSMOS provides integration with GitLab | NO | • MICROTESTCARVER: The tool does not provide integration with GitLab.<br>• FAULTSPOTTER: The tool does not provide integration with GitLab.<br>• TEST SCHEDULER: The tool does not provide integration with GitLab. |
| U15 | COSMOS is able to check for the proper signing of software components (e.g. OSGi bundles) | NOT APPLICABLE | |

**7.2 Bad Practices Detection and Anti-Patterns**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U16 | COSMOS provides tools for CI/CD best practices and anti-patterns | NOT APPLICABLE | |
| U17 | COSMOS is able to track best practices and antipatterns | NOT APPLICABLE | |
| U18 | COSMOS provides detectors for configuration, code and test smells (e.g. from static analysis, heuristics, etc.) | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U19 | Processing in COSMOS is sufficiently automated to avoid selective automation and user interventions | YES | • MICROTESTCARVER: only requires to use a dedicated instrumentation; the test generation is fully automated.<br>• FAULTSPOTTER: only requires to use a dedicated instrumentation; the fault localization is fully automated.<br>• The TEST SCHEDULER is sufficiently automated. |

**7.3 Simulators and HiL**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U20 | COSMOS is able to automatically generate tests cases that effectively explore the viable inputs for a SIL environment | NOT APPLICABLE | |
| U21 | COSMOS is able to automatically generate simulation scenarios for HIL testing | NOT APPLICABLE | |
| U22 | COSMOS supports Simulators and HIL for unit tests | NOT APPLICABLE | |
| U23 | COSMOS supports Simulators and HIL for unit integration tests | NOT APPLICABLE | |
| U24 | COSMOS supports simulation tools executing on Linux PPC or x86 platform | NOT APPLICABLE | |
| U25 | COSMOS is able to be configured to use the available API to control test executions on testing infrastructure | NOT APPLICABLE | |
| U26 | COSMOS is able to run software on emulated XMEGA | NOT APPLICABLE | |
| U27 | COSMOS supports interaction with the testing infrastructure over a local network | NOT APPLICABLE | |
| U28 | COSMOS is able to automatically generate simulation scenarios (e.g. in BeamNG.research, etc.) to provide CAN signals for HIL testing | NOT APPLICABLE | |

**7.4 Automated Testing**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U29 | COSMOS is able to generate test reports for groups of CPS or aggregate reports of multiple CPS | NOT APPLICABLE | |
| U30 | COSMOS is able to compare different test results and to provide developer feedback pointing out major differences | NOT APPLICABLE | |
| U31 | COSMOS is able to work with test storage and management facilities to allow assignability to a specific version of CPS or software component | NOT APPLICABLE | |
| U32 | COSMOS provides configurability to select individual criteria for each tested software component | NOT APPLICABLE | |
| U33 | COSMOS test results contain meta data related to the used hardware/device/CPS | PARTIALLLY | TEST SCHEDULER provides information about which test is associated to components and build tasks |
| U34 | COSMOS provides facilities for monitoring information from the system | NOT APPLICABLE | |
| U35 | COSMOS provides embedded tests cases that can be compiled in C/C++ | NOT APPLICABLE | |
| U36 | COSMOS provides facilities for comparing the results of test case with Simulator in the Loop and test cases with HIL | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U37 | COSMOS provides a facility to verify whether the target system *hardware* meets the requirements of the upgrade software package (i.e. pre-check before testing) | NOT APPLICABLE | |
| U38 | COSMOS suggests which tests must be run in which test phase to minimize efforts and redundancy while maintaining the same level of overall fault revealing power as before | NOT APPLICABLE | |
| U39 | COSMOS supports endurance testing of new software releases | NOT APPLICABLE | |
| U40 | COSMOS supports the testing if new production code are updatable | NOT APPLICABLE | |
| U41 | COSMOS supports the specification of test configurations and associated APIs for testing infrastructures | NOT APPLICABLE | |
| U42 | COSMOS testing supports variance of signals and to track/analyse corresponding output over time | NOT APPLICABLE | |
| U43 | COSMOS supports testing using a predefined set of inputs | NOT APPLICABLE | |
| U44 | COSMOS can limit automatic testing to a user defined duration | NOT APPLICABLE | |
| U45 | COSMOS can stop automatic testing upon receiving an according request from the user | NOT APPLICABLE | |
| U46 | COSMOS is able to perform black box testing of OSGi bundles | NOT APPLICABLE | |

**7.5 Test Case Generation**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U47 | COSMOS provides support for defining test oracles (functionality decides if system under test passes) | YES | MICROTESTCARVER generates unit-level test cases to strengthen the test oracle. |
| U48 | COSMOS provides automated generation of test oracles | YES | MICROTESTCARVER generates unit-level test cases to strengthen the test oracle. |
| U49 | COSMOS generates tests based on a diverse set of testing objectives | NOT APPLICABLE | |
| U50 | COSMOS is able to generate test cases based on API specification (REST/SOAP) | NOT APPLICABLE | |
| U51 | COSMOS is able to generate test cases based on sensor data from MQTT messages with JSON payload | NOT APPLICABLE | |
| U52 | COSMOS uses API / Sensor data gathered from released software to generate tests for software in development / staging phases | NOT APPLICABLE | |
| U53 | COSMOS is able to test CANbus API in development / staging based on recordings of API usage from released software | NOT APPLICABLE | |
| U54 | COSMOS uses specification of valid input data (ranges, types, ..) to generate a wide range of valid inputs for API testing | NOT APPLICABLE | |
| U55 | COSMOS is able to generate test inputs based on known parameters and simulator models | NOT APPLICABLE | |
| U56 | COSMOS supports test case generation for embedded C/C++ components | NO | MICROTESTCARVER supports generating test cases for JAVA projects. |

**7.6 Extracting Test Scenarios from User Interactions**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U57 | COSMOS can derive/adapt test oracles based on user feedback | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U58 | COSMOS provides facilities for evaluating user reactions to test sequences | NOT APPLICABLE | |
| U59 | COSMOS is able to automatically create and analyse simulation scenarios in BeamNG.research against a defined "fitness" | NOT APPLICABLE | |

**7.7 Run-time Verification and Monitoring**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U60 | COSMOS supports Simulators and HIL for performance verification | NOT APPLICABLE | |
| U61 | COSMOS is able to evaluate assertions on run-time attributes (i.e. CPU Usage, Memory, Timings) | NOT APPLICABLE | |
| U62 | COSMOS is able to evaluate assertions on the occurrence of system events and misbehaviours (e.g. deploy-install-start-stopdelete) | NOT APPLICABLE | |
| U63 | COSMOS provides automated diagnostics of failures (e.g. root cause analysis) detected during operation from the runtime monitoring framework of the application | NOT APPLICABLE | |
| U64 | COSMOS is able to analyse the reason of a particular failure and pin down its approximate location, being software, configuration, or hardware related | NOT APPLICABLE | |
| U65 | COSMOS provides runtime verification facilities for hardware-software integration | NOT APPLICABLE | |
| U66 | COSMOS supports the runtime verification of signal based properties | NOT APPLICABLE | |

