UNIVERSITÉ DU LUXEMBOURG

PhD-FSTC-2019-38
The Faculty of Sciences, Technology and Communication

Dissertation

Defence held on 14/05/2019 in Luxembourg

to obtain the degree of

Docteur De L'Université Du Luxembourg En Informatique

by

Raja BEN ABDESSALEM EP HELALI
Born on 10th April 1987 in Sousse (Tunisia)

Effective Testing Of Advanced Driver Assistance Systems Using Evolutionary Algorithms And Machine Learning

Dissertation Defense Committee

Prof. Dr. Lionel C. Briand, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg

Dr. Mehrdad Sabetzadeh, Chairman
Senior Research Scientist, University of Luxembourg, Luxembourg

Dr. Shiva Nejati, Debuty Chairman
Senior Research Scientist, University of Luxembourg, Luxembourg

Prof. Dr. Benoit Baudry, Member
Professor, KTH Royal Institute of Technology, Sweden

Dr. Markus Borg, Member
Senior Researcher, RISE Research Institutes of Sweden AB, Sweden

Abstract

Context. Improving road safety is a major concern for most car manufacturers. In recent years, the development of Advanced Driver Assistance Systems (ADAS) has subsequently seen a tremendous boost. The development of such systems requires complex testing to ensure vehicle's safety and reliability. Performing road tests tends to be dangerous, time-consuming, and costly. Hence, a large part of testing for ADAS has to be carried out using physics-based simulation platforms, which are able to emulate a wide range of virtual traffic scenarios and road environments. The main difficulties with simulation-based testing of ADAS are: (1) the test input space is large and multidimensional, (2) simulation platforms provide no guidance to engineers as to which scenarios should be selected for testing, and hence, simulation is limited to a small number of scenarios hand-picked by engineers, and (3) test executions are computationally expensive because they often involve executing high-fidelity mathematical models capturing continuous dynamic behaviors of vehicles and their environment. The complexity of testing ADAS is further exacerbated when many ADAS are employed together in a self-driving system. In particular, when self-driving systems include many ADAS (i.e., features), they tend to interact and impact one another's behavior in an unknown way and may lead to conflicting situations. The main challenge here is to detect and manage feature interactions, in particular, those that violate system safety requirements, hence leading to critical failures. In practice, once feature interaction failures are detected, engineers need to devise resolution strategies to resolve potential conflicts between features. Developing resolution strategies is a complex task and despite the extensive domain expertise, these resolution strategies can be erroneous and are too complex to be manually repaired. In this dissertation, in addition to testing individual ADAS, we focus on testing self-driving systems that include several ADAS.

Approach. In this dissertation, we propose a set of approaches based on meta-heuristic search and machine learning techniques to automate ADAS testing and to repair feature interaction failures in self-driving systems. The work presented in this dissertation is motivated by ADAS testing needs at IEE, a world-leading part supplier to the automotive industry. In this dissertation, we focus on the problem of design time testing of ADAS in a simulated environment, relying on Simulink models. To address the above-mentioned challenges, we propose the following techniques for testing ADAS: (1) We propose a testing approach for ADAS by combining multi-objective search with surrogate models developed based on neural networks. We use multi-objective search to guide testing towards the most critical behaviors of ADAS. Surrogate models enable our testing approach to explore a larger part of the input search space with less computational resources. (2) We propose an automated testing algorithm that combines multi-objective search with decision tree classification models to guide the search-based generation of tests faster towards test scenarios leading to failures. Our approach produces a decision tree model that identifies the regions of a test input space that are likely to contain most test scenarios leading to failures. (3) We propose an automated technique to detect feature interaction failures by casting our approach into a search-based test generation problem. We define a test guidance that combines existing search-based test

objectives with new heuristics specifically aimed at revealing feature interaction failures. (4) We propose a strategy to identify errors in the feature interaction resolution rules for self-driving systems and to automatically repair these errors. We develop a new search-based repairing algorithm that localizes faults and mutates the faulty rules to generate patches.

Our test generation techniques and repairing approach are evaluated using several industrial ADAS from our partner company IEE.

