Automated Repair of Feature Interaction Failures in Automated Driving Systems

Raja Ben Abdessalem
University of Luxembourg
Luxembourg
raja.benabdessalem@uni.lu

Annibale Panichella
Delft University of Technology
Netherlands
University of Luxembourg
Luxembourg
a.panichella@tudelft.nl

Shiva Nejati
University of Ottawa
Canada
University of Luxembourg
Luxembourg
snejati@uottawa.ca

Lionel C. Briand
University of Luxembourg
Luxembourg
University of Ottawa
Canada
lionel.briand@uni.lu

Thomas Stifter
IEE S.A.
Luxembourg
thomas.stifter@iee.lu

ABSTRACT

In the past years, several automated repair strategies have been proposed to fix bugs in individual software programs without any human intervention. There has been, however, little work on how automated repair techniques can resolve failures that arise at the system-level and are caused by undesired interactions among different system components or functions. Feature interaction failures are common in complex systems such as autonomous cars that are typically built as a composition of independent features (i.e., units of functionality). In this paper, we propose a repair technique to automatically resolve undesired feature interaction failures in automated driving systems (ADS) that lead to the violation of system safety requirements. Our repair strategy achieves its goal by (1) localizing faults spanning several lines of code, (2) simultaneously resolving multiple interaction failures caused by independent faults, (3) scaling repair strategies from the unit-level to the system-level, and (4) resolving failures based on their order of severity. We have evaluated our approach using two industrial ADS containing four features. Our results show that our repair strategy resolves the undesired interaction failures in these two systems in less than 16h and outperforms existing automated repair techniques.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Search-based software engineering**.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397386

KEYWORDS

Search-based Software Testing, Automated Driving Systems, Automated Software Repair, Feature Interaction Problem

ACM Reference Format:
Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2020. Automated Repair of Feature Interaction Failures in Automated Driving Systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3395363.3397386

1 INTRODUCTION

Software for automated driving systems (ADS) is often composed of several units of functionality, known as ADS features, that automate specific driving tasks (e.g., automated emergency braking, automated traffic sign recognition, and automated cruise control). Each feature executes continuously over time and receives data from the perception layer components. The latter often use artificial intelligence (e.g., deep neural networks [29]) to extract —based on sensor data— environment information, such as the estimated position of the (ego) car, the road structure and lanes, and the position and speed of pedestrians. Using this information, ADS features independently determine the car maneuver for the next time step.

Some maneuvers issued by different features may be conflicting. For example, automated cruise control may order the (ego) car to accelerate to follow the speed of the leading car while, at the same time, the automated traffic sign recognition feature orders the (ego) car to stop since the car is approaching an intersection with a traffic light that is about to turn red. To resolve conflicting maneuvers, *integration rules* are applied. Integration rules are conditional statements that determine under what conditions which feature outputs should be prioritized. They are typically defined manually by engineers based on their domain knowledge and are expected to ensure safe and desired car maneuvering. In the above example, integration rules should prioritize the braking maneuver of automated traffic sign recognition when the car can stop safely

before the intersection, and otherwise should prioritize the maneuver of the automated cruise control. Integration rules tend to become complex as they have to address a large number of different environment and traffic situations and predict how potential conflicts should be resolved in each situation. As a result, the rules may be faulty or incomplete, and there is a need to assist engineers to *repair* integration rules so that, at each time step and for each situation, they prioritize the maneuver that can ensure ADS safety.

Recently, many approaches have been proposed in the literature to repair faults in software code automatically [15, 22, 37, 44]. The inputs to these techniques are a faulty program and a test suite that contains some passing test cases capturing the desired program behavior and some failing test cases revealing a fault that needs to be fixed. Fault-repair techniques iteratively identify faulty statements in the code (fault localization [20, 45]), automatically modify the faulty statements (patch generation) and check if the patched code passes all the input test cases (patch validation). To effectively repair ADS integration faults, we need a technique, that in addition to supporting the three above steps, ensures the following objectives:

O1. The repair technique should localize faults spanning multiple lines of code. Most program repair techniques localize faults using spectrum-based techniques. These techniques rely on the assumption that there is a single faulty statement in the program [45, 46], and as shown by Pearson et al. [36], they may not identify all defective lines in multi-statements faults. As we describe in Section 3, faults in integration rules are often multi-location as they may span multiple lines of code, for example, when the rules are applied in a wrong order.

O2. The repair technique should be able to fix multiple independent faults in the code. Most program repair techniques assume that all failing test cases in the given (input) test suite fail due to the same fault in the code. Hence, they generate a single patch for a single fault at a time [28, 38, 43]. The generated patches may replace a single program statement [28, 38, 43] or they may be built based on predefined templates, prescribing hard-coded change patterns to modify programs in multiple locations [30, 31]. For ADS integration rules, however, the assumption that there is a single fault in the code does not hold since test cases may fail due to the presence of multiple faults located in different parts of the code. As we describe in Section 3, to fix all the failing test cases in a given test suite, we may have to fix more than one fault in the code by, for example, reordering some rules, and in addition, changing some clauses in preconditions of some other rules.

O3. The repair technique should scale to fixing faults at system-level where the software under test is executed using a feedback-loop simulator. Testing ADS software requires a simulation-based test platform that can execute the ADS software for different road structures, weather conditions and traffic situations. Testing ADS software using simulators presents two challenges: First, each test case is a scenario that executes over a time period and runs the ADS software through a feedback-loop simulation. Hence, a single test case runs the ADS software at every simulation time step and generates an array of outputs over time instead of a single output. As a result, dependencies in ADS software are both structural (like typical code) and temporal–*a change in the output at time t impacts the input of the system at time $t+1$ through the feedback-loop simulation which may impact the behaviour of the system in all subsequent time steps after t*. Second, test cases for ADS software are computationally expensive since each test case has to execute for a time duration and has to execute a complex simulator and the ADS software at each time step. This poses limitations on the sizes of test suites used for repair and the number of intermediary patches that we can generate.

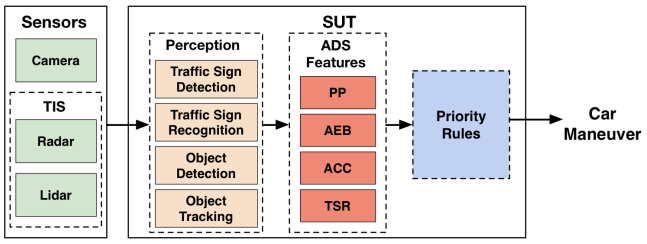

Figure 1: An overview of a self-driving system, *AutoDrive*, consisting of four ADS features.

O4. The repair technique should resolve failures in their order of severity. When testing generic software, test cases are either passing or failing, and there is no factor distinguishing failing test cases from one another. For ADS, however, some failures are more critical than others. For example, a test case violating the speed limit by 10km/h is more critical than the one violating the speed limit by 5km/h. Similarly, a test scenario in which a car hits a pedestrian with a speed lower than 10km/h is less critical than the ones where collisions occur at a higher speed. Our repair strategy should, therefore, assess failing test cases in a more nuanced way and prioritizes repairing the more critical failing test cases over the less critical ones.

Contributions. We propose an approach to Automated Repair of IntEgration ruLes (ARIEL) for ADS. Our repair strategy achieves the four objectives **O1** to **O4** described above by relying on a many-objective, single-state search algorithm that uses an archive to keep track of partially repaired solutions. We have applied ARIEL to two industry ADS case studies from our partner company [name redacted]. Our results show that ARIEL is able to repair integration faults in our two case studies in five and eleven hours on average. Further, baseline repair strategies based on Genetic Programming [26] and Random Search either never succeed to do so or repair faults with a maximum probability of 60% across different runs, i.e., in 12 out of 20 runs. Finally, we report on the feedback we received from engineers regarding the practical usefulness of ARIEL. The feedback indicates that ARIEL is able to generate patches to integration faults that are understandable by engineers and could not have been developed by them without any automation and solely based on their expertise and domain knowledge.

Structure. Section 2 provides motivation and background. Section 3 presents our approach. Section 4 describes our evaluation. Section 6 discusses the related work. Section 7 concludes the paper.

2 MOTIVATION AND BACKGROUND

We first motivate our work using a self-driving system case study, *AutoDrive*, from our industry partner (company A). We then formalize the concepts required in our approach.