**7.8 Security Assessment**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U67 | COSMOS provides security testing facilities to support security assurance processes | NOT APPLICABLE | |
| U68 | COSMOS provides cyclical evaluation of security improvements | NOT APPLICABLE | |
| U69 | COSMOS is able to support the identification and execution of security tests for remote system upgrades | NOT APPLICABLE | |
| U70 | COSMOS provides mechanisms for software vulnerabilities detection for deployed components including interactions with environment | NOT APPLICABLE | |
| U71 | COSMOS provides mechanisms for software vulnerabilities detection prior to deployment | NOT APPLICABLE | |

**7.9 Change Analysis**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U72 | COSMOS considers security requirements as part of the change analysis | NOT APPLICABLE | |
| U73 | COSMOS provides an estimation of the correction time of the identified revision (i.e. based on historical analysis/prediction) | NOT APPLICABLE | |
| U74 | COSMOS supports patch facilities with configurable file outputs | NOT APPLICABLE | |
| U75 | COSMOS is able to steer test selection and test prioritisation based on analysing software code changes in a code commit | NOT APPLICABLE | |

**7.10 Quality Assessment**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|

| | | | |
|---|---|---|---|
| U76 | COSMOS allows comparisons between test data against a set of evaluation criteria | NOT APPLICABLE | |
| U77 | COSMOS provides a facility for monitoring activities as a basis for evaluating the quality of a product / patch release | NOT APPLICABLE | |
| U78 | COSMOS provides guidance for error resolution based on an automated approach for failure analysis and fault localisation | YES | FAULTSPOTTER provides suspiciousness score for the compoments and files that may contain a bug in case of test failures |
| U79 | COSMOS is able to assess the execution of individual software components | NOT APPLICABLE | |

**7.11 Context Detection and Assessment**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U80 | COSMOS provides an assessment of the impact of component changes on other systems | NOT APPLICABLE | |
| U81 | COSMOS is able to handle requests for patch management of other products | NOT APPLICABLE | |

**7.12 Interfaces**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U82 | COSMOS is able to track version information from OSGi bundles | NOT APPLICABLE | |
| U83 | COSMOS is aware of OSGi application lifecycle | NOT APPLICABLE | |
| U84 | COSMOS provides interfaces where baseline test data and criteria can be inputted into the system | NOT APPLICABLE | |
| U85 | COSMOS interfaces with the GitLab tools | NOT APPLICABLE | |
| U86 | COSMOS provides interfaces for controlling simulator executions | NOT APPLICABLE | |
| U87 | COSMOS provides an API to allow external control, receive feedback, and test executions | NOT APPLICABLE | |
| U88 | COSMOS is able to generate testing reports in machine readable format | NOT APPLICABLE | |
| U89 | COSMOS provides at least one input Interface to receive the subjects under test | NOT APPLICABLE | |
| U90 | COSMOS provides at least one output Interface for determining the subjects under test pass/fail status | NOT APPLICABLE | |

**7.13 DevOps Performance Indicators**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U91 | COSMOS provides a KPI framework containing relevant product quality and DevOps maturity indicators as well as indicators characterising business goals | NOT APPLICABLE | |
| U92 | COSMOS KPI framework is able to collect data along the DevOps pipeline, including Ops data for instances in the field | NOT APPLICABLE | |
| U93 | The COSMOS KPI framework includes lagging (backward-looking) and leading (forward-looking) KPIs | NOT APPLICABLE | |
| U94 | COSMOS collects and calculates the KPIs of the KPI framework and stores them for further analysis | NOT APPLICABLE | |
| U95 | COSMOS provides targeted dashboards for visualisation of KPIs for different stakeholders (e.g., testers, developers, managers (incl. CEO)) | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U96 | COSMOS recommends measures based on the KPIs to improve business and development goals | NOT APPLICABLE | |
| U97 | COSMOS suggests dynamic adjustments of the KPI framework and the collected metrics based on changes in the DevOps process (meta level) | NOT APPLICABLE | |
| U98 | COSMOS enriches the KPI framework with aggregated KPIs to provide additional and targeted results for R&D steering (i.e. predictive) | NOT APPLICABLE | |

**7.14 General**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U99 | COSMOS supports software under test that is written in Java | PARTIALLLY | MICROTESTCARVER generates test cases from projects written in Java. TEST SCHEDULER analyzes test cases writte in JUnit. |
| U100 | COSMOS supports software under test that provided as JAR files | NOT APPLICABLE | |
| U101 | COSMOS supports software under test provided as OSGi bundles | NOT APPLICABLE | |
| U102 | COSMOS is able to support black box testing | YES | TEST SCHEDULER require access to the test cases only |
| U103 | COSMOS supports software under test that is written in C/C++ | PARTIALLLY | TEST SCHEDULER is black-box and could be applied to other programming languages. |
| U104 | COSMOS ensures tests respect the limitations of the embedded system (e.g. number of cores, etc.) | NOT APPLICABLE | |
| U105 | Facilities are provided to support secure access from external tools to COSMOS | NOT APPLICABLE | |
| U106 | COSMOS is able to support existing security access facilities for pipeline infrastructure | NOT APPLICABLE | |
| U107 | COSMOS will not break the signature of a correctly signed software components (e.g. OSGi bundles) | NOT APPLICABLE | |
| U108 | COSMOS provides support for JSON (e.g. for test specifications) | YES | TEST SCHEDULER provides the sorted lists of test cases in JSON format |
| U109 | COSMOS provides support for HTTP(S) and MQTT | NOT APPLICABLE | |
| U110 | COSMOS supports VNEXT Pipelines (i.e. no YAML Pipelines) with Azure DevOps Server on Premise (not in cloud) | NOT APPLICABLE | |
| U111 | COSMOS supports the version control systems TFVC (Microsoft TFS Version Control System) and GIT | NOT APPLICABLE | |
| U112 | COSMOS supports Microsoft C# programming languages for test and product code | NOT APPLICABLE | |
| U113 | COSMOS supports *Microsoft C++* programming languages for test and product code | NOT APPLICABLE | |
| U114 | COSMOS supports the frontend technology WPF and HTML5 (Angular) used in end-to-end-testing | NOT APPLICABLE | |
| U115 | COSMOS supports the Infrastructure Tooling in .NET Core / C# and PowerShell Core | NOT APPLICABLE | |
| U116 | COSMOS supports the requirements management tool Microsoft TFS Work Items | NOT APPLICABLE | |
| U117 | COSMOS supports TFS WorkItems for test management | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U118 | COSMOS provides a facility to configure and manage test flows (i.e. test sequences and dependencies) | NOT APPLICABLE | |
| U119 | COSMOS checks for the presence and completeness of formal documents such as licenses and documentation | NOT APPLICABLE | |
| U120 | COSMOS has facilities to integrate with Jenkins | NOT APPLICABLE | |
| U121 | There is at least one User Interface to verify the functionalities of COSMOS are operational | NOT APPLICABLE | |

