Testing Autonomous Cars for Feature Interaction Failures using Many-Objective Search

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

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

Shiva Nejati
University of Luxembourg
Luxembourg
shiva.nejati@uni.lu

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

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

ABSTRACT

Complex systems such as autonomous cars are typically built as a composition of features that are independent units of functionality. Features tend to interact and impact one another's behavior in unknown ways. A challenge is to detect and manage feature interactions, in particular, those that violate system requirements, hence leading to failures. In this paper, we propose a technique to detect feature interaction failures by casting this problem into a search-based test generation problem. We define a set of hybrid test objectives (distance functions) that combine traditional coverage-based heuristics with new heuristics specifically aimed at revealing feature interaction failures. We develop a new search-based test generation algorithm, called FITEST, that is guided by our hybrid test objectives. FITEST extends recently proposed many-objective evolutionary algorithms to reduce the time required to compute fitness values. We evaluate our approach using two versions of an industrial self-driving system. Our results show that our hybrid test objectives are able to identify more than twice as many feature interaction failures as two baseline test objectives used in the software testing literature (i.e., coverage-based and failure-based test objectives). Further, the feedback from domain experts indicates that the detected feature interaction failures represent real faults in their systems that were not previously identified based on analysis of the system features and their requirements.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Search-based software engineering**;

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238192

KEYWORDS

Search-based Software Testing, Many-Objective Optimization, Automotive Systems, Feature Interaction Problem

ACM Reference Format:
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 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238192

1 INTRODUCTION

Feature-based development aims to build complex systems consisting of units of functionality known as *features*. Individual features are typically traceable to specific system requirements and are mostly independent and separate from one another [37, 44, 63]. By closely mirroring requirements, features make it easier for engineers to develop complex systems iteratively and incrementally. Self-driving cars, and in general automotive systems, are among well-known examples of feature-based systems [11, 24, 76]. A self-driving system, for example, may include the following features, each automating an independent driving function: An automated emergency braking (AEB), an adaptive cruise control (ACC) and a traffic sign recognition (TSR).

Although features are typically designed to be independent, they may behave differently when composed with other features. A *feature interaction* is a situation where one feature impacts the behavior of another feature [17, 25, 44]. For example, in a self-driving system, feature interactions are likely to arise when several features control the same actuators. More specifically, in a self-driving system, both ACC and AEB control the braking actuator. A feature interaction may arise when a braking command issued by AEB to immediately stop the car is overridden by ACC commanding the car to maintain the same speed as that of the front car. Some feature interactions are desirable, and some may result in violations of system safety requirements and are therefore undesired. For example, the above feature interaction between AEB and ACC may lead to an accident, and hence, is undesirable.

The feature interaction problem has been extensively studied in

the literature [7, 17, 25, 44]. Some techniques focus on identifying feature interactions at the requirements-level by analysis of formal or semi-formal requirements models [15, 18, 75]. Several techniques detect feature interaction errors in implementations using test cases derived from feature models capturing features and their dependencies [8, 35, 58, 61]. Other approaches devise design and architectural resolution strategies to eliminate at runtime undesired feature interactions identified at the requirements-level [41, 44, 70, 76]. For self-driving systems, however, feature interactions should be identified as early as possible and before the implementation stage since late resolution of undesired interactions can be too expensive and may involve changing hardware components. Further, feature interactions in self-driving systems are numerous, complex and depend on several factors such as the characteristics of sensors and actuators, car and pedestrian dynamics, weather condition, road traffic and sidewalk objects. Without effective and automated assistance, engineers cannot detect undesired feature interactions within the space of all possible interactions and cannot assess the impact of complex environmental factors on feature interactions.

In this paper, we develop an automated approach to detect undesired feature interactions in self-driving systems at an early stage. Our approach identifies undesired feature interactions based on executable function models of self-driving systems embedded into a realistic simulator capturing the self-driving system hardware and environment. Building function models at an early stage is standard practice in model-based development of control systems and is commonly followed by the automotive and aerospace industry [57, 72, 74]. Function modeling takes place after identification of system requirements and prior to software design and architecture activities. Function models of control systems capture algorithmic behaviors of software components and physic dynamics of hardware components. Similar to the automotive and aerospace industry, the function models and the simulator of the self-driving system used in this paper are specified in the Matlab/Simulink language [1].

In this paper, we cast the problem of detecting undesired feature interactions into a *search-based testing problem*. Specifically, we aim to generate test inputs that expose undesired feature interactions when applied to executable function models of self-driving systems. Search-based techniques have been successfully applied to simulation-based testing of control systems and self-driving features [3, 13, 14, 23, 53, 54] as well as various other testing problems such as unit testing [38, 55, 69], regression testing [49, 73] and optimizing machine learning components [66].

Contributions. Our contributions are as follows:

First, we define novel hybrid *test objectives* that determine how far candidate tests are from detecting undesired interactions. Our test objectives combine three different heuristics: (i) A *branch coverage heuristic* [55] ensuring that the generated test cases exercise all branches of the component(s) integrating features. (ii) A *failure-based heuristic* based on system safety requirements ensuring that test cases stress the system into breaking its safety requirements. (iii) An *unsafe overriding heuristic* that aims to exhibit system behaviors where some feature output is overridden by other features such that some system safety requirements may be violated.

Second, we introduce FITEST (Feature Interaction TESTing), a new *many-objective test generation algorithm* to detect undesired feature interactions. We opt for a many-objective optimization algorithm since test generation in our context is driven by many competing test objectives resulting from the combination of heuristics above. Specifically, FITEST builds on the recently proposed many-objective genetic algorithms [59, 60] that effectively generate test cases satisfying a large number of test objectives. In our work, computing test objectives is expensive. Hence, at each iteration, FITEST dynamically selects the minimum number of test cases closest to satisfying test objectives, thus reducing the total number of fitness computations.

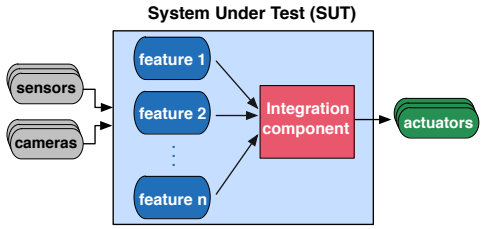

Figure 1: Overview of a typical function model capturing the software subsystem (SUT) of a self-driving car.

Third, we evaluate FITEST using two industrial self-driving systems from our partner company IEE [43]. Both systems represent a (partial) self-driving car consisting of four features. The engineers at IEE had developed alternative strategies to resolve the known feature interactions in these two systems. FITEST, however, was able to identify, on average, 5.9 and 7.2 undesired feature interactions in the two systems, respectively. The engineers confirmed that the detected interactions represent real faults that were not a priori known to them[1]. Further, we compared our hybrid test objectives used by FITEST with two baseline test objectives from the software testing literature (namely, coverage-based [38, 55] and failure-based test objectives [4, 13, 20, 23]). Our results show that our hybrid test objectives are able to identify more than twice as many feature interaction failures as the coverage-based and failure-based test objectives.

Structure. Section 2 motivates our work. Section 3 presents our approach. Section 4 describes our evaluation. Section 5 compares with related work. Section 6 concludes the paper.

2 MOTIVATION

Figure 1 shows an overview of a typical function model capturing the software subsystem of a self-driving car. The system under test (SUT) consists of a set of self-driving features and a component capturing the decision algorithm combining feature outputs. SUT receives its inputs from sensors/cameras and sends its outputs to actuators. Both inputs and outputs are sequences of timestamped values. The entire SUT runs iteratively at regular *time steps*. At every time step, the features receive sensor/camera values issued in that step, and output values are computed and sent to actuators by the end of the step. Each feature controls one or more actuators. Actuators may receive commands from more than one feature at the same time step, and sometimes these commands are conflicting. The integration component has to generate final outputs to actuators after resolving conflicting feature outputs.

[1]The material we used to get the industry feedback is available online [2].

As discussed in Section 1, our goal is to identify feature interactions at the requirements-level and in terms of system functional behavior. Hence, we base our analysis on function models specifying algorithmic and control behaviors. Feature interaction failures due to software architecture and design issues are not studied in this paper.

We use a case study system, called *SafeDrive*, from our partner company IEE. *SafeDrive* contains the following four self-driving 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 a car ahead (or a *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 is the same as PP but it prevents accidents with objects other than pedestrians. Once the risk of an accident is over and the road is clear, both PP and AEB issue acceleration commands to bring back the car to the same speed that the car had before their intervention. All the features generate braking and acceleration commands to respectively control the brake and the throttle actuators. TSR and ACC, additionally, generate steering commands.