2.1 Motivating Case Study

Figure 1 shows an overview of *AutoDrive* that consists of four ADS features: Autonomous Cruise Control (ACC), Traffic Sign Recognition (TSR), Pedestrian Protection (PP), and Automated Emergency Braking (AEB). ACC automatically adjusts the car speed and direction to maintain a safe distance from the leading car. TSR detects traffic signs and applies appropriate braking, acceleration, or steering commands to follow the traffic rules. PP detects pedestrians in front of a car with whom there is a risk of collision and applies a braking command if needed. AEB prevents accidents with objects other than pedestrians. Since these features generate braking, acceleration and steering commands to control the vehicle's actuators, they may send conflicting commands to the actuators at the same time. For example, ACC orders the car to accelerate to follow the speed of the leading car, while, at the same time, a pedestrian starts crossing the road. Hence, PP starts sending braking commands to avoid hitting the pedestrian.

Automated driving systems use integration rules to resolve conflicts among multiple maneuvers issued by different ADS features. The rules determine what feature output should be prioritized at each time step based on the environment factors and the current state of the system. Figure 2 shows a *small subset* of integration rules for AutoDrive. The rules are simplified for the sake of illustration. Each rule activates a single ADS feature (i.e., prioritizes the output of a feature over the others) when its *precondition*, i.e., the conjunctive clauses on the left-hand side of each rule, holds. All the rules are checked at every time step t, and the variables in the rules are indexed by time to indicate their values at a specific time step t. The rules in Figure 2 include the following environment variables: *pedestrianDetected* determines if a pedestrian with whom the car may risk a collision is detected; *ttc* or time to collision is a well-known metric in self-driving systems measuring the time required for a vehicle to hit an object if both continue with the same speed [40]; $dist(p, c)$ denotes the distance between the car c and the pedestrian p; $dist(c, sign)$ denotes the distance between the car c and a traffic sign *sign*; *speed* is the speed of the ego car, i.e., the car with the self-driving software; *speedLeadingCar* is the speed of the car in front of the ego car; *objectDetected* determines if an object which cannot be classified as a pedestrian is detected in front of the car, and *stopSignDetected* that determines if a stop sign is detected. For example, **rule1** states that feature PP is prioritized at time t if a pedestrian in front of the car is detected, the car and pedestrian are close ($dist(p, c)(t) < dist_{th}$) and a collision is likely ($ttc(t) < ttc_{th}$). Note that $dist_{th}$, $dist_{cs}$ and ttc_{th} are threshold values. Dually, **rule2** states that if there is a risk of collision with another object different from a pedestrian ($ttc(t) < ttc_{th} \wedge objectDetected(t) \wedge \neg pedestrianDetected(t) \wedge objectDetected(t)$), then AEB should be applied. Note that the braking command issued by AEB is less intense than that issued by PP.

Integration rules are likely to be erroneous, leading to system safety requirements violations. In particular, we identify two general ways where the integration rules may be wrong: (1) The preconditions of the rules may be wrong. Specifically, there might be missing clauses in the preconditions, or the thresholds used in some clauses may not be accurate or there might be errors in mathematical or relational operators used in the preconditions (i.e., using $<$ instead of $>$ or $+$ instead of $-$). For example, if the threshold $dist_{th}$ in **rule1** is too small, we may activate PP only when the car is too close to the pedestrian with little time left to avoid an accident. (2) The integration rules may be applied in a wrong order. Applying the rules in different orders leads to different system behaviors, some of which may violate safety requirements. For example, **rule3** and **rule4** can be checked in two different orders. However, applying **rule3** first may lead to a scenario where the car keeps increasing its speed since ACC orders to follow a fast driving leading car, and it violates the speed-limit as **rule4**, being at a lower priority than **rule3**, is never checked and cannot activate TSR. To avoid this situation, **rule4** should be prioritized over **rule3**.

At every time step t apply:

rule1: $(ttc(t) < ttc_{th}) \wedge (dist(p, c)(t) < dist_{th}) \wedge pedestrianDetected(t) \implies$ PP.active(t)

rule2: $(ttc(t) < ttc_{th}) \wedge !pedestrianDetected(t) \wedge objectDetected(t) \implies$ AEB.active(t)

rule3: $speed(t) < speedLeadingCar(t) \implies$ ACC.active(t)

rule4: $(speed(t) >$ speed-limit$) \wedge (dist(c, sign)(t) < dist_{cs}) \implies$ TSR.active(t)

rule5: $stopSignDectected(t) \implies$ TSR.active(t)

Figure 2: Ordered integration rules to resolve conflicts between different ADS features.

2.2 Context Formalization

We define an ADS as a tuple $(f_1, \ldots, f_n, \Pi)$, where $f_1, \ldots, f_n$ are features and Π is a decision tree diagram capturing the system integration rules. For example, Figure 3 shows a fragment of a decision tree diagram related to the rules in Figure 2. Each internal node of Π corresponds to a clause of a rule precondition and the tree leaves are labelled by features. Specifically, each path π of Π corresponds to one integration rule $precondition \implies f$ such that the leaf of π is labelled by f and the conjunction of clauses appearing on the non-leaf nodes of π are equal to *precondition*. We denote by $f(\pi_i)$ the label of the leaf of π_i, i.e., the feature activated when π_i is executed. A set Π of integration rules are implemented using a set S of program statements. Each statement $s \in S$ corresponds to a node of some path $\pi \in \Pi$. We use the notation $s \in \pi$ when statement s corresponds to some node on π. The conditional statements correspond to non-leaf nodes of Π and have this general format: ⟨`if op1 operator threshold`⟩ where `op1` is a variable, `operator` is a relational operator and `threshold` is boolean or numeric constant (see Figure 3)

Let $SR = \{r_1, \ldots, r_k\}$ be a set of safety requirements. Due to the modular structure of ADS, each requirement is related to one feature, which is responsible for the satisfaction of that requirement. For example, Table 1 shows the set of requirements for *AutoDrive* and the feature responsible for satisfying each requirement.

We denote by $TS = \{tc_1, \ldots, tc_l\}$ a set of test cases for an ADS. Given that ADS testing is performed via simulators, each test tc_i specifies initial conditions and parameters for the simulator to mimic a specific self-driving scenario. To run a test case, the simulator that is initialized using a test case is executed for a time duration T set by the user. The simulator computes the state of the system at regular and fine-grained steps. Let δ be the simulation

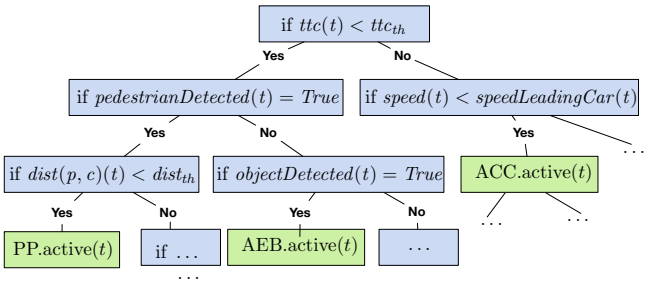

Figure 3: A fragment of the decision tree diagram related to the integration rules in Figure 2.

Table 1: Safety requirements for *AutoDrive*.

Features	Requirements
PP	The PP system shall avoid collision with pedestrians by initiating emergency braking in case of impending collision with pedestrians.
TSR	The TSR system shall stop the vehicle at the stop sign by initiating a full braking when a stop sign is detected
AEB	The AEB system shall avoid collision with vehicles by initiating emergency braking in case of impending collision with vehicles.
ACC	The ACC system shall respect the safety distance by keeping the vehicle within a safe distance behind the preceding vehicle.
TSR	The TSR system shall respect the speed limit by keeping the vehicle's speed at or below the road's speed limit.

time step size. The simulator computes the test results as vectors of size q where $T = q \cdot \delta$. Each element i s.t. $0 \leq i \leq q$ of these vectors indicates the test output at the ith time step.

At each time step $u \in \{0, \ldots, q\}$, each test case tc executes the integration rules Π that select a single feature to be applied at that step. We write $tc(u)$ to refer to the output of tc at step u. We define an oracle function O to assess test outputs with respect to the system safety requirements SR. Specifically, $O(tc_i(u), r_j)$ assess the output of tc_i at time step u with respect to requirement r_j. We define O as a quantitative function that takes a value between $[-1, 1]$ such that a negative or zero value indicates that the test case fails and a positive value implies that the test case passes. The quantitative oracle value allows us to distinguish between different degrees of satisfaction and failure (objective **O4**). When O is positive, a closer value to 0 indicates that we are close to violating, although not yet violating, some safety requirement, while a value close to 1 shows that the system is far from violating its requirements. Dually, when O is negative, an oracle value close to 0 shows a less severe failure than a value close to -1. We define the oracle function O for every requirement $r_j \in \mathcal{SR}$ separately such that O ensure the above properties. For example, for the safety requirement related to PP (denoted r_{PP})), we define $O(tc_i(u), r_{PP})$ for every test case tc_i and every time step u as a normalized value of $(dist(p,c)(u \cdot \delta) - dist_{fail})$ where $dist(p,c)$ is the same variables used in **rule1** in Figure 2, and $dist_{fail}$ is a threshold value indicating the smallest distance allowed between a car and a pedestrian below which r_{PP} is violated.