### 6.1.3 User-oriented Maintenance and Testing

Table 11: COSMOS Requirement Coverage Overview

**7.1 CI/CD Pipelines**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U1 | COSMOS can be executed in one or more Docker container(s) | YES | • SDC-PRIORITIZER: The repository contains a Dockerfile and instructions on how to run it with Docker.<br>• The *Prototype for Elicitation of User (or Human) Feedback into the DevOps Cycle of CPSs* can be run in Docker containers (it has been dockerized by UNP as an integration action).<br>• PIXEL-SOO & PIXEL-MOO: Runs in Docker. |
| U2 | COSMOS provides outputs and tools results in a human-readable format | YES | PIXEL-SOO & PIXEL-MOO provides statistics about the adversarial examples being generated and the robustness of CNN models |
| U3 | COSMOS provides outputs and tools results in a machine-readable format for further processing | YES | REWOSA generates test reports in .csv formats |
| U4 | COSMOS prevents application components that are not released by a gatekeeper in change management from being available in later pipeline stages | NOT APPLICABLE | |
| U5 | COSMOS used in change management is able to evaluate the impact of a changed software component regarding number of affected CPS, scope and which stakeholder-s/roles to inform about the change | NOT APPLICABLE | |
| U6 | COSMOS is able to support the rapid deployment of new adaptations | NOT APPLICABLE | |
| U7 | COSMOS provide results in a comparable way between adaptations (e.g. history) | YES | • SDC-PRIORITIZER: is versioned in Git.<br>• PIXEL-SOO & PIXEL-MOO is versioned in Git. |
| U8 | COSMOS is able to support testing based on data models | PARTIALLLY | PIXEL-SOO & PIXEL-MOO helps assessing the robustness of models trained or driving scenarios videos and frames |
| U9 | COSMOS is able to devise a test strategy for system upgrades | PARTIALLLY | RESTORE is able to provide support for repairing faults in integration components |
| U10 | COSMOS tools are able to work within the existing inhouse CI/CD pipeline and test infrastructure | YES | • SDC-PRIORITIZER: Supported by the use of Docker.<br>• The *Prototype for Elicitation of User (or Human) Feedback into the DevOps Cycle of CPSs* is supported by the use of Docker.<br>• pixel just requires access to the input/output of the mdoel under test |
| U11 | COSMOS supports standalone execution of tools (outside development flow not linked to code change) | YES | • SDC-PRIORITIZER: The tool does not need to track code changes.<br>• The *Prototype for Elicitation of User (or Human) Feedback into the DevOps Cycle of CPSs* does not need to track code changes. |

| | | | |
|---|---|---|---|
| U12 | COSMOS can act as a gate keeper in the Bundle Pipeline by checking test results against software quality requirements | NOT APPLICABLE | |
| U13 | COSMOS provides a test management infrastructure to balance test executions over testing infrastructures (e.g. scaling and model type testing) | PARTIALLLY | RESTORE and SDC-PRIORITIZER provide information on which tests to run. Test suite optimized with *SO-SDC-Prioritizer* and RESTORE can be run in parallel over multiple machines |
| U14 | COSMOS provides integration with GitLab | NO | |
| U15 | COSMOS is able to check for the proper signing of software components (e.g. OSGi bundles) | NO | |

**7.2 Bad Practices Detection and Anti-Patterns**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U16 | COSMOS provides tools for CI/CD best practices and anti-patterns | NOT APPLICABLE | |
| U17 | COSMOS is able to track best practices and antipatterns | PARTIALLLY | • The *Prototype for Elicitation of User (or Human) Feedback into the DevOps Cycle of CPSs* Not applicable. |
| U18 | COSMOS provides detectors for configuration, code and test smells (e.g. from static analysis, heuristics, etc.) | PARTIALLLY | • The *Prototype for Elicitation of User (or Human) Feedback into the DevOps Cycle of CPSs* |
| U19 | Processing in COSMOS is sufficiently automated to avoid selective automation and user interventions | YES | • SDC-PRIORITIZER: The tool is usable via the CLI and all results are machine-readable for enabling automation. Hence, all are sufficiently automated to avoid selective automation and user interventions.<br>• The *Prototype for Elicitation of User (or Human) Feedback into the DevOps Cycle of CPSs* is sufficiently automated to avoid selective automation and user interventions. |

**7.3 Simulators and HiL**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U20 | COSMOS is able to automatically generate tests cases that effectively explore the viable inputs for a SIL environment | NOT APPLICABLE | |
| U21 | COSMOS is able to automatically generate simulation scenarios for HIL testing | PARTIALLLY | • SDC-PRIORITIZER: Possible future integration and compatibility with TEASER |
| U22 | COSMOS supports Simulators and HIL for unit tests | PARTIALLLY | • SDC-PRIORITIZER: The tool supports currently only system-level tests. In the future, it might work with TEASER for HiL and with multiple simulators. |
| U23 | COSMOS supports Simulators and HIL for unit integration tests | NOT APPLICABLE | |
| U24 | COSMOS supports simulation tools executing on Linux PPC or x86 platform | PARTIALLLY | • SDC-PRIORITIZER: The tool prioritizes test cases for the BeamNG simulator, which only runs on Windows, but the standalone execution of SDC-PRIORITIZER works on Linux. |

| | | | |
|---|---|---|---|
| U25 | COSMOS is able to be configured to use the available API to control test executions on testing infrastructure | YES | |
| U26 | COSMOS is able to run software on emulated XMEGA | NOT APPLICABLE | |
| U27 | COSMOS supports interaction with the testing infrastructure over a local network | NOT APPLICABLE | |
| U28 | COSMOS is able to automatically generate simulation scenarios (e.g. in BeamNG.research, etc.) to provide CAN signals for HIL testing | NOT APPLICABLE | |