Contributions. The main research contributions in this dissertation are:

1. A testing approach for ADAS that combines multi-objective search with surrogate models to guide testing towards the most critical behaviors of ADAS, and to explore a larger part of the input search space with less computational resources.
2. An automated testing algorithm that builds on learnable evolution models and uses classification decision trees to guide the generation of new test scenarios within complex and multidimensional input spaces and help engineers interpret test results.
3. An automated technique that detects feature interaction failures in the context of self-driving systems based on analyzing executable function models typically developed to specify system behaviors at early development stages.
4. An automated technique that uses a new many-objective search algorithm to localize and repair errors in the feature interaction resolution rules for self-driving systems.

Acknowledgements

I would like to state my sincere gratitude to my supervisor, Lionel Briand, for his invaluable support, feedback and guidance throughout the whole PhD project. I would like to thank him for his significant effort to make my research work match the highest academic standards. I am grateful to have had the opportunity to work with and learn from one of the best researchers in the field.

I would like to express my infinite gratitude to my co-supervisor, Shiva Nejati, for her encouragement, inspiration, and counsel in both personal and professional matters. Her devotion, help and advice were crucial for the realization of this PhD project. Her care and guidance were very constructive and rich so that they helped me to hone my writing and presentation skills.

I would like to express my infinite gratitude to Annibale Pannichella. His availability and comments are a relevant lead to the success of this work. I would like to thank him for his insightful comments, his guidance, and support.

I am grateful to the members of my defense committee: Mehrdad Sabetzadeh, Benoit Baudry and Markus Borg for dedicating time and effort to review this dissertation. I would like to thank further Benoit Baudry and Markus Borg for taking the time to travel to Luxembourg to attend my defense.

The research presented in this thesis was conducted under the collaboration between academia and industry. I would like to thank IEE S.A. Luxembourg for supporting my work and providing the case study systems that were the subject of my empirical studies. In particular, I would like to thank Thomas Stifter for providing me with technical assistance and guidance related to the case study systems, and for sharing his valuable time and insights with us.

Furthermore, I would like to thank my family for believing in me. Special thanks to my dear husband Aymen Helali for his confidence in me, for his patience, love, support and encouragement that motivated me to keep up the hard work during my PhD. I also would like to thank my parents, my sister and my brothers for their lifelong support and for always encouraging me to pursue my dreams.

Last but not least, I would like to thank my colleagues and friends at the Software Verification Lab for the wonderful time we spent together. I am thankful for their helpful advice and for contributing to a positive work environment.

Fonds National de la Recherche Luxembourg

erc

a sense for innovation IEE

Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03), the European Research Council (ERC/694277), and IEE.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Context

Recent years have seen a proliferation of complex Advanced Driver Assistance Systems (ADAS), in particular, in the context of autonomous cars. Examples of ADAS include night vision, collision avoidance systems and traffic sign recognition. ADAS are typically based on vision systems and sensor technologies. They are meant to help drivers by providing proper warnings or by preventing dangerous situations. This results in reducing the number of serious crashes, and hence, improving road safety [Golias et al., 2002].

ADAS software for autonomous vehicles have to be tested and validated to ensure vehicles' safety and reliability. Many automotive companies take on-road test initiatives to drive their fleets of autonomous vehicles on real roads. However, testing ADAS with real hardware and in the real environment (e.g., by making a person or an animal cross a road while a car is approaching) is obviously dangerous, time-consuming, costly and to a great extent infeasible. It is further impractical to perform a full-fledged on-road vehicle-level testing after every change to self-driving software systems. To ensure safety of self-driving technologies, vehicle-level testing alone is neither enough nor practical. Therefore, it needs to be complemented by testing methods performed on computer software simulators [Belbachir et al., 2012, Koopman and Wagner, 2016].

Simulation, i.e., design time testing of system models, is arguably the most practical and effective way of testing software systems used for autonomous driving. Rich simulation environments are able to replicate a wide range of virtual traffic and road environments. This includes simulating various weather conditions, road types and topologies, intersections, infrastructures, vehicle types and pedestrians. Further, such platforms are able to simulate sensor technologies such as radar, camera and GPS. Simulation platforms enable engineers to execute scenarios describing different ways in which pedestrians interact with the road traffic or vehicles interact with one another, and allow them to run a much larger number of test scenarios compared to vehicle-level testing without being limited by conditions enforced during on-road testing.