In our work, we select the test suite TS such that it can cover all the paths in Π. That is, for each path $\pi_i \in \Pi$ there is some test case $tc_j \in TS$ and some time step u such that $tc_j(u)$ executes π_i, i.e., test case tc_j executes the path π_i at step u. Given a test case $tc_j \in TS$, we say tc_j fails, if it violates some safety requirement at some time step u, i.e., if there is a requirement r_j and some time step u such that $O(tc_i(u), r_j) \leq 0$. We refer to the time step u at which tc_j fails as *the time of failure for* tc_j. Otherwise, we say tc_j passes. We denote by TS_f and TS_p the set of failing and passing test cases, respectively. In ADS, a test case may fail due to a fault either in a feature or in integration rules. In order to ensure that a failure revealed by a test case tc is due to a fault in integration rules, we can remove integration rules and test features individually using tc. If the same failure is not observed, we can conclude that integration rules are faulty. In addition, we can combine our oracle function O with heuristics provided by the feature interaction analysis research [9] to distinguish failures caused by integration rules from those caused by errors inside features.

3 APPROACH

In this section, we present our approach, Automated Repair of IntEgration ruLes (ARIEL), to locate and repair faults in the integration rules of ADS. The inputs to ARIEL are (1) a faulty ADS $(f_1, \ldots, f_n, \Pi)$ and (2) a test suite TS verifying the safety requirements $SR = \{r_1, \ldots, r_k\}$. The output is a repaired ADS, i.e., $(f_1, \ldots, f_n, \Pi^*)$, where Π^* are the repaired integration rules. Below, we present different steps of ARIEL in detail.

3.1 Fault Localization

Feature interaction failures may be caused by multiple independent faults in integration rules, and some of these faults may span over multiple statements of the code of integration rules. As shown by a prior study [36], state-of-the-art fault localization methods often fail to pinpoint all faulty lines (i.e., those to mutate in program repair). Further, recall from Section 1 that to test ADS, we have to execute the system within a continuous loop over time. Hence, each ADS test case covers multiple paths of Π over the simulation time duration, i.e., one path $\pi \in \Pi$ is covered at each time step u. Hence, failing and non-failing tests cover large, overlapping subsets of paths in Π albeit with different data values. In this case, applying existing fault localization formulae (e.g., Tarantula and Ochiai) results in having most statements in Π with the same suspiciousness score, leaving the repair process with little information as to exactly where in Π mutation/patches should be applied.

Based on these observations, we modify the Tarantula formula by (1) focusing on the path $\pi \in \Pi$ covered at the time of failure, and (2) considering the "severity" of failures. There are alternative fault localization formulae that we could use. In this paper, we focus on Tarantula since a previous study [36] showed that the differences among alternative methods (e.g., Tarantula and Ochiai) are negligible when applied to real faults. Below, we describe how we modified Tarantula to rank the faulty paths in Π.

3.1.1 Localizing the Faulty Path. Given a failure revealed by a test case $tc_i \in TS$, the goal is to determine the path executed by tc_i that is more likely to have caused the failure. At each time step u, one path π_i is executed, and only one feature (i.e., $f(\pi_i)$) is selected. A failure that arises at time step u' is likely to be due to faults in the integration rules of the path π_i that is executed at u'. As discussed in Section 2, we distinguish two types of faults: (1) *wrong conditions*, that are faults in the rules' preconditions, e.g., the braking command is activated when the distance between the car

and the pedestrian is too small, and (2) *wrong ordering of rules*, that are faults due to incorrect ordering of the rules activating features. For the purpose of fault localization, we only consider the path $\pi_i \in \Pi$ executed at the time of the failure (u') as the execution trace for fault localization, and ignore all other paths covered by a failing test case before and after the failure. In contrast, in traditional fault localization, all statements (rules in our case) covered by the test cases should be considered.

3.1.2 The Severity of the Failure. In Tarantula (as well as in state-of-the-art fault localization techniques), all failing tests have the same weight (i.e., one) in the formula for computing statement suspiciousness. However, in case of multiple faults, this strategy does not help prioritize the repair of the (likely) faulty statements. Focusing on faulty statements related to the most severe failures can lead to patches with the largest potential gains in fitness. As explained in Section 2, we can measure how severely a safety requirement of the ADS is violated. The severity corresponds to the output $O(tc_i(u), rj)$ of the test case tc_i at the time stamp u. For example, let us consider the safety requirement: "*the minimum distance between the ego car and the pedestrian must be larger than a certain threshold* $dist_{fail}$". A failure happens when the distance between the car and the pedestrian falls below the threshold $dist_{fail}$. This corresponds to having a test output $O(tc_i(u), r_{PP})$ within the interval $[-1; 0]$ for the test case tc_i and requirement r_{PP}. The lower the output value, the larger the severity of the violations for r_{PP}.

3.1.3 Our Fault Localization Formula. Based on the observations above, we define the suspiciousness of each statement s by considering both failure severity and faulty paths executed by failing test cases when the tests fail. Note that for each failing test case tc, there is a unique time step u at which tc fails (i.e., when we have $O(tc(u), r) \leq 0$ for some requirement r). The execution of tc stops as soon as it fails. In fact, the time of failure for a failing test case tc is the last time step of the execution of tc. Given a failing test $tc \in TS_f$, we write $cov(tc, s) \in \{0, 1\}$ to denote whether, or not, tc covers a statement s at its last step (when it fails). Further, we denote by w_{tc} the weight (severity) of the failure of tc. We then compute the suspiciousness of each statement s as follows:

$$Susp(s) = \frac{\frac{\sum_{tc \in TS_f}[w_{tc} \cdot cov(tc,s)]}{\sum_{tc \in TS_f} w_{tc}}}{\frac{passed(s)}{total\_passed} + \frac{failed(s)}{total\_failed}} \quad (1)$$

where $passed(s)$ is the number of passed test cases that have executed s at some time step; $failed(s)$ is the number of failed test cases that have executed s at some time step; and $total\_passed$ and $total\_failed$ denote the total numbers of failing and passing test cases, respectively. Note that Equation 1 is equivalent to the standard Tarantula formula if we let the weight (severity) for failing test cases be equal to one (i.e., if $w_{tc} = 1$ for every $tc \in TS_f$):

$$Susp(s) = \frac{\frac{failed(s)}{total\_failed}}{\frac{passed(s)}{total\_passed} + \frac{failed(s)}{total\_failed}} \quad (2)$$

For each test case tc that fails at time step u and violates some requirement r, we define $w_{tc} = |O(tc(u), r)|$. That is, w_{tc} is the degree of violation caused by tc at the time step u when it fails. Hence, we assign larger weights (larger suspiciousness) to statements covered by test cases that lead to more severe failures (addressing **O4** in Section 1). Note that each test case violates at most one requirement since the simulation stop as soon as a requirement is violated.

3.2 Program Repair

Traditional evolutionary tools, such as GenProg, use population-based algorithms (e.g., Genetic programming), which evolves a pool of candidate patches iteratively. Each candidate patch is evaluated against the entire test suite to check whether changes lead to more or fewer failing tests. Hence, the overall cost of one single iteration/generation is $N \times \sum_{tc \in TS} cost(tc)$, where $cost(tc)$ is the cost of the test tc and N is the population size.

Population-based algorithms assume that the cost of evaluating individual patches is not large, and hence, test results can be collected within a few seconds and used to give feedback (i.e., compute the fitness function) to the search. However, as discussed in **O3** in Section 1, this assumption does not hold in our context, and it takes in the order of some minutes to run a single simulation-based test case. Hence, evaluating a pool of patches in each iteration/generation becomes too expensive (in the order of hours).

Based on the observations above, we opted for the (1+1) Evolutionary Algorithm (EA) with an archive. (1+1) EA selects only one parent patch and generates only one offspring patch in each generation. Generating and assessing only one patch at a time allows to reduce the cost of each iteration, thus addressing **O3** in Section 1. Our program repair approach, ARIEL, includes (i) (1+1) EA, (ii) our fault localization formula (Equation 1), and (iii) the archiving strategy. Algorithm 1 provides the pseudo-code of ARIEL.