**7.4 Automated Testing**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U29 | COSMOS is able to generate test reports for groups of CPS or aggregate reports of multiple CPS | NOT APPLICABLE | |
| U30 | COSMOS is able to compare different test results and to provide developer feedback pointing out major differences | PARTIALLLY | RESTORE provides information about how to fix buggy intergration code |
| U31 | COSMOS is able to work with test storage and management facilities to allow assignability to a specific version of CPS or software component | NOT APPLICABLE | |
| U32 | COSMOS provides configurability to select individual criteria for each tested software component | PARTIALLLY | TEST SCHEDULER consider the cost of the test when providing prioritization suggestions |
| U33 | COSMOS test results contain meta data related to the used hardware/device/CPS | NOT APPLICABLE | |
| U34 | COSMOS provides facilities for monitoring information from the system | NOT APPLICABLE | |
| U35 | COSMOS provides embedded tests cases that can be compiled in C/C++ | NOT APPLICABLE | |
| U36 | COSMOS provides facilities for comparing the results of test case with Simulator in the Loop and test cases with HIL | NOT APPLICABLE | |
| U37 | COSMOS provides a facility to verify whether the target system *hardware* meets the requirements of the upgrade software package (i.e. pre-check before testing) | NOT APPLICABLE | |
| U38 | COSMOS suggests which tests must be run in which test phase to minimize efforts and redundancy while maintaining the same level of overall fault revealing power as before | YES | TEST SCHEDULER provides information about which test to run for each component |
| U39 | COSMOS supports endurance testing of new software releases | PARTIALLLY | Test suite optimized by TEST SCHEDULER can be ran over multiple releases |
| U40 | COSMOS supports the testing if new production code are updatable | NOT APPLICABLE | |
| U41 | COSMOS supports the specification of test configurations and associated APIs for testing infrastructures | NO | |
| U42 | COSMOS testing supports variance of signals and to track/analyse corresponding output over time | NOT APPLICABLE | |
| U43 | COSMOS supports testing using a predefined set of inputs | YES | TEST SCHEDULER requires in input the folder(s) with the test cases to analyse |
| U44 | COSMOS can limit automatic testing to a user defined duration | YES | With TEST SCHEDULER, users can specify how much time and resources to spend on regression testing |

| | | | |
|---|---|---|---|
| U45 | COSMOS can stop automatic testing upon receiving an according request from the user | NO | |
| U46 | COSMOS is able to perform black box testing of OSGi bundles | NOT APPLICABLE | |

**7.5 Test Case Generation**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U47 | COSMOS provides support for defining test oracles (functionality decides if system under test passes) | NO | |
| U48 | COSMOS provides automated generation of test oracles | YES | PIXEL-SOO & PIXEL-MOOrely on metamorphic testing to determine whether an attack is successful or not |
| U49 | COSMOS generates tests based on a diverse set of testing objectives | YES | TEST SCHEDULER and SDC-PRIORITIZER use multi-objective approaches to optimize a diverse set of objectices |
| U50 | COSMOS is able to generate test cases based on API specification (REST/SOAP) | NOT APPLICABLE | |
| U51 | COSMOS is able to generate test cases based on sensor data from MQTT messages with JSON payload | NOT APPLICABLE | |
| U52 | COSMOS uses API / Sensor data gathered from released software to generate tests for software in development / staging phases | NOT APPLICABLE | |
| U53 | COSMOS is able to test CANbus API in development / staging based on recordings of API usage from released software | NOT APPLICABLE | |
| U54 | COSMOS uses specification of valid input data (ranges, types, ..) to generate a wide range of valid inputs for API testing | NOT APPLICABLE | |
| U55 | COSMOS is able to generate test inputs based on known parameters and simulator models | NOT APPLICABLE | |
| U56 | COSMOS supports test case generation for embedded C/C++ components | NO | |

**7.6 Extracting Test Scenarios from User Interactions**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U57 | COSMOS can derive/adapt test oracles based on user feedback | PARTIALLLY | |
| U58 | COSMOS provides facilities for evaluating user reactions to test sequences | PARTIALLLY | |
| U59 | COSMOS is able to automatically create and analyse simulation scenarios in BeamNG.research against a defined "fitness" | YES | • SDC-PRIORITIZER: The multi-objective approach optimizes the test diversity and the execution time. Both aspects are reflected in a fitness function(s). |

**7.7 Run-time Verification and Monitoring**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U60 | COSMOS supports Simulators and HIL for performance verification | NO | |
| U61 | COSMOS is able to evaluate assertions on run-time attributes (i.e. CPU Usage, Memory, Timings) | NO | |
| U62 | COSMOS is able to evaluate assertions on the occurrence of system events and misbehaviours (e.g. deploy-install-start-stopdelete) | NOT APPLICABLE | |
| U63 | COSMOS provides automated diagnostics of failures (e.g. root cause analysis) detected during operation from the runtime monitoring framework of the application | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U64 | COSMOS is able to analyse the reason of a particular failure and pin down its approximate location, being software, configuration, or hardware related | NOT APPLICABLE | |
| U65 | COSMOS provides runtime verification facilities for hardware-software integration | NOT APPLICABLE | |
| U66 | COSMOS supports the runtime verification of signal based properties | NOT APPLICABLE | |

**7.8 Security Assessment**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U67 | COSMOS provides security testing facilities to support security assurance processes | NOT APPLICABLE | |
| U68 | COSMOS provides cyclical evaluation of security improvements | NOT APPLICABLE | |
| U69 | COSMOS is able to support the identification and execution of security tests for remote system upgrades | NOT APPLICABLE | |
| U70 | COSMOS provides mechanisms for software vulnerabilities detection for deployed components including interactions with environment | NOT APPLICABLE | |
| U71 | COSMOS provides mechanisms for software vulnerabilities detection prior to deployment | NOT APPLICABLE | |

**7.9 Change Analysis**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U72 | COSMOS considers security requirements as part of the change analysis | NOT APPLICABLE | |
| U73 | COSMOS provides an estimation of the correction time of the identified revision (i.e. based on historical analysis/prediction) | NOT APPLICABLE | |
| U74 | COSMOS supports patch facilities with configurable file outputs | NOT APPLICABLE | |
| U75 | COSMOS is able to steer test selection and test prioritisation based on analysing software code changes in a code commit | NOT APPLICABLE | |

**7.10 Quality Assessment**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U76 | COSMOS allows comparisons between test data against a set of evaluation criteria | YES | PIXEL-SOO & PIXEL-MOOprovides test results comparison between original data-sample and generated attacks |
| U77 | COSMOS provides a facility for monitoring activities as a basis for evaluating the quality of a product / patch release | NOT APPLICABLE | |
| U78 | COSMOS provides guidance for error resolution based on an automated approach for failure analysis and fault localisation | PARTIALLLY | RESTORE uses fault localization and provide candidate patches for the integ¯ration components |
| U79 | COSMOS is able to assess the execution of individual software components | NOT APPLICABLE | |