The *SafeDrive* features may issue conflicting commands to the same actuators. For example, *Scenario-1:* ACC orders the car to accelerate, while a pedestrian starts crossing the road. Hence, at the same time, PP starts sending braking commands to avoid hitting the pedestrian. *Scenario-2:* The car reaches an intersection while the traffic light turning from orange to red. ACC orders the car to accelerate since the leading car has also accelerated to pass the intersection while the light is orange. At the same time, TSR orders to brake since it detects that a red light is about to come.

When feature interactions are known, engineers can develop the decision logic of the integration component (see Figure 1) such that the interactions do not lead to failures (e.g., using existing feature interaction resolution techniques [44, 76]). For example, for *Scenario-1*, engineers may decide to prioritize the braking command of PP over the acceleration command of ACC to avoid hitting a pedestrian. The resolution strategy for *Scenario-2* can be prioritizing TSR if the car can safely stop by the traffic light, and otherwise, prioritizing ACC. However, feature interactions in *SafeDrive* are numerous and many of them may not be known, particularly at early development stages. Further, the feature interaction resolution strategies cannot always be determined statically and may depend on complex environment factors. For example, deciding "if the car can safely stop" in the resolution strategy for *Scenario-2* depends on the speed and the position of the car, the distance to the car behind, road topology and the weather condition. Therefore, we need techniques that, at early development stages, (1) detect undesired feature interactions in *SafeDrive*, and (2) test whether the proposed resolution strategies can avoid failures under different environment conditions.

In the next sections, we present and evaluate a technique that tests the functional behavior of autonomous cars to detect their undesired feature interactions. Our technique accounts for the impact of the environment factors on the self-driving system behavior. It, further, ensures that feature interaction resolution strategies

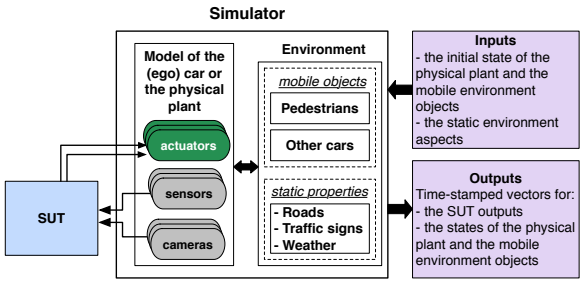

Figure 2: Early testing of control system function models using simulators.

devised by engineers satisfy system safety requirements under different environment conditions. We note that in Section 3.3, we will provide a precise formalization of the context upon which we build. The formalism is generic and based on simple assumptions that can be accommodated by many feature-based systems. Hence, in addition to autonomous cars, our work applies to any feature-based system expressible using our formalism.

3 APPROACH

In this section, we present our feature interaction detection technique. As discussed earlier, our technique generates test inputs for function models of self-driving systems, exposing their undesired feature interactions. Section 3.1 describes how we integrate the function models into a high-fidelity, physics-based simulator for self-driving systems. Section 3.2 characterizes the test inputs and outputs for self-driving systems. Section 3.3 introduces our hybrid test objectives. Section 3.4 presents FITEST, our proposed many-objective test generation algorithm that utilizes our test objectives to generate test inputs revealing feature interaction failures.

3.1 Testing Feature-Based Control Systems

Testing Cyber-Physical Systems (CPSs) at early stages is generally performed using simulators. To test the function model in Figure 1, we connect the SUT model to a simulator such that it receives inputs from the sensor and camera models of the simulator and sends its outputs to the actuator models of the simulator (see Figure 2). The sensor, camera and actuator models are within a physical model of a car (or a physical plant according to general CPS terminology) in the simulator. To run the simulator, we specify the initial state of the simulator physical plant and mobile environment objects as well as the static environment properties (e.g., weather condition and road shapes for self-driving systems). The simulator can execute the SUT in a feedback loop with the plant and the environment. For *SafeDrive*, we use PreScan, a physics-based simulator for self-driving systems [67]. PreScan relies on dynamic Simulink models to compute movements of cars and pedestrians and is able to capture the environment static properties such as the weather condition and the road topology. Some examples of *SafeDrive* simulations are available online [2].

3.2 Test Inputs and Outputs

The test inputs for a self-driving system are the inputs required to execute the simulation framework in Figure 2. For example, to

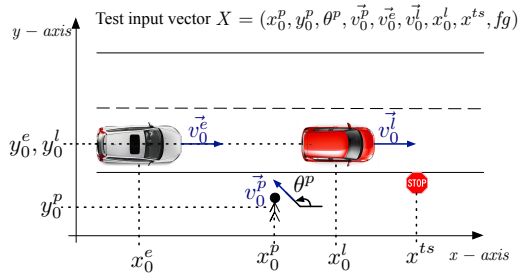

Figure 3: Test inputs required to simulate *SafeDrive*, our case study system.

test *SafeDrive*, we start by instantiating the simulation framework so that the simulator is able to exercise the behaviors of the PP, AEB, TSR and ACC features. Our simulation framework contains the following objects: (1) An *ego car* equipped with *SafeDrive*, (2) a *leading car* to test both the ACC and the AEB features of the ego car, and (3) a pedestrian that crosses the road starting from an initial position on the sidewalk and is used to exercise PP. The simulation environment, further, includes one traffic sign to test the TSR feature. We only consider a stop sign or a speed limit sign for our case study. This setup is meant to reduce the complexity of simulations and was suggested by the domain experts.

The test inputs of *SafeDrive* are shown in Figure 3. They include the following variables: (1) The initial position x_0^e, y_0^e and the initial speed v_0^e of the ego car. (2) The initial position x_0^l, y_0^l and the initial speed v_0^l of the leading car. (3) The initial position x_0^p, y_0^p, the initial speed v_0^p and the orientation θ^p of the pedestrian. (4) The position x^{ts} of the traffic sign that varies along the x-axis, but is fixed along the y-axis. (5) The fog degree fg. In our simulator, among different weather-related properties (e.g., snow and rain), the fog level has the largest impact on the object detection capabilities of *SafeDrive*. Hence, we include the fog level in the test inputs.

All the above variables except for fg are float numbers varying within ranges specified by domain experts. The variable fg is an enumeration specifying ten different degrees of fog. In addition to the domain value ranges, there are often some constraints over test inputs to ensure that simulations start from a valid and meaningful state. Specifically, we have the following two constraints for *SafeDrive*: (i) The ego car starts behind the leading car with a safety distance gap, denoted sd, and with a speed close to the speed of the leading car. This constraint is specified as follows: $sd - \epsilon \leq x_0^l - x_0^e \leq sd + \epsilon$ and $|v_0^e - v_0^l| \leq \epsilon'$ where ϵ and ϵ' are two small constants, and sd, which is the safety distance gap between the ego and the leading cars, is determined based on the car speeds. (ii) The traffic sign is located within a sufficiently long distance from the ego car to give enough time to the TSR feature to react (i.e., $|x^{ts} - x_0^e| < c$ where c is constant value). Finally, to simulate the system, we need to specify the duration of the simulation T and the simulation step size δ.

As shown in Figure 2, the simulator outputs are time-stamped vectors specifying (1) SUT outputs, (2) states of the physical plants and (3) states of any mobile environment object. All these outputs are vectors with $\frac{T}{\delta}$ elements where the element at position i specifies the output at time $i \cdot \delta$. For example, Figure 4 illustrates the SUT outputs generated by simulating *SafeDrive*. Specifically, the

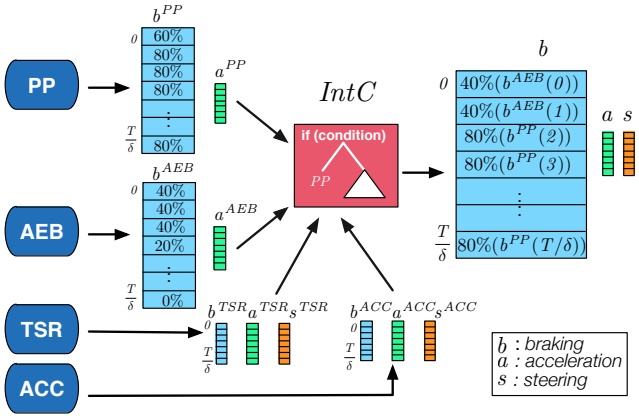

Figure 4: Actuator command vectors generated at the feature-level and at the system-level by simulating *SafeDrive*. Vectors b^f, a^f and s^f indicate command vectors generated by feature f for the braking, acceleration and steering actuators, respectively. The *IntC* component analyzes the command vectors generated by all the features and issues the final command vectors b, a and s to the braking, acceleration and steering actuators, respectively.