ARIEL receives as input the test suite TS and the faulty self-driving system $(f_1, \ldots, f_n, \Pi)$, where Π denotes the faulty integration rules. The output are a set of repaired integration rules (denoted by Π^*) that pass all test cases in TS. The algorithm starts by initializing the *archive* with the faulty rules Π (line 2) and computing the objective scores, which we describe in Section 3.2.3 (line 3). Then, the archive is evolved iteratively within the loop in lines 4-8. In each iteration, ARIEL selects one patch $\Pi_p \in archive$ randomly (line 5) and creates one offspring (Π_o) in line 6 (routine GENERATE-PATCH) by (1) applying fault localization (Equation 1) and (2) mutating the rules in Π_p. The routine GENERATE-PATCH is presented in subsection 3.2.1.

Then, the offspring Π_o is evaluated (line 7) by running the test suite TS, extracting the remaining failures, and computing their corresponding objective scores (Ω). Note that the severities of the failures are our search objectives and are further discussed in Section 3.2.3. The offspring Π_o is added to the *archive* (line 8 of Algorithm 1) if it decreases the severity of the failures compared to the patches currently stored in the *archive*. The *archive* and its updating routine are described in details in subsection 3.2.4. The search stops when the termination criteria are met (see Section 3.2.5).

3.2.1 Generating a Patch. ARIEL generates patches using the routine in Algorithm 2. First, it applies our fault localization formula (routine FAULT-LOCALIZATION in line 2) to determine the suspiciousness of the statements in Π and based on the test results.

To select a statement $s \in \Pi$ among the most suspicious ones, we use the Roulette Wheel Selection (RWS) [16], which assigns

Algorithm 1: ARIEL

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

```
1 begin
2     Archive ⟵ Π
3     Ω ⟵ RUN-EVALUATE(Π, TS)
4     while not(|Archive|==1 & Archive satisfies all tc ∈ TS) do
5         Π_p ⟵ SELECT-A-PARENT(Archive)      // Random selection
6         Π_o ⟵ GENERATE-PATCH(Π_p, TS, Ω)
7         Ω ⟵ RUN-EVALUATE(Π_o, TS)
8         Archive ⟵ UPDATE-ARCHIVE(Archive , Π_o, Ω)
9     return Archive
```

Algorithm 2: GENERATE-PATCH

Input:
Π: A faulty rule-set
TS: Test suite
Ω: Severity of failures (Objectives Scores)
Result: Π_o: A mutated Π

```
1  begin
2      {π, s} ⟵ FAULT-LOCALIZATION(Π, TS, Ω)      // Section 3.1
3      counter ⟵ 0
4      p = RANDOM-NUMBER(0,1)                      // p ∈ [0, 1]
5      Π_o ⟵ Π
6      while p ≤ 0.5^counter do
7          Π_o ⟵ APPLY-MUTATION(Π_o, π, s)
8          counter ⟵ counter + 1
9          p= RANDOM-NUMBER(0,1)                   // p ∈ [0, 1]
10     return Π_o
```

each statement a probability of being selected for mutation. The probabilities are based on the suspiciousness of each statement (Equation 1). More specifically, the probability of a statement $s \in \Pi$ is $prob(s) = \frac{Susp(s)}{\sum_{s_i \in \Pi} Susp(s_i)}$. The higher the suspiciousness of a statement s, the higher its probability of being selected for mutation. The routine FAULT-LOCALIZATION returns (1) the statement s to be mutated, which is randomly selected based on RWS, and (2) a path $\pi \in \Pi$ such that $s \in \pi$ (i.e., s is on path π) and π is executed by some failing test case at its last time step (i.e., π is executed at the time of failure of some test case). If several paths satisfy the latter condition, FAULT-LOCALIZATION returns one randomly.

Once a (likely) faulty statement s is selected, Algorithm 2 applies multiple mutations within the loop in lines 6-9. First, one mutation is applied with probability $p = 0.5^0 = 1$ using the routine APPLY-MUTATION (line 7). Then, a second mutation is applied with probability $p = 0.5^1$, a third mutation with probability $p = 0.5^2 = 0.25$, and so on. Therefore, in each iteration of the loop in lines 6-9, a mutation is applied with probability $p = 0.5^{counter}$, where $counter + 1$ is the number of executed iterations. In the first iteration, $p = 1$, and hence, it is guaranteed that at least one mutation is performed. Such a formula has been previously used in the context of test case generation [14]. Section 3.2.2 describes the mutation operators used to generate candidate patches. These operators receive as input π and s produced by the fault localization step.

3.2.2 Mutation Operators. We define two search operators *modify* and *shift* based on the types of errors that can occur in integration

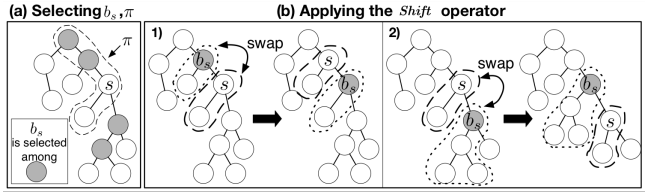

Figure 4: Illustrating the shift operator: (a) selecting b_s and path π, and (b) applying the shift operator.

rules of an ADS. *Modify* modifies the conditions in the non-leaf nodes while *shift* switches the order of the rules. The operators are defined below:

Modify. Let s be the decision statement (or non-leaf node) selected for mutation (based on its suspiciousness). As discussed in Section 2, s has the following form: ⟨`if op1 operator threshold`⟩. The operator *modify* performs three types of changes: (1) changing a threshold value, (2) altering the direction of a relational operator (e.g., $\leq$ to $\geq$), or (3) changing arithmetic operations (e.g., + to −). Note that the left operand `op1` may be an arithmetic expression involving some arithmetic operators and variables.

Shift. Given the path $\pi \in \Pi$ executed at the time of the failure, the *shift* operator changes the order of rules in π. Let $s \in \pi$ be the decision statement (or non-leaf node) selected using the fault localization formula. The mutation swaps s with a randomly selected node that either dominates s (i.e., it precedes s in π) or is dominated by s (i.e., it succeeds s in π). Dominators and post-dominators of a node s are highlighted in grey in Figure 4(a). Among all dominators and post-dominators, one node b_s is selected randomly. We identify two cases: (Case1) b_s is a dominator of s. In this case, the shift operator swaps b_s with s such that the former becomes a post-dominator of the latter. Further, the operator shifts up the sub-path from s to the leaf node in π (see Figure 4(b-1) for an illustration). This is because features activated based on the condition in node s may not be consistent with the condition in node b_s or vice versa. Therefore, the nodes s and b_s should be swapped together with the sub-paths connecting them to their leaves (i.e., to their corresponding features). (Case2) b_s is a post-dominator of s. The shift operator swaps b_s with s. As before, the sub-path from s to the leaf node of π is also shifted down with s (see Figure 4(b-2) for an illustration).

3.2.3 Search Objectives. In our approach, the search objectives are based on the failures exposed by the test suite TS. Each failing test $tc \in TS$ exposes one (or more) failure(s) whose intensity is $O(tc(u), r_j) \in [-1; 0]$, where r_j is the violated safety requirements. Recall that smaller $O(tc(u), r_j)$ values indicate more severe violations of r_j. Since ADS can have multiple failures/violations, the program repair problem can be formulated as a many-objective optimization problem, whose search objectives are:

$$\begin{cases} \max \Omega_1(\Pi) = \min\limits_{tc \in TS} \{O(tc(u), r_1)\} \\ \ldots \\ \max \Omega_k(\Pi) = \min\limits_{tc \in TS} \{O(tc(u), r_k)\} \end{cases} \quad (3)$$

where $\min\limits_{tc \in TS} \{O(tc(u), r_i)\}$ correspond to the most severe failure among all $tc \in TS$ for requirement r_i. The concept of optimality in many-objective optimization is based on the concept of *dominance* [32]. More precisely, *a patch* Π_1 *dominates another patch* Π_2

if and only if $\Omega_i(\Pi_1) \geq \Omega_i(\Pi_2)$ *for all* Ω_i *and there exists one objective* Ω_j *such that* $\Omega_j(\Pi_1) > \Omega_j(\Pi_2)$. In other words, Π_1 dominates Π_2, if Π_1 is not worse than Π_2 in all objectives, and it is strictly better than Π_2 in at least one objective [32].

3.2.4 Archive. The archive stores the best partial fixes found during the search. At the beginning of the search (line 2 of Algorithm 1), the archive is initialized with the faulty rule set Π. Every time a new patch Π_o is created and evaluated, we compare it with all the patches stored in the archive. The comparison is based on the notion of dominance described in Section 3.2.3:

- If Π_o dominates one solution A in the archive, Π_o is added to the archive and A is removed from the archive.
- If no element in the archive dominates or is dominated by the new patch Π_o, the new patch is added to the archive as well.
- The archive remains unchanged if Π_o is dominated by one (or more) solution(s) stored in the archive.