In this dissertation, we focus on the problem of design time testing of ADAS in a simulated environment. This dissertation presents a set of approaches based on meta-heuristic search and machine learning techniques to automate ADAS testing. The work presented in this dissertation has been done in collaboration with IEE [IEE, 2019], a leading supplier in automotive sensing systems enhancing safety and comfort in vehicles produced by major car manufacturers worldwide.

In this dissertation, we use the PreScan simulator [TASS-International, 2019] to test ADAS. PreScan is a widely-used, commercial ADAS simulator in the automotive sector and has been used by IEE to test ADAS at early development stages. Figure 1.1 shows a snapshot of PreScan. The physics-based simulation function of PreScan is enabled via a network of connected Matlab/Simulink models [Matlab, 2019] implementing dynamic behavior of vehicles, pedestrians and sensors. To test ADAS, the software under test (SUT) should be also developed in Simulink and should be integrated into PreScan using inter-block connections between the Simulink models of the car and the SUT.

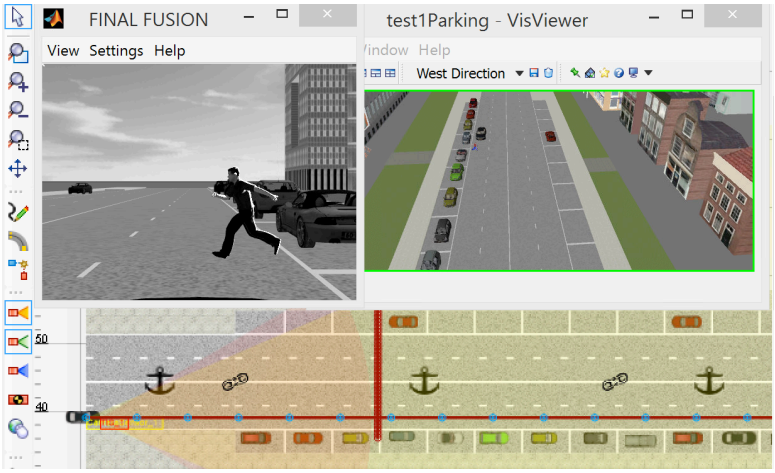

Figure 1.1. A snapshot of the simulation platform used to test ADAS.

Each ADAS is usually tested separately. However, autonomous cars are typically built as a composition of features (i.e., individual ADAS) that are independent units of functionality. Features tend to interact and impact one another's behavior in unknown ways. A *feature interaction* is a situation where one feature impacts the behavior of another feature [Jackson and Zave, 1998, Calder et al., 2003, Braithwaite and Atlee, 1994]. Some feature interactions are desirable, and some may result in violations of system safety requirements and are therefore undesired. Detecting undesired feature interactions (i.e., feature interactions leading to failures) is challenging. Once feature interactions are detected, engineers develop algorithms to resolve potential conflicts between features. This requires developing complex rules that determine what feature output should be prioritized at each situation based on the environment factors as well as other conditions. In this dissertation, we focus on testing individual ADAS as well as testing self-driving systems containing several ADAS.

1.2 Challenges

Testing ADAS using existing simulation platforms, is limited to executing the ADAS models for a small number of simulation scenarios hand-picked by engineers, and manually inspecting the results of individual simulations. Manual test generation is expensive and time-consuming, and further, manually picked test scenarios are unlikely to uncover faults that the engineers are not aware of a priori.

Our primary goal in this thesis is to develop automated test generation techniques for testing ADAS and to develop an automated repairing technique for feature interaction failures in autonomous cars. To achieve this goal, we should address the following challenges:

- Existing simulation tools lack the intelligence and automation necessary to guide the simulation scenarios in a direction such that they are likely uncover faulty behaviors.
- Executing simulation scenarios is computationally expensive. Hence, given a limited time budget for testing, only a small fraction of system behaviors can be simulated and explored.
- The space of test input scenarios is complex and multidimensional. Engineers require techniques that allow them to explore complex test input spaces and to identify critical test scenarios (i.e., failure revealing test scenarios).
- Concerning the problem of feature interactions, 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 conditions, road traffic and sidewalk objects. Without effective and automated assistance, engineers cannot detect undesired feature interactions within the search space of all possible interactions and cannot assess the impact of complex environmental factors on feature interactions.
- Developing rules to resolve feature interactions is a difficult task, and requires extensive domain expertise and a thorough analysis of system requirements. There is no guarantee that the set of system requirements or the set of conflict resolution rules is complete. Further, the rules or their implementation can be erroneous. The decision rules are too complex to be manually repaired. Engineers require techniques that automatically repair the developed rules in order to avoid conflicts between feature outputs.