SUT outputs in that figure include both the outputs of each feature inside the SUT and the output of the integration component, i.e., the final command vector sent to the actuators.

3.3 Hybrid Test Objectives

Our test objectives aim to guide the test generation process towards test inputs that reveal undesired feature interactions. We first present our formal notation and assumptions and then we introduce our test objectives. Note that since in this paper we are primarily interested in the feature interaction problem, we design our test objectives such that they focus on detecting failures that arise due to feature interactions, but not failures that arise due to an individual feature being faulty.

Notation. We define a feature-based control system $\mathcal{F}$ as a tuple $(f_1, \ldots, f_n, IntC)$ where $f_1, \ldots, f_n$ are features and *IntC* is an integration component. The system $\mathcal{F}$ controls a set *Act* of actuators. Each feature f_i controls a set $Act^{f_i} \subseteq Act$ of actuators. Since we are interested in identifying feature interaction failures and not failures due to errors inside individual features, our approach does not require any visibility into the internals of features. But, in our work, *IntC* is a white-box component. The *IntC* behavior is typically conditional where each condition checks a specific feature interaction situation and resolves potential conflicts that may arise under that condition. We assume $\mathcal{F}$ has a set of safety requirements such that each requirement is related to one feature which is responsible for the satisfaction of that requirement. For example, the second column of Table 1 shows the safety requirements for SafeDrive. The feature responsible for satisfying each requirement is shown in the first column.

As discussed earlier, testing $\mathcal{F}$ is performed by connecting $\mathcal{F}$ to a simulation framework (see Figure 2). A test case for $\mathcal{F}$ is a vector

Table 1: Safety requirements and failure distance functions for *SafeDrive*.

Feature	Requirement	Failure distance functions ($FD_1, \ldots, FD_5$)
PP	No collision with pedestrians	$FD_1(i)$ is the distance between the ego car and the pedestrian at step i.
AEB	No collision with cars	$FD_2(i)$ is the distance between the ego car and the leading car at step i.
TSR	Stop at a stop sign	Let $u(i)$ be the speed of the ego car at time step i if a stop sign is detected, and let $u(i) = 0$ if there is no stop sign. We define $FD_3(i) = 0$ if $u(i) \geq 5km/h$; $FD_3(i) = \frac{1}{u(i)}$ if $u(i) \neq 0$; and otherwise, $FD_3(i) = 1$.
TSR	Respect the speed limit	Let $u'(i)$ be the difference between the speed of the ego car and the speed limit at step i if a speed-limit sign is detected, and let $u'(i) = 0$ if there is no speed-limit sign. We define $FD_4(i) = 0$ if $u'(i) \geq 10km/h$; $FD_4(i) = \frac{1}{u'(i)}$ if $u'(i) \neq 0$; and otherwise, $FD_4(i) = 1$.
ACC	Respect the safety distance	$FD_5(i)$ is the absolute difference between the safety distance sd and $FD_2(i)$.

X of inputs required to execute the simulation framework into which $\mathcal{F}$ is embedded (e.g., Figure 3 shows the test input vector for *SafeDrive*). The test output of $\mathcal{F}$ includes: (1) a vector v^f_{act} generated by every feature f and for every actuator $act \in Act^f$; (2) a vector v_{act} generated by *IntC* for each actuator $act \in Act$; and (3) a trajectory vector for the physical plant and every mobile environment object.

Test objectives. A key aspect in search-based software testing [40, 55] is the notion of distance functions $D(.)$ that measure how far a candidate test X is from reaching testing targets (e.g., covering branches in white-box testing). Our testing targets aim to reveal undesired feature interactions. An undesired feature interaction is revealed when: (1) Some safety requirement r is violated such that (2) the integration component (i.e., *IntC*) *overrides* the output of the feature responsible for r. We note that if r is violated while *IntC* selects the output of the feature responsible for r, then the violation is likely to be due to the internals of that feature and not due to feature interactions. Therefore, we define two distance functions, namely *failure distance* and *unsafe overriding distance* to respectively capture the conditions (1) and (2) above. Further, we ensure that the generated tests exercise all branches of *IntC*. Hence, our third distance corresponds to the well-known *distance* used in coverage-based testing [55]. In the following, we present each distance separately and then we describe how we combine them to build our test objectives.

Coverage distance. First, the generated test cases have to exercise every branch of *IntC*. Given that *IntC* is white-box, we rely on two widely-used heuristics in branch coverage, namely the *approach level* [55] and the normalized *branch distance* [38, 55]. Each branch b_i in *IntC* has its own distance function BD_i to minimize which is defined according to the two heuristics above. The distance BD_i is equal to zero iff a candidate test case tc covers the associated branch b_i.

Failure distance: The failure distance evaluates how close the system $\mathcal{F}$ is from violating its safety requirements at each simulation time step. For each system safety requirement $j \in \{1, \ldots, m\}$, we define a failure distance FD_j such that $FD_j(i) = 0$ iff requirement j is violated at time step i. FD_j is a black-box heuristic, i.e., it relies on system outputs only.

For example, the third column of Table 1 describes functions $FD_1(i)$ to $FD_5(i)$ for the five safety requirements of *SafeDrive* in the second column of that table. Since self-driving safety requirements typically concern mobile environment objects and physical plants, the failure distance is computed based on the trajectories of the physical plant and the environment mobile objects generated by simulation. Recall that for each safety requirement of $\mathcal{F}$, there is only one feature that is responsible for its satisfaction. Hence, each FD_j is related to a feature f of $\mathcal{F}$ such that f is the feature responsible for satisfying j. When any of the $FD_1(i)$ to $FD_5(i)$ functions in Table 1 yields a zero value at step i, it means that a requirement failure corresponding to that function is detected. Further, small or large values of these functions indicate that the system is, respectively, close to or far from exhibiting a failure. For example, function $FD_1(i)$ related to *PP* measures the distance between the ego car and the pedestrian. A search algorithm guided by FD_1 generates simulations during which the distance between the ego car and the pedestrian is minimized, hence increasing the likelihood of an accident. As another example, the distance functions related to the TSR requirements are defined as the inverse of the speed of the ego car for the stop sign, and the inverse of the difference between the speed of the ego car and the speed limit for the speed limit sign. According to domain experts, the stop sign requirement is certainly violated when the speed of the car never falls below $5km/h$ after detecting the stop sign, and the speed limit sign requirement is certainly violated when the speed of the car exceeds the speed limit by more than $10km/h$. For both cases we set the concerned failure function to zero indicating that a safety violation has occurred.

Unsafe overriding distance: This distance function aims to prioritize behaviors that violate safety requirements due to errors inside *IntC* over the behaviors that fail due to errors inside features. At each simulation time step, the *IntC* component *prioritizes* the output of some feature and *overrides* those of the rest. Recall that for each actuator act, *IntC* always generates the v_{act} vector, and every feature f generates v^f_{act} iff f controls act (i.e., $act \in Act^f$). If $v_{act}(i) = v^f_{act}(i)$, it means at time step i, *IntC* prioritizes f over other features controlling act. Dually, if $v_{act}(i) \neq v^f_{act}(i)$, it means at time step i, *IntC overrides* the command issued by f for act. For example, in Figure 4, the *IntC* component of SafeDrive prioritizes *AEB* over the other three features to control the braking actuator at time steps 0 and 1.

For an actuator act and at time step i, we say *IntC unsafely overrides* f if the command at $v_{act}(i)$ is *less safe* than the command at $v^f_{act}(i)$ for act. We say a command c is less safe than a command c' for an actuator act, when act executing c is more likely to break some requirement compared to act executing c'. For example, in the SafeDrive system, a mild and late braking more likely leads to violating one of the requirements in Table 1 compared to a firm and early braking. Dually, the requirements in Table 1 are more likely to fail when we accelerate faster than when we accelerate more slowly.

Note that test cases that violate safety requirements without *IntC* unsafely overriding any feature do not fail due to faults in *IntC*. This

is because, for such test cases, either *IntC* does not override any decision of any individual feature or its decision to override a feature does not increase the likelihood of violating a safety requirement. Hence, such test cases fail due to a fault in a feature. For *IntC* to be faulty, it is necessary that v_{act} unsafely overrides v^f_{act} in some simulation time step. For each feature f, we define an *unsafe overriding distance* UOD_f such that $UOD_f = 0$ iff *IntC* unsafely overrides the output of f at least once during the simulation, and otherwise, $UOD_f > 0$. Such a distance guides the search towards generating tests that cause *IntC* to unsafely override f.