Therefore, in each iteration, the archive is updated (lines 8 of Algorithm 1) such that it includes the non-dominated solutions (i.e., the best partial patches) found so far.

The number of non-dominated solutions stored in the archive can become extremely large. To avoid this *bloating effect*, we add an upper bound to the maximum size for the archive. In this paper, we set the size of the archive to $2 \times k$, where k is the number of safety requirements. If the size of the archive exceeds the upper-bound, the solutions in the archive are compared based on the aggregated fitness value computed as follows:

$$D(\Pi) = \sum_{tc \in TS} \left(\sum_{r_j \in SR} O(tc(u), r_j) \right)$$

Then, we update the archive by selecting the $2 \times k$ patches that have the largest D values.

One possible limitation of the evolutionary algorithms is that they can get trapped in some local optimum (premature convergence) [18]. A proven strategy to handle this potential limitation is to restart (re-initialize) the search [18]. We implement the following restarting policy in ARIEL: First, ARIEL uses a counter to count the number of subsequent generations with no improvements in the search objectives (i.e., the archive is never updated). If the counter reaches a threshold of h generations, the search is restarted by removing all partial patches in the archive and re-initializing it with the original faulty rule-set Π (see Section 4.3 for the value of h).

3.2.5 Termination Criteria. The search stops when the search budget is consumed or as soon as a test-adequate patch is found, i.e., the *archive* contains one single patch Π^* that satisfies all test cases $tc \in TS$. Note that even though our repair algorithm is multi-objective, it generates only one final solution. This is because our algorithm, being a single-state search, generates one solution in each iteration and stops if that solution is Pareto optimal, i.e., it dominates all partial patches in the archive.

Indeed, our repair strategy decomposes the single objective of finding a single patch that passes all test cases in TS into multiple sub-objectives of finding partial patches that pass a subset of test cases in TS. This search strategy is called *multi-objectivization* [17, 23] and is known to help promote diversity. As shown in prior work in numerical optimization, *multi-objectivization* can lead to better results than classical single objective approaches when solving/targeting the same optimization problem [23]. In our context, multi-objectivization helps to store partial patches that individually fix different faults by saving them in the *archive* (elitism). Such partial patches can be further mutated in an attempts to combine multiple partial patches, thus, building up the final patch/solution that fixes all faults. As shown in our evaluation in Section 4, our multi-objective approach outperforms single-objective search.

3.2.6 Patch Minimization. At the end of the search, ARIEL returns a patch that is test-adequate (i.e., it passes all tests), but it may include spurious mutations that are not needed for the fix. This problem can be addressed through a patch minimization algorithm [27], i.e., a greedy algorithm that iteratively removes one mutation at a time and verifies whether the patch after the reversed change still passes the test suite. If so, a smaller (minimized) patch that is still test-adequate is obtained.

3.2.7 How ARIEL Addresses **O1**-**O4***?* ARIEL includes several ingredients to address the challenges of repairing ADS integration rules described in Section 1. First, ARIEL applied many-objective search, where each objective corresponds to a different failure (**O2**). Through many-objective optimization, ARIEL can repair several failures simultaneously. **O1** and **O4** are addressed by our fault localization formula (Equation 1) that assigns the suspiciousness score based on (1) the paths covered at the time of each failure, and (2) the severity of failures (i.e., failing tests). In addition, our search objectives address **O4** by measuring how a generated patch affects the severity of the failures. Finally, EA(1+1) generates and evaluates only one patch (offspring) at a time rather than evolving a pool of patches (like in GP-based approaches), thus addressing **O3**.

4 EVALUATION

In this section, we evaluate our approach to repairing integration rules using industria ADS systems.

4.1 Research Questions

RQ1: *How effective is ARIEL in repairing integration faults?* We assess the effectiveness of ARIEL by applying it to industrial ADS systems with faulty integration rules.

RQ2: *How does ARIEL compare to baseline program repair approaches?* In this research question, we compare ARIEL with three baselines that we refer to as Random Search (RS), Random Search with Multiple Mutations (RS-MM) and Genetic Programming (GP). We compare ARIEL with the baselines by evaluating their effectiveness (the ability to repair faults) and efficiency (the time it takes to repair faults). Section 4.3 describes the details of the baselines and our rationale for selecting them.

We conclude our evaluation by reporting on the feedback we obtained from automotive engineers regarding the usefulness and effectiveness of ARIEL in repairing faults in integration rules.

4.2 Case Study Systems

In our experiments, we use two case study systems developed in collaboration with company A. These two systems contain the four self-driving features introduced in Section 2, but contain two different sets of integration rules to resolve conflicts among multiple maneuvers issued by these features. We refer to these systems as *AutoDrive1* and *AutoDrive2*. Their features and their integration

rules are implemented in the Matlab/Simulink language [1]. We use PreScan [39], a commercial simulation platform available for academic research to execute *AutoDrive1* and *AutoDrive2*. The PreScan simulator is also implemented in Matlab/Simulink and *AutoDrive1* and *AutoDrive2* can be easily integrated into PreScan.

AutoDrive1 and *AutoDrive2* include a TIS (Technology Independent Sensor) [33] sensor to detect objects and to identify their position and speed and a camera. The perception layer receives the data and images from the sensor and the camera, and consists, among others, tracking algorithms [13] to predict the future trajectory of mobile objects and a machine learning component that is based on support vector machine (SVM) to detect and classify objects. Our case study systems both include a Simulink model with around 7k blocks and 700k lines of Matlab code capturing the simulator, the four ADS features and the integration rules. Each decision tree for the integration rules of *AutoDrive1* and *AutoDrive2* has 204 rules (branches), and the total clauses in the rules' preconditions is 278 for each case study system. The set of rules of these two systems vary in their order and their threshold values. Syntactic diff shows 30% overlap between the ordered rules of *AutoDrive1* and *AutoDrive2*. The matlab/Simulink models of our case studies and our implementation of ARIEL are available online [2] and are also submitted alongside the paper.

AutoDrive1 and *AutoDrive2* also include test suites with nine and seven system-level test cases, respectively. The test suite for *AutoDrive1* contains four failing test cases, while the test suite for *AutoDrive2* has two failing tests. Each failing test case for *AutoDrive1* and *AutoDrive2* violates one of the safety requirements in Table 1. Each test case for *AutoDrive1* and *AutoDrive2* executes its respective ADS systems (including the rules) for 1200 simulation steps. The two system-level test suites have been manually designed by developers to achieve high structural coverage (i.e., 100% branch and statement coverage) on the code of the integration rules. The total test execution time for the test suites of *AutoDrive1* and *AutoDrive1* are 20 and 30 minutes, respectively. Furthermore, each test suite contains at least one passing test case that exercises exactly one of the five safety requirements shown in Table 1.

Note that each test case in the test suites of *AutoDrive1* and *AutoDrive2* executes the system for 1200 times. The test suites achieve full structural coverage over the integration rules while requiring 20min and 30min to execute. Hence, larger test suites were neither needed nor practical in our context. Further, *AutoDrive1* and *AutoDrive2* give rise to several complex feature interaction failures one of which is discussed in Section 4.4. As the results of RQ1 and RQ2 confirm, the feature interaction failures in our case study systems cannot be repaired by existing standard and baseline automated repair algorithms, demonstrating both the complexity of our case studies and the need for our approach, ARIEL.

4.3 Baseline Methods and Parameters

In this section, we describe our baseline methods: Random Search (RS), Random Search with Multiple Mutations (RS-MM) and Genetic Programming (GP) and the parameters used in our experiments.

Unlike ARIEL[1], our baselines use one objective, *baselineO*, similar to that used in most existing repair approaches [28, 44] and defined as follows: $baselineO = W_p \times total\_passed + W_f \times total\_failed$, where W_p and W_f are weights applied to the total number passing and failing test cases, respectively. The weight W_f is typically much larger than the weight W_p since typically there are more passing test cases than the failing ones in a test suite.

4.3.1 Random Search (RS). RS is a natural baseline to compare with because prior work [37] showed that it is particularly effective in the context of program repair, often outperforming evolutionary algorithms. RS does not evolve candidate patches but generates random patches by mutating the original rules Π only once using either the *modify* or *shift* operator. The final solution (patch) among all randomly generated patches is the one with the best *baselineO* value.

4.3.2 Random Search with Multiple Mutations (RS-MM) . RS-MM is an improved variant of RS where multiple mutations are applied to the faulty rule set rather than one single mutation as done in RS. In other words, RS-MM generates random patches using the same operator applied in ARIEL (Algorithm 2). Thus, with the second baseline, we aim to assess whether the differences (if any) between random search and ARIEL are due to the mutation operator or the evolutionary search we proposed in this paper.