1.3 Research Contributions

In this dissertation, we address the challenges of testing ADAS. Specifically, we propose the following contributions:

1. A testing approach for ADAS that combines multi-objective search with surrogate models developed based on neural networks. Our approach uses multi-objective search to guide testing towards the most critical behaviors of ADAS. Surrogate modeling enables our testing approach to explore a larger part of the input search space with less computational resources. This contri-

bution has been published in a conference paper [Ben Abdessalem et al., 2016] and is presented in Chapter 3.

2. An automated testing algorithm that builds on learnable evolutionary algorithms [Michalski, 2000, Wojtusiak and Michalski, 2004]. These algorithms rely on machine learning or a combination of machine learning and Darwinian genetic operators to guide the generation of new solutions (test scenarios in our context). Our approach combines multi-objective population-based search algorithms and decision tree classification models to achieve the following goals: First, classification models guide the search-based generation of tests faster towards critical test scenarios (i.e., test scenarios leading to failures). Second, search algorithms refine classification models so that the models can accurately characterize critical regions (i.e., the regions of a test input space that are likely to contain most critical test scenarios). Using classification models helps explore the complex input space of ADAS by focusing the search on the most critical parts of the space. This contribution has been published in a conference paper [Ben Abdessalem et al., 2018a] and is presented in Chapter 4.
3. An automated technique that detects feature interaction failures in self-driving systems. We cast the problem of detecting undesired feature interactions 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 [Panichella et al., 2018] to reduce the time required to compute fitness values. This contribution has been published in a conference paper [Ben Abdessalem et al., 2018b] and is presented in Chapter 5.
4. An automated technique that localizes and repairs feature interaction failures in self-driving systems. We develop a new search-based repairing algorithm, called RUFI, that localizes faults and mutates the faulty program (i.e., decision rules) to generate patches. We propose mutation operators that are specifically designed to address the specificities of our problem, and hence, increase the likelihood of generating correct patches. Our approach can repair programs with multiple faults. This contribution is presented in Chapter 6.
5. We have evaluated all of our proposed techniques by applying them into real industrial case studies from our industrial partner, IEE.

1.4 Dissertation Outline

Chapter 2 provides some foundational background on meta-heuristic search algorithms, supervised learning techniques, and existing program repair approaches.

Chapter 3 describes our search-based approach for testing ADAS. The testing approach uses surrogate models to reduce the execution time of the search algorithm.

Chapter 4 presents a testing approach that helps in exploring the complex input space of ADAS and identifying critical test scenarios.

Chapter 5 introduces our approach for detecting feature interaction failures in self-driving systems.

Chapter 6 describes our automated repair algorithm for the decision rules of self-driving systems.

Chapter 7 discusses related work.

Chapter 8 summarizes the thesis contributions and discusses perspectives on future work.

Chapter 2

Background

This chapter presents several background concepts, which are used throughout this thesis. This chapter is organized as follows: Section 2.1 introduces meta-heuristic search and describes the search algorithms used in our work. Section 2.2 introduces supervised learning techniques that are used in this dissertation. Section 2.3 presents the state-of-the-art automated repair techniques.

2.1 Meta-Heuristic Search

Meta-heuristic search is a randomized optimization method that aims to find optimal (or near optimal) solutions to hard problems [Luke, 2013]. It has been applied to a wide range of problems, where no analytic or numerical approximation technique can be used and brute force search is inapplicable due to the large size of the input space.