**7.11 Context Detection and Assessment**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U80 | COSMOS provides an assessment of the impact of component changes on other systems | NOT APPLICABLE | |
| U81 | COSMOS is able to handle requests for patch management of other products | NOT APPLICABLE | |

**7.12 Interfaces**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|

| | | | |
|---|---|---|---|
| U82 | COSMOS is able to track version information from OSGi bundles | NOT APPLICABLE | |
| U83 | COSMOS is aware of OSGi application life-cycle | NOT APPLICABLE | |
| U84 | COSMOS provides interfaces where baseline test data and criteria can be inputted into the system | NOT APPLICABLE | |
| U85 | COSMOS interfaces with the GitLab tools | NOT APPLICABLE | |
| U86 | COSMOS provides interfaces for controlling simulator executions | NOT APPLICABLE | |
| U87 | COSMOS provides an API to allow external control, receive feedback, and test executions | NOT APPLICABLE | |
| U88 | COSMOS is able to generate testing reports in machine readable format | NOT APPLICABLE | |
| U89 | COSMOS provides at least one input Interface to receive the subjects under test | NOT APPLICABLE | |
| U90 | COSMOS provides at least one output Interface for determining the subjects under test pass/fail status | NOT APPLICABLE | |

**7.13 DevOps Performance Indicators**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U91 | COSMOS provides a KPI framework containing relevant product quality and DevOps maturity indicators as well as indicators characterising business goals | NOT APPLICABLE | |
| U92 | COSMOS KPI framework is able to collect data along the DevOps pipeline, including Ops data for instances in the field | NOT APPLICABLE | |
| U93 | The COSMOS KPI framework includes lagging (backward-looking) and leading (forward-looking) KPIs | NOT APPLICABLE | |
| U94 | COSMOS collects and calculates the KPIs of the KPI framework and stores them for further analysis | NOT APPLICABLE | |
| U95 | COSMOS provides targeted dashboards for visualisation of KPIs for different stakeholders (e.g., testers, developers, managers (incl. CEO)) | NOT APPLICABLE | |
| U96 | COSMOS recommends measures based on the KPIs to improve business and development goals | NOT APPLICABLE | |
| U97 | COSMOS suggests dynamic adjustments of the KPI framework and the collected metrics based on changes in the DevOps process (meta level) | NOT APPLICABLE | |
| U98 | COSMOS enriches the KPI framework with aggregated KPIs to provide additional and targeted results for R&D steering (i.e. predictive) | NOT APPLICABLE | |

**7.14 General**

| *ID* | *Requirement* | *Coverage level* | *Description* |
|---|---|---|---|
| U99 | COSMOS supports software under test that is written in Java | NOT APPLICABLE | |
| U100 | COSMOS supports software under test that provided as JAR files | NOT APPLICABLE | |
| U101 | COSMOS supports software under test provided as OSGi bundles | NOT APPLICABLE | |

| | | | |
|---|---|---|---|
| U102 | COSMOS is able to support black box testing | YES | PIXEL-SOO & PIXEL-MOO, REWOSA, and SDC-PRIORITIZER are fully black-box |
| U103 | COSMOS supports software under test that is written in C/C++ | NOT APPLICABLE | |
| U104 | COSMOS ensures tests respect the limitations of the embedded system (e.g. number of cores, etc.) | NOT APPLICABLE | |
| U105 | Facilities are provided to support secure access from external tools to COSMOS | NOT APPLICABLE | |
| U106 | COSMOS is able to support existing security access facilities for pipeline infrastructure | NOT APPLICABLE | |
| U107 | COSMOS will not break the signature of a correctly signed software components (e.g. OSGi bundles) | NOT APPLICABLE | |
| U108 | COSMOS provides support for JSON (e.g. for test specifications) | NOT APPLICABLE | |
| U109 | COSMOS provides support for HTTP(S) and MQTT | NOT APPLICABLE | |
| U110 | COSMOS supports VNEXT Pipelines (i.e. no YAML Pipelines) with Azure DevOps Server on Premise (not in cloud) | NOT APPLICABLE | |
| U111 | COSMOS supports the version control systems TFVC (Microsoft TFS Version Control System) and GIT | NOT APPLICABLE | |
| U112 | COSMOS supports Microsoft C# programming languages for test and product code | NOT APPLICABLE | |
| U113 | COSMOS supports *Microsoft C++* programming languages for test and product code | NOT APPLICABLE | |
| U114 | COSMOS supports the frontend technology WPF and HTML5 (Angular) used in end-to-end-testing | NOT APPLICABLE | |
| U115 | COSMOS supports the Infrastructure Tooling in .NET Core / C# and PowerShell Core | NOT APPLICABLE | |
| U116 | COSMOS supports the requirements management tool Microsoft TFS Work Items | NOT APPLICABLE | |
| U117 | COSMOS supports TFS WorkItems for test management | NOT APPLICABLE | |
| U118 | COSMOS provides a facility to configure and manage test flows (i.e. test sequences and dependencies) | PARTIALLLY | The TEST SCHEDULER considers test sequences and dependencies. |
| U119 | COSMOS checks for the presence and completeness of formal documents such as licenses and documentation | NOT APPLICABLE | |
| U120 | COSMOS has facilities to integrate with Jenkins | NOT APPLICABLE | |
| U121 | There is at least one User Interface to verify the functionalities of COSMOS are operational | NOT APPLICABLE | |

## 6.2 Future work

In previous sections, we elaborated on the status of the current development. In the following sections, we elaborate on the future work and with specific focus on activities that are expected to be addressed in next months of the COSMOS projects. The main future activities will be devoted to the following aspects:

- For PIXEL-SOO & PIXEL-MOO, we have shown how our tool is effective and efficient in generating adversarial example of vision components of CPS, and convolutional neural networks in particular. As part of our future work, we aim to more explicitly look at the potential difficulty of images due to semantic ambiguity, which already can show in the initial classification confidence of a model [98]. Furthermore, we intend to extend our comparison to multiple deep learning models, and different search algorithms (e.g., MOEA/D [179] and AGE-MOEA [124]) or combine the strengths of `Pixel-MOO` and `Pixel-SOO` with hybrid approaches.
- We have implemented our RESTORE as a prototype. Part of our future agenda is to complete the evaluation of the tool.
- REWOSA has been implemented and evaluated with for BeamNG v0.24.0.1. However, this version of the BeamNG simulation engine is no longer supported. Part of our activities will be devoted in updating REWOSA with the latest BeamNG version (v0.26.2.0).
- The implementation of the TEST SCHEDULER tool will be evaluated with both open source and industrial data.
- The evaluation for the MICROTESTCARVER tool will be expanded with a larger set of CPS projects. Moreover, Docker support will be added.