4.3.3 Genetic Programming (GP). We apply **GP** to assess the impact of using a single-state many-objective search algorithm rather than classic genetic programming. Similar to the state-of-the-art automated repair algorithm [28], GP relies on genetic programming. It starts with a pool (population) of n randomly generated patches and evaluates them using the *baselineO* objective function, which is the single objective to optimize [28]. Instead, ARIEL evolves one single patch and starts with a singleton archive containing only Π (see line 2 in Algorithm 1). Further, at each iteration, GP generates n new offspring patches by applying the mutation operators to the parents patches. Then, it selects the best n elements among the new and old individuals to maintain constant the size of the population. Similar to ARIEL, GP applies mutation operators multiple times to each parent patch. Thus, GP and ARIEL share the same mutation operator (Algorithm 2). Note that GP does not use the crossover operator because the alternative integration rules in the population are incomparable (e.g., rules/trees with different roots and rules orderings).

4.3.4 Parameters. As suggested in the literature [28, 44], for the fitness function *baselineO* used in the three baseline methods RS, RS-MM, and GP, we set $W_f = 10$ and $W_p = 1$. For GP, the recommended population size used in the literature is 40 [28, 44]. However, computing the fitness for a population of 40 individuals takes on average around 20 hours for our case studies (i.e., one iteration of GP takes around 20 hours with a population size of 40). Hence, both for our experiments and in practice, it will take a long time to run GP for several iterations with a large population size. Therefore, we set the (initial) population size to 10. Note that for ARIEL, as described in Section 3.2.4, the archive size is dynamically updated at

[1]Recall that ARIEL uses multiple objectives (i.e., one objective per each safety requirement) to select the best element from an archive (see Section 3.2.3).

rule3: $speed(t) >$ speed-limit $\wedge \boxed{(dist(c, sign(t)) < dist_{cs} + \alpha)} \implies$ TSR.active(t)

rule4: $speed(t) < speedLeadingCar(t) \implies$ ACC.active(t)

Figure 5: Resolving a feature interaction failure between TSR and ACC in the set of rules in Figure 2. To resolve this failure, rule3 and rule4 in Figure 2 are swapped and the threshold value in the precondition of rule3 is modified.

each iteration and does not need to be set as a parameter. Based on some preliminary experiments, we set the parameter h, discussed in Section 3.2.4 to randomly restart the search, to eight.

To account for the randomness of ARIEL, GP, RS and RS-MM, we reran each of them for 20 times on each case study system. For RQ1, we ran ARIEL on both case study systems until ARIEL finds a patch that passes all the test cases in the given test suite. We then used the maximum time needed by ARIEL to find patches as a timeout for the baseline methods in RQ2. We ran all the experiments on a laptop with a 2.5 GHz CPU and 16GB of memory.

4.4 Results

RQ1. To answer this question, we apply ARIEL 20 times to *AutoDrive1* and *AutoDrive2* until a patch is found that passes all the test cases given as input. For both case study systems and for all the runs, ARIEL was able to repair the integration rules in the underlying ADS and terminate successfully. The box-plots in Figures 6(a) and (b) show the time needed for ARIEL to repair the integration rules of *AutoDrive1* and *AutoDrive2*, respectively, over 20 independent runs. As shown in the figures, ARIEL is able to repair all the integration faults in less than nine hours for *AutoDrive1* and in less than 16 hours for *AutoDrive2*. The average repair time for *AutoDrive1* and *AutoDrive2* is five hours and 11 hours, respectively.

We note that failures in AutoDrive1-2 were caused by independent faults in different parts of the code. Recall from Section 3.2 that in order to resolve failures, ARIEL fixes the following fault types in the integration rules: (I) wrong operator/threshold, (II) wrong rule ordering and (III) a combination of both (I) and (II). For AutoDrive1, three faults were of type (I) and one was of type (III). For AutoDrive2, both faults were of type (II).

Figure 5 shows an example output of ARIEL when applied to the rules in Figure 2 to resolve the feature interaction failure between ACC and TSR exhibited by AutoDrive1 and described in Section 2. In particular, the failure occurs when an ego car that is following a leading car with a speed higher than 50 reaches a traffic sign limiting the speed to 50. Since the rules in Figure 2 prioritize ACC over TSR, the car disregards the speed limit sign and continues with a speed above 50. The patch generated by ARIEL in Figure 5 swaps **rule3** and **rule4** in Figure 2, and in addition, modifies the threshold value for the pre-condition of the rule activating TSR. This ensures that TSR is applied before reaching the speed limit sign to be able to reduce the car speed below 50. Note that the passing test cases in the test suite of AutoDrive1 will ensure ACC is still activated in scenarios when it should be activated.

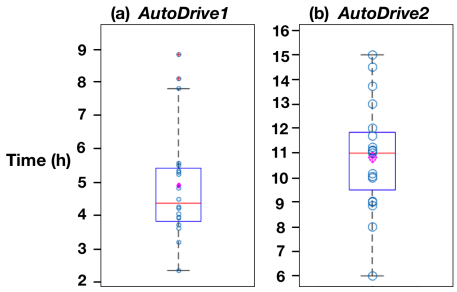

Figure 6: Time required for ARIEL to repair (a) *AutoDrive1* and (b) *AutoDrive2*.

Table 2: The results of applying GP, RS-MM, and RS to *AutoDrive1* and *AutoDrive2* for 16h.

	GP (20 runs)	**RS-MM** (20 runs)	**RS** (20 runs)
AutoDrive1 (4 failures)	- 6 runs fixed 1 failure - 11 runs fixed 2 failures - 3 runs fixed 3 failures	- 8 runs fixed 3 failures - 12 runs fixed 4 failures	- 6 runs fixed 2 failures - 14 runs fixed 3 failures
AutoDrive2 (2 failures)	- 20 runs fixed 1 failure	- 15 runs fixed 1 failure - 5 runs fixed 2 failures	- 20 runs fixed 1 failure

The answer to **RQ1** is that, on average, ARIEL is able to find correct and complete patches in five hours and 11 hours for *AutoDrive1* and *AutoDrive2*, respectively.

RQ2. We ran ARIEL, GP, RS-MM and RS 20 times for 16 hours. We selected a 16 hour timeout since it was the maximum time required by ARIEL to find patches for both *AutoDrive1* and *AutoDrive2*. Also, recall that we had four failing test cases for *AutoDrive1* and two failing test cases for *AutoDrive2*. Figures 7(a) and (b), respectively, show the number of failing test cases in *AutoDrive1* and *AutoDrive2* that are left unresolved over time by different runs of ARIEL, GP, RS-MM and RS. We show the results at every two-hour interval from 0 to 16h. As shown in Figures 7(a) and (b), all 20 runs of ARIEL are able to solve all the failing test cases in *AutoDrive1* after nine hours and in *AutoDrive2* after 16 hours. Note that this is consistent with the results shown in Figure 6. Table 2 reports the number of runs of GP, RS-MM, and RS, out of 20 runs, that resolved failures in *AutoDrive1* and in *AutoDrive2*. As shown in Figures 7(a) and (b) and in Table 2, none of the runs of GP and RS were able to resolve all the failures in *AutoDrive1* or in *AutoDrive2*. RS-MM had a better performance as some of its runs could fix all the failing test cases in our case studies.

As the above results show, GP had the worst performance in repairing the integration rules. This is because GP is a population-based search algorithm that evolves a pool of candidate patches iteratively. As a result, none of the GP runs could perform more than two iterations, hence its poor performance. Comparing RS and RS-MM shows that applying a sequence of mutations instead of a single mutation at each time is important in our context and helps the repair algorithms converge more quickly into a correct patch. Finally, ARIEL has the best performance compared to the

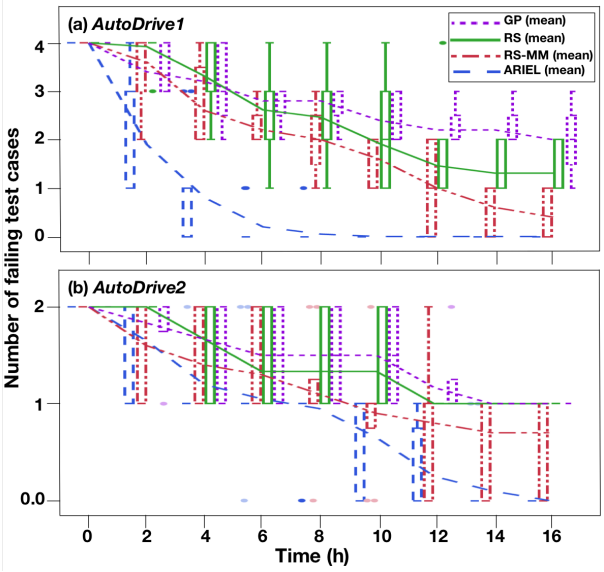

Figure 7: Comparing the number of failing test cases obtained by ARIEL, RS-MM, RS and GP when applied.