To compute UOD_f, we define UOD^{act}_f for each actuator *act* controlled by f. For actuators where higher force values are safer (e.g., braking), *IntC* unsafely overrides f when $v^f_{act}(i) > v_{act}(i)$ (i.e., when, at step i, f orders to brake more strongly than *IntC*). We use the traditional branch distance for the *greater-than* condition [47] to translate this condition into a distance function. That is, for such actuators, we define UOD^{act}_f at each simulation step i, as follows:

$$UOD^{act}_f(i) = \begin{cases} v_{act}(i) - v^f_{act}(i), & \text{if } v^f_{act}(i) < v_{act}(i) \\ 0, & \text{otherwise} \end{cases}$$

Dually, for actuators that lower force values are safer (e.g., acceleration), *IntC* unsafely overrides f when $v_{act}(i) > v^f_{act}(i)$ (i.e., when the accelerating command of f is less than that of *IntC* at step i). Following the traditional branch distance for the *less-than* condition [47], we define UOD^{act}_f for this kind of actuators as follows:

$$UOD^{act}_f(i) = \begin{cases} v^f_{act}(i) - v_{act}(i), & \text{if } v_{act}(i) < v^f_{act}(i) \\ 0, & \text{otherwise} \end{cases}$$

We compute $UOD_f(i) = \sum_{act \in Act^f} UOD^{act}_f(i)$ where each UOD^{act}_f is defined as either one of the above equations depending on the type of *act*. The UOD_f function is our *unsafe overriding distance* function. Specifically, $UOD_f(i) = 0$ implies that *IntC* unsafely overrides the output of f at step i. Similarly, a small or large value of $UOD_f(i)$ indicates that a test case is, respectively, close to or far from causing *IntC* to unsafely override f at step i.

Combined distances. We now describe how we combine the three distance functions to obtain our final hybrid test objectives for detecting undesired feature interactions. Note that *coverage distance*, *failure distance* and *unsafe overriding distance* have different units of measure (e.g., km/h, meters) and different ranges. Thus, we first normalize these distances before combining them into one single hybrid function. To this aim, we rely on the well-known rational function $\omega_1(x) = x/(x+1)$ since prior studies [9] have empirically shown that, compared to other normalization functions, it provides better guidance to the search for minimization problems (e.g., distance functions in our case). In the following, we denote the normalized forms of the functions above as $\overline{FD}$, $\overline{UOD}$ and $\overline{BD}$, respectively.

To maximize the likelihood of detecting undesired feature interactions, we aim to execute every branch of *IntC* such that while executing that branch, *IntC* unsafely overrides every feature f, and further, its outputs violate every safety requirement related to f. Therefore, for every branch j of *IntC*, every safety requirement l of $\mathcal{F}$, and every simulation time step i, we define a hybrid distance $\Omega_{j,l}(i)$ as follows:

$$\Omega_{j,l}(i) = \begin{cases} \overline{BD}_j(i) + \overline{UOD}_{max} + \overline{FD}_{max} & \text{(1) If } j \text{ is not covered } (\overline{BD}_j(i) > 0) \\ \overline{UOD}_f(i) + \overline{FD}_{max} & \text{(2) If } j \text{ is covered, but } f \text{ is not unsafely} \\ & \quad \text{overridden } (\overline{BD}_j(i) = 0 \wedge \overline{UOD}_f(i) > 0) \\ \overline{FD}_l(i) & \text{(3) Otherwise } (\overline{BD}_j(i) = 0 \wedge \overline{UOD}_f(i) = 0) \end{cases}$$

where f is the feature responsible for the requirement l, while $\overline{FD}_{max} = 1$ and $\overline{UOD}_{max} = 1$, indicating the maximum value of the normalized functions.

Each hybrid distance function $\Omega_{j,l}(i)$ is defined for each simulation step i. Corresponding to each hybrid distance function, we define a *test objective* $\Omega_{j,l}$ for the entire simulation time interval as follows: $\Omega_{j,l} = Min\{\Omega_{j,l}(i)\}_{0 \le i \le \frac{T}{\delta}}$. Given a test case tc, each test objective $\Omega_{j,l}(tc)$ always yields a value in [0..3]; $\Omega_{j,l}(tc) > 2$ indicates that tc has not covered branch j; $2 \ge \Omega_{j,l}(tc) > 1$ indicates that tc has covered branch j, but has not caused *IntC* to unsafely override some feature f related to requirement l; $1 \ge \Omega_{j,l}(tc) > 0$ indicates that tc has covered branch j, and has caused *IntC* to unsafely override some feature f related to requirement l, but has not violated requirement l; and finally, $\Omega_{j,l}(tc)$ is zero when tc has covered branch j, has caused *IntC* to unsafely override some feature f related to l and has violated requirement l.

3.4 Search Algorithm

When testing a system we do not know a priori which safety requirements may be violated. Neither do we know in which branches of *IntC* the violations may be detected. Therefore, we search for any violation of system safety requirements that may arise when exercising any branch of *IntC*. This leads to $k \times n$ test objectives where k is the number of branches of *IntC* and n is the number of safety requirements. More formally, given a feature-based control system $\mathcal{F}$ under test, our test generation problem can be formulated as follows:

Definition. Let $\Omega = \{\Omega_{1,1}, \ldots, \Omega_{k,n}\}$ *be the set of test objectives for* $\mathcal{F}$*, where* k *is the number of branches in IntC and* n *is the number of safety requirements of* $\mathcal{F}$*. Find a test suite that covers as many objectives* $\Omega_{i,j}$ *as possible.*

Our problem is many-objective as we attempt to optimize a relatively large number of test objectives. As a consequence, we have to consider many-objective optimization algorithms, which are a class of search algorithms suitably defined for problems with more than three objectives. Various many-objective metaheuristics have been proposed in the literature, such as NSGA-III [33], HypE [12]. These algorithms are designed to produce different alternative trade-offs that can be made among the search objectives [48].

Recently, Panichella et al. [59, 60] argued that the purpose of test case generation is to find test cases that separately cover individual test objectives rather than finding solutions capturing well-distributed and diverse trade-offs among the search objectives. Hence, they introduced a new search algorithm, namely MOSA [59], that (i) rewards test cases that cover at least one objective over those that yield a low value on several objectives without covering any; (ii) focuses the search on the yet uncovered objectives; and (iii) stores all tests covering one or more objectives into an *archive*. MOSA has been introduced in the context of white-box unit testing and has shown to outperform alternative search algorithms [59, 60].

Algorithm 1: Feature Interaction Testing (FITEST)

```
   Input: Ω: Set of objectives
   Result: A: Archive
1  begin
2      P ⟵ ADAPTIVE-RANDOM-POPULATION(| Ω |)
3      W ⟵ CALCULATE-OBJECTIVES(P, Ω)
4      [Ω^c, T_c] ⟵ GET-COVERED-OBJECTIVE(P,W)
5      A ⟵ T_c
6      Ω ⟵ Ω − Ω^c
7      while not (stop_condition) do
8          Q ⟵ RECOMBINE(P)
9          Q ⟵ CORRECT-OFFSPRINGS(Q)
10         W ⟵ CALCULATE-OBJECTIVES(Q, Ω)
11         [Ω^c, T_c] ⟵ GET-COVERED-OBJECTIVE(P,W)
12         A ⟵ A ∪ T_c                          // Update the archive
13         Ω ⟵ Ω − Ω^c                  // Update the set of objectives
14         F_0 ⟵ ENVIRONMENTAL-SELECTION(P ∪ Q, Ω)
15         P ⟵ F_0                                  // New population
16     return A
```

In this paper, we introduce FITEST, a novel search algorithm that extends MOSA and adapts it to testing feature-based self-driving systems. Below, we describe the main loop of FITEST whose pseudo-code is shown in Algorithms 1. We then discuss the differences between FITEST and MOSA.

Main loop. As Algorithm 1 shows, FITEST starts by generating an initial set P of randomly generated test cases (line 2), called *population*. Each test case $X \in P$ is a vector of inputs required to simulate the SUT (e.g., see Figure 3). After simulating each test $X \in P$, the test objectives $\Omega_{j,l}$ for X are computed based on the simulation results (see Section 3.3). Next, tests are evolved through subsequent iterations (loop in lines 7-16), called *generations*. In each generation, the *binary tournament selection* [34] is used to select pairs of fittest test cases for reproduction. During reproduction (line 8), two tests (*parents*) are recombined to form new test cases (*offsprings*) using the *crossover* and *mutation* operators. Finally, fittest tests are selected among the parents and offsprings to form the new population for the next generation (line 14). Below, we describe the new and specific features of FITEST.