# References

[1] Btrace - a safe, dynamic tracing tool for the java platform, December 2022.

[2] Serialize java objects to xml and back again., December 2022.

[3] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 143–154, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Automated repair of feature interaction failures in automated driving systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 88–100, 2020.

[5] Sheeva Afshan, Phil McMinn, and Mark Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 352–361. IEEE, 2013.

[6] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Software Eng.*, 36(6):742–762, 2010.

[7] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proc. Int'l Conf. on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017.

[8] Juan Carlos Alonso, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. Arte: Automated generation of realistic test inputs for web apis. *IEEE Transactions on Software Engineering*, pages 1–1, 2022.

[9] Hirohisa Aman, Takashi Nakano, Hideto Ogasawara, and Minoru Kawahara. A topic model and test history-based test case recommendation method for regression testing. In *2018 IEEE international conference on software testing, verification and validation workshops (ICSTW)*, pages 392–397. IEEE, 2018.

[10] Rakesh Angira and BV Babu. Non-dominated sorting differential evolution (nsde): An extension of differential evolution for multi-objective optimization. In *IICAI*, pages 1428–1443, 2005.

[11] Maurício Finavaro Aniche, Christoph Treude, and Andy Zaidman. How developers engineer test cases: An observational study. *IEEE Trans. Software Eng.*, 48(12):4925–4946, 2022.

[12] Shay Artzi, Sunghun Kim, and Michael D Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP 2008–Object-Oriented Programming*, pages 542–565. Springer, 2008.

[13] Luciano Baresi and Matteo Miraz. Testful: automatic unit-test generation for java classes. In *32nd IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 281–284. ACM, 2010.

[14] BeamNG GmbH. BeamNG.tech, 2017. version: 0.25.0.0.

[15] Kent L. Beck. *Test-Driven Development - By Example*. The Addison-Wesley signature series. Addison-Wesley, 2003.

[16] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 362–371. IEEE Press, 2013.

[17] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the IDE: patterns, beliefs, and behavior. *IEEE Trans. Software Eng.*, 45(3):261–284, 2019.

[18] Aman Bhalla, Munipalle Sai Nikhila, and Pradeep Singh. Simulation of self-driving car using deep learning. In *2020 3rd International Conference on Intelligent Sustainable Systems (ICISS)*, pages 519–525. IEEE, 2020.

[19] Christian Birchler, Sajad Khatiri, Pouria Derakhshanfar, Sebastiano Panichella, and Annibale Panichella. Single and multi-objective test cases prioritization for self-driving cars in virtual environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2022.

[20] J. Blank and K. Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

[21] Renée C Bryce and Charles J Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960–970, 2006.

[22] Renee C. Bryce, Charles J. Colbourn, and Myra B. Cohen. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the 27th International Conference on Software Engineering*, pages 146–155, 2005.

[23] Renée C Bryce and Atif M Memon. Test suite prioritization by interaction coverage. In *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*, pages 1–7. ACM, 2007.

[24] Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Trans. on Software Engineering*, 36(4):546–558, 2010.

[25] Yu Cao, Hongyu Zhang, and Sun Ding. Symcrash: selective recording for reproducing crashes. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 791–802. ACM, 2014.

[26] Yuxiang Cao, Zhi Quan Zhou, and Tsong Yueh Chen. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 153–162. IEEE, 2013.

[27] Kenneth Chan and Betty HC Cheng. Evoattack: An evolutionary search-based adversarial attack for object detection models. In *International Symposium on Search-Based Software Engineering*, pages 83–97. Springer, 2022.

[28] Hong Chen. Applications of Cyber-Physical System: A Literature Review. *Journal of Industrial Integration and Management*, 02(03):1750012, 9 2017.

[29] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Semantic image segmentation with deep convolutional nets and fully connected crfs. *arXiv preprint arXiv:1412.7062*, 2014.

[30] Ning Chen and Sunghun Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Tr. on Sw. Eng.*, 41(2):198–220, 2015.

[31] Tsong Yueh Chen and Man Fai Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, November 1996.

[32] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959, 2019.

[33] François Chollet et al. Keras. `https://keras.io`, 2015.

[34] Tinkle Chugh, Karthik Sindhya, Jussi Hakanen, and Kaisa Miettinen. A survey on handling computationally expensive multiobjective optimization problems with evolutionary algorithms. *Soft Computing*, 23, 2019.

[35] James Clause and Alessandro Orso. A technique for enabling and supporting debugging of field failures. In *Proceedings of the 29th international conference on Software Engineering*, pages 261–270. IEEE Computer Society, 2007.

[36] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[37] M.B. Cohen, M.B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34:633–650, 2008.

[38] William Jay Conover. *Practical nonparametric statistics*, volume 350. John Wiley & Sons, 1998.

[39] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Proceedings of Joint Meeting on Foundations of Software Engineering (FSE)*, pages 107–118. ACM, 2015.

[40] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 57–67. ACM, 2017.

[41] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Applying a smoothing filter to improve IR-based traceability recovery processes: An empirical investigation. *Information and Software Technology (IST)*, 55(4):741–754, 2013.

[42] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2000.

[43] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[44] Joanna F. DeFranco and Dimitrios Serpanos. The 12 Flavors of Cyberphysical Systems. *Computer*, 54(12):104–108, 3 2021.

[45] Amirhossein Deljouyi. Carving unit tests from e2e tests, 2023.

[46] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *15th International Symposium on Software Reliability Engineering*, pages 113–124. IEEE Computer Society, 2004.

[47] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[48] Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 85–91, 2016.

[49] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*, pages 253–264. ACM, 2006.

[50] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceeding of the 23rd International Conference on Software Engineering*, pages 329–338. IEEE, 2001.

[51] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 329–338. IEEE, 2001.

[52] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 102–112. ACM, 2000.

[53] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

[54] Michael G Epitropakis, Shin Yoo, Mark Harman, and Edmund K Burke. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 234–245. ACM, 2015.

[55] Mark Fewster and Dorothy Graham. *Software test automation*. Addison-Wesley Reading, 1999.

[56] Ronald Aylmer Fisher. Statistical methods for research workers. In *Breakthroughs in statistics*, pages 66–70. Springer, 1992.

[57] Carlos M. Fonseca and Peter J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 3:1–16, 1995.

[58] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proc. Joint Meeting Symp. Foundations of Software Engineering and the European Softw. Eng. Conf. (ESEC/FSE)*, pages 416–419. ACM, 2011.

[59] Gordon Fraser and Andrea Arcuri. EvoSuite: On the challenges of test case generation in the real world. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 362–369. IEEE, 2013.

[60] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[61] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? A controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, 24(4):23:1–23:49, 2015.