Table 3: $\hat{A}_{12}$ statistics obtained by comparing ARIEL with RS-MM, RS, and GP. ↑ indicates statistically significant results (p-value<0.05).

Time	*AutoDrive1*			*AutoDrive2*		
	vs. *RS-MM*	vs. *RS*	vs. *GP*	vs. *RS-MM*	vs. *RS*	vs. *GP*
2h	↑ 0.03 (L)	↑ 0.04 (L)	↑ 0.04 (L)	0.26 (L)	↑ 0.04 (L)	0.12 (L)
4h	↑ 0.02 (L)	↑ 0.02 (L)	↑ 0.01 (L)	0.20(L)	↑ 0.04(L)	↑ 0.08(L)
6h	↑0.00 (L)	↑ 0.01 (L)	↑0.00 (L)	0.19 (L)	0.05(L)	↑ 0.08(L)
8h	↑ 0.01 (L)	↑ 0.01 (L)	↑0.00 (L)	0.21 (L)	↑ 0.04 (L)	↑ 0.07 (L)
10h	↑0.00 (L)	↑0.00 (L)	↑0.00 (L)	0.19 (L)	0.03 (L)	↑0.04 (L)
12h	↑ 0.05 (L)	↑0.00 (L)	↑0.00 (L)	↑ 0.13 (L)	↑ 0.02 (L)	↑ 0.03 (L)
14h	↑ 0.05 (L)	↑0.00 (L)	↑0.00 (L)	↑ 0.10 (L)	↑ 0.01 (L)	↑ 0.02 (L)
16h	↑ 0.07 (L)	↑0.00 (L)	↑ 0.00 (L)	↑ 0.08 (L)	↑ 0.00 (L)	↑ 0.00(L)

three baseline methods as all its runs find the desired patch within the 16h search time budget.

We compare the results in Figure 7 using a statistical test and effect sizes. Following existing guidelines [7], we use the non-parametric pairwise Wilcoxon rank sum test [10] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [41]. Table 3 reports the results obtained when comparing the number of failed test cases with ARIEL against GP, RS-MM and RS, when they were executed over time for *AutoDrive1* and *AutoDrive2*. As shown in the table, for *AutoDrive1*, the p-values are all below 0.05 and the $\hat{A}_{12}$ statistics show once again large effect sizes. For *AutoDrive2*, the p-values related to the results produced when the search time ranges between 12h and 16h are all below 0.05 and the $\hat{A}_{12}$ statistics show large effect sizes. Hence, the number of failing test cases obtained by ARIEL when applied to *AutoDrive1* and *AutoDrive2* is significantly lower (with a large effect size) than those obtained by RS-MM, RS and GP.

The answer to **RQ2** is that ARIEL significantly outperforms the baseline techniques for repairing integration faults when applied to our two case study systems.

Feedback from domain experts. We conclude our evaluation by reporting the feedback we received from engineers regarding the practical usefulness and benefits of our automated repair technique. In particular, we investigated whether the patched integration rules generated by our approach were understandable by engineers, and whether engineers could come up with such patches on their own using their domain knowledge and reasoning, all this without relying on our technique. The feedback we report here is based on the comments the lead engineers at company A (our partner company) made during two separate meetings. The engineers who participated in these meetings were involved in the development of *AutoDrive1* and *AutoDrive2* and were fully knowledgeable about the details of these two systems and the simulation platform used to test these systems. During the meetings, we discussed the four failing test cases of *AutoDrive1*, and the two failing test cases of *AutoDrive2* that we had received from company A. Recall that as discussed in Section 4.3, these test cases were failing due to errors in the integration rules of *AutoDrive1* and *AutoDrive2*. For each test case, *we asked engineers whether they could think of ways to resolve the failure* by suggesting modifications to the integration rules. Then, we presented to the engineers the patches generated by ARIEL. Note that we have alternative patches for each failing test case since we applied ARIEL several times to our case studies. We randomly selected two patches for each failure to be used in our discussions. We then asked the engineers *whether they understood the automatically generated patches*. We further discussed whether or not the automatically generated patches are the same as the fixes suggested by the engineers earlier in the meetings.

When at the beginning of the meetings, we asked engineers to propose patches based on their domain knowledge and reasoning and, in almost all cases, they proposed additional integration rules. Specifically, after reviewing the simulation of each failing test case, the engineers identified the conditions that had led to that specific failure. Then, they synthesized a new rule such that upon satisfaction of the conditions leading to the failure, the rule activates a feature that could avoid that failure. In contrast, ARIEL repairs integration faults either by modifying operators and thresholds in the clauses of the rules preconditions or by re-ordering the integration rules (see Section 3.2.2), and it never adds new rules.

In summary, the feedback we received from the engineers showed that they found the patches generated by ARIEL understandable and optimal. They agreed that adding new rules must be avoided if the integration faults can be resolved by modifying the existing rules since it is difficult to maintain a large number of rules. Further, as they add new rules they may introduce dead code or new faults in the system. The engineers further admitted that it is impossible to manually come up with resolutions that ARIEL can generate automatically since one cannot analyze the impact of varying thresholds, logical operators or rule reordering without automated assistance. Based on the feedback we received from the engineers, they concurred that the resolutions generated by ARIEL are valid, understandable, useful and optimal and they cannot be

produced by engineers. We note that ARIEL is not able to handle integration failures when the failure is due to a missing integration rule. In such cases, engineers can review the unaddressed failing test cases and try to manually synthesize rules that can handle such failures.

5 THREATS TO VALIDITY

The main threat to external validity is that our results may not generalize to other contexts. We distinguish two dimensions for external validity: (1) the applicability of our approach beyond our case study system, and (2) obtaining the same level of benefits as observed in our case study. As for the first dimension, we note that, in this paper, we provided two industrial ADS case studies. Our approach is dedicated to cases when there is an integration component deciding, based on rules, what commands are sent to actuators among alternative commands coming from different features, at every time step. In our context, as it is commonly done for cyber-physical systems, testing is done through simulation of the environment and hardware. The mutation operators used in our approach are general for integration rules in automated driving systems (the target of the paper) and for any cyber-physical systems where features send commands to common actuators and conflicts are prevented by a rule-based integration component (common in self-driving cars, robots, etc.). Our framework can be further extended with other operators if needed in different contexts. With respect to the second dimension, while our case study was performed in a representative industrial setting, additional case studies remain necessary to further validate our approach. In summary, our framework is generalizable to other domains with expensive test suites and similar integration architecture. If needed, mutation operators can be easily extended given the general structure of our framework.

6 RELATED WORK

We classified the related work (Table 4) by analyzing whether the existing automated repair approaches can handle multi-location faults (**C1**)? whether they can handle repairing systems that are expensive to execute (**C2**)? and whether they are able to distinguish between faults with different severity levels (**C3**)? As shown in the table, none of the existing repair techniques can address these three criteria, while as discussed earlier in the paper, ARIEL is designed under the assumption that test cases are expensive to run. Further, ARIEL can simultaneously repair multiple faults that might be multi-location and can prioritise repairing more severe faults over less severe ones. Below we discuss the related work included in Table 4 in more detail.

Many existing automated repair techniques rely on Genetic Programming (e.g., Arcuri et. al. [6, 8], GenProg [28], Marriagent [25], pyEDB[3], Par [22], and ARC [21]) and are similar to the GP baseline we used in Section 4 to compare with ARIEL. As shown in our evaluation, GP could not repair faults in our computationally expensive case studies (hence, failing **C2**). Among the techniques based on Genetic Programming, only ARC [21] is able to handle multi-location faults (**C1**) since it uses a set of pre-defined templates to generate patches. However, it does not address **C2** nor **C3**.

Table 4: Classification of the related work based on the following criteria: Can the approach handle multi-location faults (C1)? Can the approach repair computationally expensive systems (C2)? Does the approach prioritise fixing more critical faults over the less critical ones (C3)?

Ref	C1	C2	C3
[3, 22, 25, 28]	−	−	−
[21]	+	−	−
[5, 11, 12, 19, 35, 37, 38, 43]	−	+	−
[30, 31, 34, 42]	+	+	−

Several techniques use random search (e.g., RSRepair [37] and SCRepair [19]) to automate program repair. Although RSRepair [37] indicates that random search performs better than GenProg in terms of the number of iterations required for repair, a recent study [24] showed that GenProg performs better than RSRepair when applied to subjects different from those included in the original dataset of GenProg. We included two baselines based on random search in our evaluation (RS and RS-MM in Section 4). As shown there, while random search is more efficient than GP in repairing computationally expensive systems, it still underperforms ARIEL since it does not maintain an archive of partially repaired solutions. Further, as Table 4 shows, repair approaches based on random search do not address **C1** nor **C3**.