Initialization. The size of the initial population in FITEST is equal to the number of test objectives. This is because, in our context, running each single test case is expensive, taking up to few minutes, as it requires running computationally intensive simulations. Hence, in FITEST, we aim to cover each test objective at most once by at most one test case. Therefore, we do not need to start the search with a population larger than the number of test objectives.

We select the initial population such that it includes a diverse and randomly selected set of test input vectors. This is because we aim to include different traffic situations, (e.g., different trajectory angles and speeds of pedestrians) in our initial population. To do so, we use an adaptive random search algorithm [51], which is an extension of the naive random search that attempts to maximize the Euclidean distance between the vectors selected in the input space. In contrast to FITEST, the initial population in MOSA is a set of randomly generated tests without any diversity mechanism, and the size of the population is an input parameter of the algorithm.

Genetic recombination. Since our test inputs (i.e., X) are vectors of float values (see Figure 3), we use two widely-used genetic operators proposed for real number solution encodings: the *simulated binary crossover* [30] (SBX) and the *gaussian mutation* [32]. Prior studies [32, 42] show that, for numerical vectors, these operators outperform the more classical ones. In contrast, MOSA uses the classical *single-point crossover* and *uniform* mutation implemented in EvoSuite [38] to handle different types of test data, e.g., strings, Java objects, etc.

Correction operator. Recall from Section 3.2 that our test inputs are characterized by constraints. Hence, genetic operators may yield invalid tests (e.g., a test input where the leading car is behind the ego car). To modify and correct such cases, FITEST applies *correction operators* (line 9 in Algorithm 1). For example, in *SafeDrive*, if after applying genetic operators, the leading car position (x_0^l) and speed (v_o^l), and the traffic sign position (x^{ts}) violate any of the constraints described in Section 3.2, we discard their values and randomly select new values for these variables within ranges enforced by the ego car position (x_0^e) and speed (v_0^e).

Archive. Similar to MOSA, every time new tests are generated and evaluated (either at the beginning or during the search), FITEST uses the GET-COVERED-OBJECTIVE routine to identify newly covered objectives and the test cases covering them. These objectives are removed from the set of test objectives (line 6, 13) to not be used by the environmental selection in the subsequent iterations. Further, test cases covering the removed test objectives are put in an *archive* [59, 60, 64] (i.e., A). The archive at the end contains the FITEST results. Each test case in the archive covers one of the test objectives being satisfied during the search. Note that some test objectives may not be covered within the search time or they may be infeasible (unreachable).

Environmental selection. In FITEST, at each iteration, a new population with a size not necessarily the same as the previous population size is formed (line 15 in Algorithm 1) by selecting, for each uncovered test objective $\Omega_{i,j}$, the test case in $P \cup Q$ that is closest to covering that objective (*preference criterion* [59]). The population size at each iteration is lower than the number of objectives. It can even be less than the number of test objectives because a single test case may be selected as the closest (fittest) test for multiple objectives. Further, the population size is likely to decrease over iterations since, at each iteration, test objectives are covered and excluded from the environmental selection in the subsequent iterations.

The population size represents the main difference between FITEST and similar search-based test generation algorithms. In classical many-objective search algorithms, the environment selection chooses a fixed number N of tests (i.e., to maintain a constant population size) from offsprings and their parents (i.e., from $P \cup Q$) using the *Pareto optimality* [31, 34] (i.e., selecting solutions that are non-dominated by any other solutions in $P \cup Q$). In MOSA, the population size is kept constant as well but the selection is performed by first selecting the test cases in the first front F_0 built using the preference criterion; then, if the size of F_0 is less than N, MOSA uses the *Pareto optimality* criterion to select enough test cases such that in total N test cases are selected.

In contrast, FITEST minimizes the number of test cases generated at each search iteration by evolving only test cases that are closest to satisfying uncovered objectives, i.e., those in F_0. This helps reducing the search computation time compared to existing many-objective

search algorithms that typically maintain and evolve a fixed number of solutions at each iteration. This is particularly important in the context of our work, since running each test case is expensive.

4 EVALUATION

In this section, we evaluate our approach to detecting undesired feature interactions using real-world automotive systems.

4.1 Research Questions

The goal of our study is to assess how effectively our hybrid test objectives (hereafter referred to as *Hybrid*) guide the search toward revealing feature interaction failures. As described in Section 3.3, *Hybrid* builds on three distance functions: (1) coverage, (2) failure and (3) unsafe overriding. Among these, *coverage distance* is a well-known heuristic that has been extensively used in white-box testing [38, 39, 55]. For example, Fraser and Arcuri [39] showed that pure coverage-based distance can be used to generate unit tests capable of detecting real faults. Variations of the failure distance have also been used in different contexts to generate tests revealing requirements violations [4, 13, 20]. Therefore, we want to assess whether *Hybrid* provides any benefits compared to pure coverage-based and failure-based objectives. In particular, we formulate the following research questions:

RQ. *Does Hybrid reveal more feature interaction failures compared to coverage-based and failure-based test objectives?*

Coverage based-objectives, hereafter referred to as *Cov*, correspond to the *BD* functions described in Section 3.3 and are computed as the sum of the approach level [56] and the normalized branch distance [56]. Therefore, *Cov* aims to execute as many branches of *IntC* as possible.

Failure-based test objectives, hereafter referred as to *Fail*, aim to generate test cases that execute as many branches of *IntC* as possible while violating as many system safety requirements as possible when executing each branch. Thus, *Fail* is defined by combining branch distance *BD* and failure distance *FD* functions described in Section 3.3. More precisely, for each branch j of *IntC* and every safety requirement l of $\mathcal{F}$, a failure-based test objective is defined as $Min\{Fail_{j,l}(i)\}_{0 \leq i \leq \frac{T}{\delta}}$ where

$$Fail_{j,l}(i) = \begin{cases} \overline{BD}_j(i) + \overline{FD}_{max} & \text{if } j \text{ is not covered} \\ \overline{FD}_l(i) & \text{otherwise} \end{cases}$$

In this paper, we focus our empirical evaluation on comparing *Hybrid* with alternative test objectives, but we do not compare FITEST with alternative many-objective search algorithms because, as discussed in Section 3.4, our changes to MOSA are primarily motivated by the practical needs of (1) using genetic operators for numerical vectors (often called real-coded operators [32, 42]) and (2) lowering the running time of our algorithm by reducing the number of (expensive) fitness computations at each generation. In our preliminary experiments, running MOSA with its default population size of 50 [59] required more than 24 hours for only 10 generations. Further, previous studies showed that MOSA, which is the algorithm underlying FITEST, outperforms other search-based algorithms in unit testing, such as random search [26], whole suite search [26, 59], and other many-objective evolutionary algorithms [60].

4.2 Case Study Systems

We evaluate our approach by applying it to two case study systems developed by IEE. Both systems contain the four self-driving features introduced in Section 2. However since engineers had developed two alternative sets of rules to prioritize these features and to resolve their undesired interactions, they developed two different function models for the integration component (i.e., *IntC*). Due to confidentiality reasons, we do not share the details of the *IntC* models used in these two systems. Both systems are developed in Matlab/Simulink and can be integrated into PreScan, the simulator used in this paper. We refer to these systems as *SafeDrive1* and *SafeDrive2*.

4.3 Experimental Settings

For the genetic operators used in FITEST, we use the parameter values suggested in the literature [21, 29, 34]: We use the *simulated binary crossover* (SBX) with a crossover probability 0.60, as the recommended interval is $[0.45, 0.95]$ [21, 29]. The gaussian mutation changes the test inputs by adding a random value selected from a normal distribution $G(\mu, \sigma)$ with mean $\mu = 0$ and variance $\sigma^2 = 1.0$. As the guidelines suggest [34], the mutation probability is set to $1/l$ where l is the length of test inputs (chromosomes). In FITEST, we do not need to manually set the population size since, as described in Section 3.4, it is dynamically updated at each generation. The search stops when all the objectives are covered or when the timeout of 12 hours is reached. We set a timeout of 12 hours because as we will discuss in Section 4.4, the search results start to stabilize and reach a plateau within this time budget. Further, according to domain experts, longer search time budgets are not practical.