[62] S Gautam and A Kumar. Automatic traffic light detection for self-driving cars using transfer learning. In *Intelligent Sustainable Systems: Selected Papers of WorldS4 2021, Volume 1*, pages 597–606. Springer, 2022.

[63] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2017.

[64] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[65] Salah Ghamizi, Maxime Cordy, Martin Gubri, Mike Papadakis, Andrey Boystov, Yves Le Traon, and Anne Goujon. *Search-Based Adversarial Testing and Improvement of Constrained Credit Scoring Systems*, page 1089–1100. Association for Computing Machinery, New York, NY, USA, 2020.

[66] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[67] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*, pages 88–99. IEEE, 2016.

[68] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[69] Giovanni Grano, Simone Scalabrino, Harald C Gall, and Rocco Oliveto. An empirical investigation on the readability of manual and generated test cases. In *International Conference on Program Comprehension (ICPC)*, pages 348–351. IEEE, 2018.

[70] Jianmin Guo, Yue Zhao, Houbing Song, and Yu Jiang. Coverage guided differential adversarial testing of deep learning systems. *IEEE Transactions on Network Science and Engineering*, 8(2):933–942, 2021.

[71] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. A unified test case prioritization approach. *ACM Transactions on Software Engineering Methodology*, 24(2):10:1–10:31, 2014.

[72] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. Technical report, 2015.

[73] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):6, 2013.

[74] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing white-box and black-box test prioritization. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 523–534. IEEE, 2016.

[75] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 40(7):650–670, 2014.

[76] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 188–197. IEEE, 2013.

[77] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[78] Shoma Ishida and Satoshi Ono. Adjust-free adversarial example generation in speech recognition using evolutionary multi-objective optimization under black-box condition. *Artif. Life Robot.*, 26(2):243–249, may 2021.

[79] M.M. Islam, A. Marchetto, A. Susi, and G. Scanniello. A multi-objective technique to prioritize test cases based on latent semantic indexing. In *Proceedings of European Conference on Software Maintenance and Reengineering*, pages 21–30. IEEE, 2012.

[80] Huatao Jiang, Lin Chang, Qing Li, and Dapeng Chen. Deep transfer learning enable end-to-end steering angles prediction for self-driving car. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 405–412. IEEE, 2020.

[81] Wei Jin and Alessandro Orso. F3: fault localization for field failures. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA:13)*, pages 213–223, Lugano, Switzerland, 2013. ACM.

[82] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[83] Shreya Khare, Rahul Aralikatte, and Senthil Man. Adversarial Black-Box Attacks on Automatic Speech Recognition Systems using Multi-Objective Evolutionary Optimization. In *Proceedings of INTERSPEECH*, 2019.

[84] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Manning, 2019.

[85] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*, pages 802–811, San Francisco, CA, USA, 2013. IEEE.

[86] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 1039–1049. IEEE Press, 2019.

[87] Amy J. Ko, Bryan Dosono, and Neeraja Duriseti. Thirty years of software problems in the news. In *Proc. Int'l Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 32–39. ACM, 2014.

[88] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25:1980–2024, 2020.

[89] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[90] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)*, 38(1):54–72, Jan 2012.

[91] Claudia Leacock and Martin Chodorow. Combining local context and wordnet similarity for word sense identification. *WordNet: An electronic lexical database*, 49(2):265–283, 1998.

[92] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, 2012.

[93] Edouard Leurent. An environment for autonomous driving decision-making. `https://github.com/eleurent/highway-env`, 2018.

[94] Quanyi Li, Zhenghao Peng, Lan Feng, Qihang Zhang, Zhenghai Xue, and Bolei Zhou. Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[95] Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614, 2020.

[96] Zheng Li, Yi Bian, Ruilian Zhao, and Jun Cheng. A fine-grained parallel multi-objective test case prioritization on gpu. In *Proceeding of the Symposium on Search-Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 111–125. Springer Berlin Heidelberg, 2013.

[97] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[98] Cynthia C. S. Liem and Annibale Panichella. Oracle Issues in Machine Learning and Where To Find Them. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020.

[99] Junyu Lin, Lei Xu, Yingqi Liu, and Xiangyu Zhang. Black-box adversarial sample generation based on differential evolution, 2020.

[100] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE, 2019.

[101] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42, 2019.

[102] Yiling Lou, Samuel Benton, Dan Hao, Lu Zhang, and Lingming Zhang. How does regression test selection affect program repair? an extensive study on 2 million patches. *arXiv preprint arXiv:2105.07311*, 2021.

[103] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. DeepMutation: Mutation testing of deep learning systems. In *Proceedings of the 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018.

[104] Alexey G. Malishevsky, Joseph R. Ruthruff, Gregg Rothermel, and Sebastian Elbaum. Cost-cognizant test case prioritization. Technical report, Department of Computer Science and Engineering, 2006.

[105] A. Marchetto, M. M. Islam, W. Asghar, A. Susi, and G. Scanniello. A multi-objective technique to prioritize test cases. *IEEE Transactions on Software Engineering*, 42(10):918–940, 2016.

[106] Martina Marré and Antonia Bertolino. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, 29(11):974–984, 2003.

[107] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th international conference on software engineering*, pages 492–495, 2014.

[108] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the International Conference on Software Engineering (ICSE'16)*, pages 691–701, New York, NY, USA, 2016. ACM.

[109] G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.

[110] Vojta Molda. Autodrome, 2022. Retrieved on 2022.

[111] James Montgomery and Stephen Chen. An analysis of the operation of differential evolution at high and low crossover rates. In *IEEE congress on evolutionary computation*, pages 1–8. IEEE, 2010.

[112] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks, 2015.

[113] Olaf Musch. *Factory Method*, pages 205–216. Springer Fachmedien Wiesbaden, Wiesbaden, 2023.

[114] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News*, pages 284–295. IEEE Computer Society, 2005.

[115] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiène Tahar, and Alf Larsson. Jcharming: A bug reproduction approach using crash traces and directed model checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 101–110. IEEE, 2015.

[116] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[117] John Nickolls. Scalable parallel programming with cuda. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 1–2, 2008.

[118] Nienke Nijkamp, Carolin Brandt, and Andy Zaidman. Naming amplified tests based on improved coverage. In *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM)*, pages 237–241, 2021.

[119] S.O. Okolie, S.O. Kuyoro, and O. B Ohwo. Emerging Cyber-Physical Systems : An Overview. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pages 306–316, 12 2018.

[120] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, Hidehiko Masuhara, and Fernando Castor. A systematic literature review on the impact of formatting elements on program understandability, 2022.