Similar to ARIEL, SPR [30], Prophet [31], AutoFix-E [42], and Angelix [34] can address both **C1** and **C2**. To handle multi-location faults, SPR [30], Prophet [31] and AutoFix-E [42] use pre-defined templates that are general repair solutions and are typically developed based on historical data or for the most recurring faults. In our context and at early stages of function modelling for ADS, we often do not have access to archival data and cannot develop such generic templates. Further, faults not conforming to some pre-defined templates may not be repaired if we only rely on template-induced patches. Angelix [34] uses constraint solving to synthesise multi-line fixes. Exhaustive and symbolic constraint solvers, however, face scalability issues and often are inapplicable to black-box simulation systems such as ADS simulators [4]. In contrast to these techniques, and as demonstrated by our evaluation, ARIEL succeeds in addressing **C1** to **C3** for complex simulation-based systems in the context of automated driving systems.

7 CONCLUSION

We proposed a repair technique to automatically resolve integration faults in automated driving systems (ADS). Our approach localizes faults over several lines of code to fix complex integration faults and deals with the scalability issues of testing and repairing ADS. Our repair algorithm relies on a many-objective, single-state search algorithm that uses an archive to keep track of partial repairs. Our approach is evaluated using two industrial ADS. The results indicate that our repair strategy can fix the integration faults in these two systems and outperforms existing automated repair techniques. Feedback from domain experts indicates that the patches generated by our approach are understandable by engineers and could not have been developed by them without any automation assistance.

ACKNOWLEDGMENTS

This project has received funding from IEE S.A.,Luxembourg, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277), and NSERC of Canada under the Discovery and CRC programs.

REFERENCES

[1] 2019. Matlab/Simulink. https://nl.mathworks.com/products/simulink.html.
[2] 2020. Appendix. https://bitbucket.org/anonymous83/faultrepair/src/master/. Also submitted along with the paper.
[3] Thomas Ackling, Bradley Alexander, and Ian Grunert. 2011. Evolving Patches for Software Repair. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO'11)* (Dublin, Ireland). ACM, New York, NY, USA, 1427–1434.
[4] Rajeev Alur. 2015. *Principles of Cyber-Physical Systems.* MIT Press.
[5] Andrea Arcuri. 2008. On the Automation of Fixing Software Bugs. In *Companion of the International Conference on Software Engineering (ICSE Companion'08)* (Leipzig, Germany). ACM, New York, NY, USA, 1003–1006.
[6] Andrea Arcuri. 2011. Evolutionary Repair of Faulty Software. *Applied Software Computing* 11, 4 (June 2011), 3494–3514.
[7] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.
[8] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence (WCCI'08))*. IEEE, Hong Kong, 162–168.
[9] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing Autonomous Cars for Feature Interaction Failures Using Many-objective Search. In *Proceedings of the International Conference on Automated Software Engineering (ASE'18)*. ACM, Montpellier, France, 143–154.
[10] J. Anthony Capon. 1991. *Elementary Statistics for the Social Sciences: Study Guide.* Wadsworth Publishing Company, Belmont, CA, USA.
[11] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating Fixes from Object Behavior Anomalies. In *Proceedings of the International Conference on Automated Software Engineering (ASE'09)*. IEEE, San Diego, CA, USA, 550–554.
[12] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic Repair of Buggy if Conditions and Missing Preconditions with SMT. In *Proceedings of the International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA'14)* (Hyderabad, India). ACM, New York, NY, USA, 30–39.
[13] Eric Foxlin. 2005. Pedestrian tracking with shoe-mounted inertial sensors. *IEEE Computer graphics and applications* 25, 6 (2005), 38–46.
[14] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.
[15] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.
[16] John H. Holland. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA.
[17] Martin Jähne, Xiaodong Li, and Jürgen Branke. 2009. Evolutionary algorithms and multi-objectivization for the travelling salesman problem. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO'09)*. ACM, Montréal, Canada, 595–602.
[18] Thomas Jansen. 2002. On the analysis of dynamic restart strategies for evolutionary algorithms. In *International Conference on Parallel Problem Solving from Nature (PPSN'02)*, Vol. 2. Springer, Granada, Spain, 33–43.
[19] Tao Ji, Liqian Chen, Xiaoguang Mao, and Xin Yi. 2016. Automated Program Repair by Using Similar Code Containing Fix Ingredients. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC'16)*, Vol. 1. IEEE, Atlanta, GA, USA, 197–202.
[20] Wei Jin and Alessandro Orso. 2013. F3: fault localization for field failures. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA:13)*. ACM, Lugano, Switzerland, 213–223.
[21] David Kelk, Kevin Jalbert, and Jeremy S. Bradbury. 2013. Automatically Repairing Concurrency Bugs with ARC. In *Proceedings of the International Conference on Multicore Software Engineering, Performance, and Tools (MSEPT'13)*, João M. Lourenço and Eitan Farchi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–84.
[22] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. IEEE, San Francisco, CA, USA, 802–811.
[23] Joshua D Knowles, Richard A Watson, and David W Corne. 2001. Reducing local optima in single-objective problems by multi-objectivization. In *Proceedings of the International Conference on Evolutionary Multi-Criterion Optimization (EMO'01)*. Springer, Zurich, Switzerland, 269–283.
[24] Xianglong Kong, Lingming Zhang, W Eric Wong, and Bixin Li. 2015. Experience report: How do techniques, programs, and tests impact automated program repair?. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'15)*. IEEE, Washington DC, USA, 194–204.
[25] Ryotaro Kou, Yoshiki Higo, and Shinji Kusumoto. 2016. A Capable Crossover Technique on Automatic Program Repair. In *Proceedings of the International Workshop on Empirical Software Engineering in Practice (IWESEP'16)*. IEEE, Osaka, Japan, 45–50.
[26] John R Koza and John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection.* Vol. 1. MIT press, Cambridge, MA, USA.
[27] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the International Conference on Software Engineering (ICSE '12)* (Zurich, Switzerland). IEEE Press, Piscataway, NJ, USA, 3–13.
[28] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering (TSE)* 38, 1 (Jan 2012), 54–72.
[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436.
[30] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)* (Bergamo, Italy). ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811
[31] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)* (St. Petersburg, FL, USA). ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617
[32] Sean Luke. 2013. *Essentials of Metaheuristics* (second ed.). Lulu, Fairfax, Virginie, USA.
[33] Ankith Manjunath, Ying Liu, Bernardo Henriques, and Armin Engstle. 2018. Radar Based Object Detection and Tracking for Autonomous Driving. In *Proceedings of the MTT-S International Conference on Microwaves for Intelligent Mobility (ICMIM'18)*. IEEE, Munich, Germany, 1–4.
[34] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE'16)* (Austin, Texas). ACM, New York, NY, USA, 691–701.
[35] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE '13)* (San Francisco, CA, USA). IEEE Press, Piscataway, NJ, USA, 772–781.
[36] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the International Conference on Software Engineering (ICSE '17)* (Buenos Aires, Argentina). IEEE Press, Piscataway, NJ, USA, 609–620. https://doi.org/10.1109/ICSE.2017.62
[37] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*. ACM, New York, USA, 254–265.
[38] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'15)* (Baltimore, MD, USA). ACM, New York, NY, USA, 24–36.
[39] TASS-International. 2019. PreScan. https://www.tassinternational.com/prescan.
[40] Richard van der Horst and Jeroen Hogema. 1993. Time-to-collision and collision avoidance systems. In *Proceedings of the workshop of the International Cooperation on Theories and Concepts in Traffic Safety (ICTCT'93)*. -, Salzburg, Austria, 109–121.
[41] András Vargha and Harold D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.
[42] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated Fixing of Programs with Contracts. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'10)* (Trento, Italy). ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/1831708.1831716
[43] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the International Conference on Automated Software Engineering (ASE'13)*. IEEE, Silicon Valley, CA, USA, 356–366.
[44] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering (ICSE'09)*. IEEE, Vancouver, Canada, 364–374.

[45] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 4, Article 31 (Oct. 2013), 40 pages. https://doi.org/10.1145/2522920.2522924

[46] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. 2017. Human Competitiveness of Genetic Programming in Spectrum-Based Fault Localisation: Theoretical and Empirical Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 1, Article 4 (June 2017), 30 pages. https://doi.org/10.1145/3078840