To account for the randomness of the search algorithm, FITEST was executed 20 times on each case study system and with each of the three test objectives. The total duration of the experiment was 20 (repetitions) × 2 (systems) × 3 (test objectives) × 12 (hours) = 1440 hours (60 days). All experiments were executed on the same machine with a 2.5 GHz Intel Core i7-4870HQ CPU and 16 GB DDR3 memory.

We use *the number of feature interaction failures* that each of the test objectives in our study can reveal as our evaluation metric. We compute this metric by automatically checking test cases generated by each test objective to determine whether or not they reveal a feature interaction failure. A test case reveals a feature interaction failure iff: (1) it violates some system safety requirement in Table 1 when it is applied to a system consisting of multiple features, but (2) it does not violate that same safety requirement when it is applied to the feature responsible for the satisfaction of that requirement. Specifically, a test case *tc* reveals a feature interaction if $FD_i(tc) = 0$ for some safety requirement i when *tc* is applied to *SafeDrive1* or *SafeDrive2*, but $FD_i(tc) > 0$ when *tc* is applied to the feature responsible for requirement i.

4.4 Results

In this section, we answer our research question by comparing *Hybrid*, *Fail* and *Cov* test objectives. Specifically, we run FITEST with *Hybrid*, *Fail* and *Cov* as test objectives separately and repeat each run for 20 times. Figures 5(a) and (b) compare the number of feature interaction failures identified over different runs of FITEST with *Hybrid*, *Fail* and *Cov* applied to SafeDrive1 and SaveDrive2,

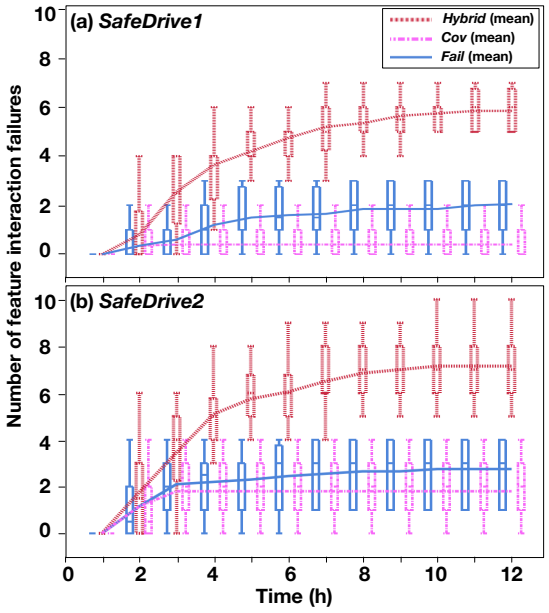

Figure 5: The number of feature interaction failures found by *Hybrid*, *Fail* and *Cov* over time for (a) *SafeDrive1* and (b) *SafeDrive2* systems.

respectively. We show the results at every one-hour interval from 0 to 12h. As shown in the two figures, the average number of feature interaction failures computed using *Hybrid* is always larger than those identified by *Fail* and *Cov*. Specifically, after 12h, on average, *Hybrid* is able to find 5.9 and 7.2 feature interaction failures for SafeDrive1 and SaveDrive2, respectively. In contrast, *Fail* uncovers, on average, 2.1 and 2.8 feature interaction failures for SafeDrive1 and SaveDrive2, respectively; and *Cov* only uncovers, on average, 0.4 and 1.8 feature interaction failures for SafeDrive1 and SaveDrive2, respectively. Further, after executing the algorithms for 10h, the results obtained by the three test objective alternatives reach a plateau.

Note that every run of FITEST with *Hybrid*, *Fail* and *Cov* achieved 100% branch coverage on the function model of the integration component (i.e., *IntC*) for both *SafeDrive1* and *SafeDrive2*. Hence, *Fail* and *Cov*, despite being able to exercise all branches of *IntC*, perform poorly in terms of the number of feature interaction failures that they can reveal. Further, we note that, among the *Hybrid*, *Fail* and *Cov* test objectives, only *Cov* was fully achieved by the generated test suites, while the *Hybrid* and *Fail* test objectives were only partially achieved. This is expected since, as discussed in Section 3.4, *Hybrid* and *Fail* search for violations of every safety requirement at every branch of *IntC*. Some of these test objectives may be infeasible (uncoverable) because not all safety requirements may be violated at every branch of *IntC*. However, we cannot know a priori which objectives are infeasible, and hence, we include all of them in our search.

We compare the results in Figure 5 using a statistical test. Following existing guidelines [10], we use the non-parametric pairwise Wilcoxon rank sum test [27] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [71]. Table 2 reports the results of the statistical tests obtained when comparing the number of feature interaction failures uncovered by *Hybrid*, *Fail* and *Cov*, over time for *SafeDrive1* and *SafeDrive2*. As shown in the table, the *p*-values related to the results produced when the search time ranges between 3h and 12h are all lower than 0.05 and the $\hat{A}_{12}$ statistics show large effect sizes. Hence, the number of feature interaction failures obtained by *Hybrid* is significantly higher (with a large effect size) than those obtained by *Fail* and *Cov*.

Table 2: Statistical test results comparing the number of feature interaction failures found by *Hybrid*, *Fail* and *Cov* over time for *SafeDrive1* and *SafeDrive2* systems (see Figure 5).

	SafeDrive1				*SafeDrive2*			
	Hybrid vs. *Cov*		*Hybrid* vs. *Fail*		*Hybrid* vs. *Cov*		*Hybrid* vs. *Fail*	
time	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
1h	NA	0.5 (N)	NA	0.5 (N)	NA	0.5 (N)	NA	0.5 (N)
2h	0.663	0.53 (N)	0.663	0.53 (N)	0.33	0.58 (S)	0.33	0.58 (S)
3h	8.83e-6	0.89 (L)	5.16e-5	0.86 (L)	0.003	0.77 (L)	0.009	0.73 (L)
4h	7.02e-8	0.98 (L)	4.68e-6	0.91 (L)	1.97e-7	0.97 (L)	5.27e-7	0.95 (L)
5h	3.08e-8	0.99 (L)	4.71e-7	0.95 (L)	9.97e-8	0.99 (L)	1.65e-7	0.98 (L)
6h	3.2e-8	1 (L)	1.43e-7	0.98 (L)	7.14e-8	0.99 (L)	1.0e-7	0.98 (L)
7h	3.32e-8	1 (L)	1.02e-7	0.98 (L)	5.52e-8	0.99 (L)	6.65e-8	0.99 (L)
8h	3.25e-8	1 (L)	7.78e-8	0.99 (L)	5.40e-8	1 (L)	4.74e-8	1 (L)
9h	2.9e-8	1 (L)	4.3e-8	1 (L)	5.54e-8	1 (L)	4.86e-8	1 (L)
10h	2.84e-8	1 (L)	4.16e-8	1 (L)	5.58e-8	1 (L)	4.98e-8	1 (L)
11h	2.96e-8	1 (L)	4.4e-8	1 (L)	5.58e-8	1 (L)	4.98e-8	1 (L)
12h	2.96e-8	1 (L)	4.23e-8	1 (L)	5.58e-8	1 (L)	4.98e-8	1 (L)

The answer to **RQ** *is that* our proposed test objectives (*Hybrid*) reveals significantly more feature interaction failures compared to coverage-based and failure-based test objectives. In particular, on average, *Hybrid* identifies more than twice as many feature interaction failures as the coverage-based and failure-based test objectives.

Feedback from domain experts. We conclude this section by summarizing the qualitative feedback of the domain experts from IEE with whom we have been collaborating on the research presented in this paper. During two meetings, we presented to our domain experts four test scenarios revealing different feature interaction failures. The four test scenarios were selected randomly among the ones detected by our approach. Each test scenario *tc* was presented by showing: (1) a video simulation of *tc* generated by PreScan based on one of our case study systems (*SafeDrive1* or *SafeDrive2*) and violating one of the safety requirements in Table 1 and (2) a video simulation of *tc* generated by PreScan based on running only the feature related to the violated requirement. Note that since *tc* reveals a feature interaction failure, the latter simulation videos (i.e., the ones based on running individual features) do not exhibit any requirements violation. After presenting the simulations, we discussed with our domain experts each failure, its root causes and whether or how it can be addressed by modifying the current feature interaction resolution rules implemented in *IntC*. We drew the following conclusions from our discussions: (1) Our domain experts agreed with us that the four failures were due to interactions between the features and were not caused by faults in individual features, (2) they confirmed that the failures were not previously known to them and (3) they identified ways to modify or extend the integration component (*IntC*) to avoid the failures. The simulations and the detailed failure descriptions used in our meetings are available online [2].