[121] PabloAlvarezLopezandMichaelBehrischandLauraBieker-WalzandJakobErdmannandYun-PangFlötteröandRobertHilbrichandLeonhardLückenandJohannesRummelandPeterWagnerandEvamarieWießner. Microscopictrafficsimulationusingsumo. In *The21stIEEEInternationalConferenceonIntelligentTransportationSystems*. IEEE, 2018.

[122] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Conf. on Object-Oriented Programming Systems and Applications (OOPSLA-Companion)*, pages 815–816. ACM, 2007.

[123] A. Panichella, R. Oliveto, M. Di Penta, and A. De Lucia. Improving multi-objective test case selection by injecting diversity in genetic algorithms. *IEEE Transactions on Software Engineering*, 2015.

[124] Annibale Panichella. An adaptive evolutionary algorithm based on non-euclidean geometry for many-objective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 595–603, 2019.

[125] Annibale Panichella. A systematic comparison of search algorithms for topic modelling—a study on duplicate bug report identification. In *11th International Symposium on Search Based Software Engineering (SSBSE)*, pages 11–26. Springer, 2019.

[126] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. Parameterizing and assembling IR-based solutions for SE tasks using genetic algorithms. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016.

[127] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Trans. Software Eng.*, 44:122–158, 2018.

[128] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In *2020 IEEE international conference on software maintenance and evolution (ICSME)*, pages 523–533. IEEE, 2020.

[129] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering*, 27(7):170, 2022.

[130] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. How can I improve my app? Classifying user reviews for software maintenance and evolution. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015.

[131] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proc. Int'l Conference on Software Engineering (ICSE)*, pages 547–558, 2016.

[132] Mike Papadakis, Christopher Henard, and Yves Le Traon. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 1–10. IEEE, 2014.

[133] Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, Daniel Hoehener, Shih-Yuan Liu, Michael Novitzky, Igor Franzoni Okuyama, Jason Pazis, Guy Rosman, Valerio Varricchio, Hsueh-Cheng Wang, Dmitry Yershov, Hang Zhao, Michael Benjamin, Christopher Carr, Maria Zuber, Sertac Karaman, Emilio Frazzoli, Domitilla Del Vecchio, Daniela Rus, Jonathan How, John Leonard, and Andrea Censi. Duckietown: An open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1497–1504, 2017.

[134] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

[135] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. *Commun. ACM*, 62(11):137–145, oct 2019.

[136] Justyna Petke, Shin Yoo, Myra B Cohen, and Mark Harman. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 26–36. ACM, 2013.

[137] Martin F Porter. An algorithm for suffix stripping. *Program*, 14, 1980.

[138] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*, pages 254–265, New York, USA, 2014. ACM.

[139] Craig Quiter. Deepdrive, 2022. Retrieved on 2022.

[140] Philip Resnik. Using information content to evaluate semantic similarity in a taxonomy. *arXiv preprint cmp-lg/9511007*, 1995.

[141] Erik Rogstad, Lionel Briand, and Richard Torkar. Test case selection for black-box regression testing of database applications. *Information and Software Technology*, 55(10):1781–1795, 2013.

[142] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. Lgsvl simulator: A high fidelity simulator for autonomous driving. *arXiv preprint arXiv:2005.03778*, 2020.

[143] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. Reconstructing core dumps. In *2013 IEEE Sixth Int. Conf. on Software Testing, Verification and Validation*, pages 114–123. IEEE, 2013.

[144] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[145] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 34–43. IEEE, 1998.

[146] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 179–188. IEEE, 1999.

[147] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. Deeptc-enhancer: Improving the readability of automatically generated tests. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, pages 287–298, 2020.

[148] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Computer Vision*, 115(3):211–252, 2015.

[149] Abhinav Sagar and RajKumar Soundrapandiyan. Semantic segmentation with multi scale spatial attention for self driving cars. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2650–2656, 2021.

[150] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. An information retrieval approach for regression test prioritization based on program changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 268–279. IEEE, 2015.

[151] Martin Serpell and James E. Smith. Self-Adaptation of Mutation Operator and Probability for Permutation Representations in Genetic Algorithms. *Evolutionary Computation*, 18(3):491–514, 09 2010.

[152] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[153] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *International Conference on Automated Software Engineering (ASE)*, pages 201—-211. IEEE, 2015.

[154] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

[155] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *Proceedings of International Symposium on Empirical Software Engineering*, 2005.

[156] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[157] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. *jRapture: A capture/replay tool for observation-based testing*. ACM, 2000.

[158] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11(4):341–359, dec 1997.

[159] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, oct 2019.

[160] Wen Sun, Jiankai Jin, and Weisi Lin. Minimum Noticeable Difference based Adversarial Privacy Preserving Image Generation, 2022.

[161] Takahiro Suzuki, Shingo Takeshita, and Satoshi Ono. Adversarial Example Generation using Evolutionary Multi-objective Optimization. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 2136–2144, 2019.

[162] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[163] Christian Szegedy, Wojchiech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

[164] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 303–314, New York, NY, USA, 2018. Association for Computing Machinery.

[165] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[166] Joakim Verona. *Practical DevOps*. Packt Publishing Ltd, 2016.

[167] Tanapuch Wanwarang, Nataniel P. Borges, Leon Bettscheider, and Andreas Zeller. Testing apps with real-world inputs. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 1–10. ACM, 2020.

[168] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering (ICSE'09)*, pages 364–374, Vancouver, Canada, 2009. IEEE.

[169] Zhibiao Wu and Martha Palmer. Verb semantics and lexical selection. *arXiv preprint cmp-lg/9406033*, 1994.

[170] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4(6):2, 2000.

[171] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31:1–31:40, October 2013.

[172] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2016.

[173] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, March 2012.

[174] Shin Yoo and Mark Harman. Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, 83(4):689–701, 2010.

[175] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021.

[176] Benwen Zhang, Emily Hill, and James Clause. Towards automatically generating descriptive names for unit tests. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, pages 625–636. ACM, 2016.

[177] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 35th International Conference on Software Engineering*, pages 192–201. IEEE, 2013.

[178] Pengcheng Zhang, Bin Ren, Hai Dong, and Qiyin Dai. Cagfuzz: Coverage-guided adversarial generative fuzzing testing for image-based deep learning systems. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[179] Qingfu Zhang and Hui Li. Moea/d: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on evolutionary computation*, 11(6):712–731, 2007.

[180] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2881–2890, 2017.

[181] Yutian Zhou, Yu-an Tan, Quanxin Zhang, Xiaohui Kuang, Yahong Han, and Jingjing Hu. An evolutionary-based black-box attack to deep neural network classifiers. In *Mobile Networks and Applications*, 2021.

[182] Zhi Quan Zhou, Arnaldo Sinaga, and Willy Susilo. On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 5584–5593. IEEE, 2012.