5 RELATED WORK

In this section, we discuss and compare with different strands of related research in the areas of testing autonomous cars, and testing and model checking feature-based systems.

Testing autonomous cars. Search-based approaches have been used for black-box testing of driver-assistance features [13, 14, 22, 23]. Bühler and Wegener use a single-objective search algorithm to test a vehicle-to-vehicle braking assistance [23] and an autonomous parking feature [22]. Ben Abdessalem et. al. rely on multi-objective search [13] and learnable evolutionary algorithms [14] to generate test cases violating safety requirements of self-driving systems. Recently, Tian et. al. [68] proposed a notion of neuron coverage and used it to guide the generation of tests for neural networks used in autonomous cars. None of these approaches study the feature interaction problem in autonomous cars. We advance the research on testing autonomous cars by devising test objectives that specifically detect feature interaction failures. Our test objectives combine existing software testing heuristics (i.e., branch-coverage [38, 55, 69] and failure-based [4, 13, 20, 23]) with our proposed unsafe overriding heuristic. Further, we tailor existing many-objective search algorithms [59, 60] to detect feature interaction failures in our context.

Feature interactions in software product lines. In the context of software product lines (SPL), testing approaches are proposed to ensure product implementations satisfy their feature specifications [50, 58, 61]. These approaches largely follow a model-based testing paradigm [6]. For example, they use combinatorial testing to drive test cases and oracles from feature models to verify individual products [58, 61]. Our work, in contrast, is model testing [19]. Specifically, we take advantage of the availability of executable function models and test executable function models of the system and its environment. Further, in contrast to the SPL testing work, our approach does not need descriptions of features and their dependencies to be provided.

Some SPL approaches are proposed to automatically derive feature dependencies specifying valid feature combinations [7, 36, 46]. For example, interactions between observable feature behaviors (i.e., external feature interactions [7]) have been identified by static analysis of software code [36, 46]. In contrast, our approach detects feature interactions prior to any software coding. It dynamically detects undesired feature interactions by testing function models capturing the SUT and its environment.

Feature interaction detection via model checking. Several approaches are proposed to detect feature interactions by model checking requirements or design artifacts against formal specifications [8, 11, 45, 62, 65]. For example, Apel et. al. [8] verify features described in a formal feature-oriented language against temporal logic properties [28]. Arora et. al. verify features defined as state machines against live sequence charts specifications. Dominguez et. al. [45] verify features captured as StateFlows, and Sobotka and J. Novak [65] specify features in timed automata [5]. Similar to our work, these approaches verify early requirements and design models against system requirements. However, our work differs with this line of research in the following ways: First, most of these approaches identify pairwise feature interactions only. We can, however, identify feature interactions between an arbitrary number of features. Second, these techniques model system features only. However, to analyze autonomous cars, we have to capture, in addition to features, system's sensors and actuators, and the system environment. Third, in contrast to these approaches, our approach does not require additional formal modeling. We take advantage of the availability of function models, which are developed anyway in the CPS domain, to test the system in its environment. Fourth, our function models use numerical and continuous Matlab/Simulink computations to capture dynamics of cars and pedestrians. These models are not, in general, amenable to model checking due to scalability and incompatibility issues [3, 52, 54]. Therefore, as suggested in the recent research on testing CPS models [3, 52, 54, 78], instead of model checking, we rely on simulation-based testing guided by meta-heuristics to analyze our function models.

Feature interaction resolution. Several approaches are proposed to devise resolution strategies to eliminate undesired feature interactions, for example, by proposing specific feature-oriented architectures [44, 70], by statically prioritizing features [41, 77] or using runtime resolution mechanisms [16, 76]. These techniques are complementary to our approach. They can be used to develop the integration component (*IntC*) to resolve undesired feature interactions, but our approach is still necessary to test the system behavior and to determine if the proposed resolution strategy can eliminate undesired behaviors under different environment conditions.

6 CONCLUSION

We presented a technique for detecting feature interaction failures in the context of autonomous cars. Our technique is based on analyzing executable function models typically developed in the cyber physical domain to specify system behaviors at early development stages. Our contributions over prior work include: (1) casting the problem of detecting undesired feature interactions into a search-based testing problem, (2) defining a test guidance that combines existing search-based test objectives with new heuristics specifically aimed at revealing feature interaction failures, (3) tailoring existing many-objective search algorithms [59, 60] to automatically reveal feature interaction failures in a scalable way, and (4) evaluating our approach using two versions of an industrial self-driving system and demonstrating significant improvement in feature interaction failure identification compared to baseline search-based testing approaches. Finally, we note that our research was motivated and carried out in the context of a partnership with IEE. The feedback from domain experts from IEE indicates that the detected feature interaction failures represent real faults in their systems that were not previously identified based on analysis of the system features and their requirements.

In future, we plan to devise strategies to use feature interaction failures to localize faults and help engineers effectively debug and refine their feature interaction resolution strategies.

ACKNOWLEDGMENTS

We gratefully acknowledge the funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277) and from IEE S.A. Luxembourg.

REFERENCES

[1] 2018. Matlab/Simulink. https://nl.mathworks.com/products/simulink.html. (2018).

[2] 2018. Supplementary Materials. https://figshare.com/s/50193ea5652147d2f036. (2018).

[3] Houssam Abbas, Georgios Fainekos, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. 2013. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 2s (2013), 95.

[4] Wasif Afzal, Richard Torkar, and Robert Feldt. 2009. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology* 51, 6 (2009), 957–976.

[5] Rajeev Alur. 1999. Timed automata. In *Proceedings of the International Conference on Computer Aided Verification (CAV'99)*. Springer, Trento, Italy, 8–22.

[6] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press, New York, NY, USA.

[7] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD'13)*. ACM, Indianapolis, USA, 1–8.

[8] Sven Apel, Alexander Von Rhein, Thomas ThüM, and Christian KäStner. 2013. Feature-interaction detection based on feature-based specifications. *Computer Networks* 57, 12 (2013), 2399–2409.

[9] Andrea Arcuri. 2013. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability* 23, 2 (2013), 119–147. https://doi.org/10.1002/stvr.457

[10] 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.

[11] Silky Arora, Prahlad Sampath, and S Ramesh. 2012. Resolving uncertainty in automotive feature interactions. In *Proceedings of the International Requirements Engineering Conference (RE'12)*. Chicago, Illinois, USA, 21–30.

[12] Johannes Bader and Eckart Zitzler. 2011. HypE: An algorithm for fast hypervolume-based many-objective optimization. *IEEE Transactions on Evolutionary computation* 19, 1 (2011), 45–76.

[13] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the International Conference on Automated Software Engineering (ASE'16)*. IEEE, Singapore, 63–74.

[14] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms. In *Proceedings of the International Conference on Software Engineering (ICSE'18)*. ACM, Gothenburg, Sweden, to appear.

[15] Johan Blom, Bengt Jonsson, and Lars Kempe. 1994. Using Temporal Logic for Modular Specification of Telephone Services. In *Proceedings of the International Workshop on Feature Interactions in Telecommunications Systems (FIW'94)*. IOS Press, Amsterdam, Netherlands, 197–216.

[16] Cecylia Bocovich and Joanne M Atlee. 2014. Variable-specific resolutions for feature interactions. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, Hong Kong, China, 553–563.

[17] Kenneth H. Braithwaite and Joanne M. Atlee. 1994. Towards automated detection of feature interactions. In *Proceedings of the International Workshop on Feature Interactions in Telecommunications Systems (FIW'94)*. IOS Press, Amsterdam, Netherlands, 36–59.

[18] J. Bredereke. 2000. Families of formal requirements in telephone switching. In *Proceedings of the International Workshop on Feature Interactions in Telecommunications and Software Systems(FIW'00)*. IOS Press, Glasgow, Scotland, UK, 257–273.

[19] Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. 2016. Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the International Conference on Software Engineering Companion (ICSE'16)*. ACM, Austin, TX, US, 789–792.

[20] Lionel C Briand, Yvan Labiche, and Marwa Shousha. 2006. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines* 7, 2 (2006), 145–170.

[21] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. 2006. Using Genetic Algorithms for Early Schedulability Analysis and Stress Testing in Real-time Systems. *Genetic Programming and Evolvable Machines* 7, 2 (2006), 145–170.

[22] Oliver Bühler and Joachim Wegener. 2004. *Automatic testing of an autonomous parking system using evolutionary computation.* Technical Report. SAE Technical Paper.

[23] Oliver Bühler and Joachim Wegener. 2008. Evolutionary functional testing. *Computers & Operations Research* 35, 10 (2008), 3144–3160.

[24] Stan BÃijhne, Kim Lauenroth, and Klaus Pohl. 2004. Modelling Features for Multi-Criteria Product-Lines in the Automotive Industry.. In *Proceedings of the International Workshop on Software Engineering for Automotive Systems (SEAS'04), co-located at ICSE'04*. Edinburgh, UK, 9–16.

[25] Muffy Calder, Mario Kolberg, Evan H Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41, 1 (2003), 115–141.

[26] José Campos, Yan Ge, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2017. An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE'17)*. Paderborn, Germany, 33–48.

[27] J. Anthony Capon. 1991. *Elementary Statistics for the Social Sciences: Study Guide.* Wadsworth Publishing Company, Belmont, CA, USA.

[28] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking.* MIT Press.

[29] Helen G. Cobb and John J. Grefenstette. 1993. Genetic Algorithms for Tracking Changing Environments. In *Proceedings of the International Conference on Genetic Algorithms (ICGA'93)*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 523–530.

[30] Kalyanmoy Deb. 1995. Simulated binary crossover for continuous search space. *Complex systems* 9 (1995), 115–148.

[31] Kalyanmoy Deb. 2014. Multi-objective Optimization. In *Search Methodologies.* Springer US, 403–449. https://doi.org/10.1007/978-1-4614-6940-7_15

[32] Kalyanmoy Deb and Debayan Deb. 2014. Analysing Mutation Schemes for Real-parameter Genetic Algorithms. *International Journal of Artificial Intelligence and Soft Computing* 4, 1 (Feb 2014), 1–28. https://doi.org/10.1504/IJAISC.2014.059280

[33] Kalyanmoy Deb and Himanshu Jain. 2014. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation* 18, 4 (2014), 577–601.

[34] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2000. A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6 (2000), 182–197.

[35] Stefan Ferber, Jürgen Haag, and Juha Savolainen. 2002. Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In *Proceedings of the International Conference on Software Product Lines (SPLC'02)*. Springer, San Diego, CA, USA, 235–256.

[36] Gabriel Ferreira, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2015. Characterizing complexity of highly-configurable systems with variational call graphs: analyzing configuration options interactions complexity in function calls. In *Proceedings of the Symposium and Bootcamp on the Science of Security (HotSoS'15)*. ACM, Urbana, IL, USA, 17.

[37] K. Fisler and S. Krishnamurthi. 2005. Decomposing Verification by Features. In *Proceedings of the International Conference on Verified Software: Theories, Tools and Experiments (VSTTE'05)*. Zurich, Switzerland.

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

[39] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. *Empirical Software Engineering* 20, 3 (2015), 611–639.

[40] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based Software Engineering: Trends, Techniques and Applications. *Comput. Surveys* 45, 1, Article 11 (Dec 2012), 61 pages. https://doi.org/10.1145/2379776.2379787

[41] Jonathan D Hay and Joanne M Atlee. 2000. Composing features and resolving interactions. In *ACM SIGSOFT Software Engineering Notes (SEN'00)*, Vol. 25. ACM, 110–119.

[42] F. Herrera, M. Lozano, and A. M. SÂůnchez. 2003. A taxonomy for the crossover operator for real-coded genetic algorithms: An experimental study. *International Journal of Intelligent Systems* 18, 3 (2003), 309–338. https://doi.org/10.1002/int.10091

[43] IEE. 2018. International Electronics & Engineering. https://www.iee.lu/. (2018).

[44] M. Jackson and P. Zave. 1998. Distributed Feature Composition: a Virtual Architecture for Telecommunications Services. *IEEE TSE* 24, 10 (1998), 831–847.

[45] Alma L Juarez-Dominguez, Nancy A Day, and Jeffrey J Joyce. 2008. Modelling feature interactions in the automotive domain. In *Proceedings of the International Workshop on Modeling in Software Engineering (MISE'08)*. ACM, Leipzig, Germany, 45–50.

[46] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. 2017. On the Relation of External and Internal Feature Interactions: A Case Study. *arXiv preprint arXiv:1712.07440* (2017).

[47] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering* 16, 8 (1990), 870–879.

[48] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. 2015. Many-objective evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)* 48, 1 (2015), 13.

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

[50] Malte Lochau, Sebastian Oster, Ursula Goltz, and Andy Schürr. 2012. Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Journal* 20, 3-4 (2012), 567–604.

[51] Sean Luke. 2013. *Essentials of Metaheuristics* (second ed.). Lulu, Fairfax, Virginie, USA. https://cs.gmu.edu/$\sim$sean/book/metaheuristics/

[52] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann. 2018. Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior. *IEEE Transactions on Software Engineering* (2018), to appear.

[53] Reza Matinnejad, Shiva Nejati, Lionel Briand, Thomas Bruckmann, and Claude Poull. 2015. Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology* 57 (2015), 705–722.

[54] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2016. Automated test suite generation for time-continuous simulink models. In *Proceedings of the International Conference on Software Engineering (ICSE'16)*. ACM, Austin, TX, US, 595–606.

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

[56] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software Testing Verification and Reliability Journal* 14, 2 (2004), 105–156.

[57] N. S. Nise. 2004. *Control Systems Engineering, 4th ed.* John-Wiely Sons.

[58] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. 2011. Pairwise feature-interaction testing for SPLs: potentials and limitations. In *Proceedings of the International Software Product Line Conference, Volume 2 (SPLC'11)*. ACM, Munich, Germany, 6.

[59] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *Proceedings of the International Conference on Software Testing, Verification and Validation, (ICST'15)*. Graz, Austria, 1–10.

[60] Annibale Panichella, Fitsum Mesheha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (Feb 2018), 122–158.

[61] Sachin Patel, Priya Gupta, and Vipul Shah. 2013. Feature interaction testing of variability intensive systems. In *Proceedings of the International Workshop on Product Line Approaches in Software Engineering (PLEASE'13)*. IEEE, San Francisco, CA, USA, 53–56.

[62] Malte Plath and Mark Ryan. 2001. Feature integration using a feature construct. *Science of Computer Programming* 41, 1 (2001), 53–84.

[63] C. Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*. JyvÃďskylÃď, Finland, 419–443.

[64] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893. https://doi.org/10.1007/s10664-015-9424-2

[65] Jan Sobotka and Jiri Novak. 2013. Automation of automotive integration testing process. In *Proceedings of the International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS'13)*, Vol. 1. IEEE, Berlin, Germany, 349–352.

[66] Thorsten Suttorp and Christian Igel. 2006. Multi-objective optimization of support vector machines. In *Multi-objective machine learning*. Springer, -, 199–220.

[67] TASS-International. 2018. PreScan. https://www.tassinternational.com/prescan. (2018).

[68] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars. In *Proceedings of the International Conference on Software Engineering (ICSE'18)*. ACM, Gothenburg, Sweden, to appear.

[69] Paolo Tonella. 2004. Evolutionary testing of classes. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*, Vol. 29. ACM, Boston, MA, USA, 119–128.

[70] Rob van der Linden. 1994. Using an architecture to help beat feature interaction. In *Proceedings of the International Workshop on Feature Interactions in Telecommunications Systems (FIW'94)*. IOS Press, Amsterdam, Netherlands, 24–35.

[71] 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.

[72] Gabriel A Wainer. 2009. *Discrete-event modeling and simulation: a practitioner's approach*. CRC press.

[73] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case selection. In *Proceedings of tthe ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*. ACM, London, UK, 140–150.

[74] Justyna Zander, Ina Schieferdecker, and Pieter J Mosterman. 2017. *Model-based testing for embedded systems*. CRC press.

[75] Pamela Zave. 1993. Feature interactions and formal specifications in telecommunications. *Computer* 26, 8 (Aug 1993), 20–28.

[76] M Hadi Zibaeenejad, Chi Zhang, and Joanne M Atlee. 2017. Continuous variable-specific resolutions of feature interactions. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, Paderborn, Germany, 408–418.

[77] P Ann Zimmer and Joanne M Atlee. 2012. Ordering features by category. *Journal of Systems and Software* 85, 8 (2012), 1782–1800.

[78] Paolo Zuliani, André Platzer, and Edmund M Clarke. 2013. Bayesian statistical model checking with application to Stateflow/Simulink verification. *Formal Methods in System Design* 43, 2 (2013), 338–367.