Meta-heuristics algorithms can be classified into *Single-state search* and *Population-based search*. Single-state meta-heuristic search algorithms keep only one single candidate solution in the solution set. The most naive single-state meta-heuristic search algorithm is Random Search (RS). The RS algorithm generates solutions randomly and evaluates them against the search criteria [Hamlet, 2002]. The generation process can continue until the time budget expires, and usually returns only the best solution found. Population-based search algorithms, on the other hand, keep multiple candidate solutions in the solution set, called the population. Genetic Algorithms (GAs) [Goldberg, 1989] are examples of population-based algorithms. GAs rely on four basic features: (1) *population* of individuals, (2) *selection* according to fitness, (3) *crossover* to produce new offsprings, and (4) *mutation* of new offsprings. Each individual is referred to as *chromosome*, and represents a candidate solution for a given problem. Chromosomes consist of a set of *genes*, where each gene encodes a value in the solution. GA starts by randomly generating a population of chromosomes, and evaluating their fitness. Then, the population is evolved toward better solutions through subsequent iterations. Specifically, at each iteration, a pair of chromosomes is randomly selected with increasing probability according to fitness. Then, the pair is crossed over to create two individuals (offspring), whose genes are then ran-

domly mutated. Finally, the offspring replaces two chromosomes, chosen with decreasing probability according to the fitness.

Many variants of the standard GA have been proposed to solve single-objective, multi-objective, or many-objective problems. Single-objective optimization problems have typically one solution (or multiple solutions with the same optimal fitness value). Multi-objective and many-objective optimization problems involve multiple and conflicting objective functions to be optimized at the same time. Finding optimal solutions for problems with multiple objectives involves analyzing trade-offs. Multi-objective optimization involves two or three objectives, while many-objective optimization refers to a class of optimization problems that have more than three objectives. Below, we describe the population-based search algorithms that we use in our work.

2.1.1 Non-dominated Sorting Genetic Algorithm version 2 (NSGAII)

Non-dominated Sorting Genetic Algorithm version 2 (NSGAII) [Deb et al., 2002, Luke, 2013] is a multi-objective search optimization algorithm, which has been previously applied to several software engineering problems. The NSGAII algorithm generates a set of solutions forming a *Pareto nondominated front* [Deb et al., 2002, Luke, 2013]. A dominance relation over solutions is defined as follows: A solution x dominates another solution y if x is not worse than y in all fitness values, and x is strictly better than y in at least one fitness value. The output of NSGAII is a non-dominating (equally viable) set of solutions, representing best-found trade-offs among fitness functions.

Algorithm 1: NSGAII Algorithm

```
   Input: m: Population and archive size                          // |A| = |P| = m
   Input: g: Maximum number of search iterations
   Result: BestSolution: The best solutions found in g iterations
1  begin
2      A ⟵ ∅                                                      // Empty archive
3      P ⟵ {p_1, ..., p_m}                    // Initial population (randomly selected)
4      for g iterations do
5          ComputeFitness(P)  // For all p∈P, fitness values F_1(p),...F_k(p) are computed
6          Q ⟵ P∪A                                                // |Q| = 2m
7          rank ⟵ ComputeRanks(Q)
8          A ⟵ ∅
9          while |A| < m do
10             p ⟵ BestRanked(Q, rank)
11             A ⟵ A∪{p}
12         BestSolution ⟵ A
13         P = Breed(A)      // breeding a new population P from the parent archive A
14     return BestSolutionFound
```

Algorithm 1 illustrates the NSGAII algorithm. The algorithm works as follows: Initially, a random population P is generated (Line 3). After computing fitness functions $F_1, \ldots, F_k$ for each individual in P (Line 5), the individuals in P as well as those in the archive A from the previous iteration are sorted based on the non-domination relation (Line 7). In particular, a partial order relation *rank* is computed to sort elements in $Q = P \cup A$ based on the fitness functions $F_1, \ldots, F_k$. Assuming that the goal of

optimization is to minimize the fitness functions F_1 to F_k, the partial order $rank \subseteq Q \times Q$ is defined as follows:

$$\forall p, p' \in Q \cdot rank(p, p') \quad \Leftrightarrow \quad \forall i \in \{1, \ldots, k\} \cdot F_i(p) \leq F_i(p') \wedge \exists i \in \{1, \ldots, k\} \cdot F_i(p) < F_i(p')$$

Specifically, $rank(p, p')$ if and only if p dominates p' at least in one fitness value, i.e., p is not worse (higher) than p' in all the fitness values and p is strictly better (less) than p' in at least one fitness value. Note that it might happen for a given pair of individuals that neither of them dominates the other. For these individuals, we use the notion of crowding distance [Deb et al., 2002], denoted by cd, to be able to partially rank them. We write $non\text{-}dominating(p, p')$ when p and p' are non-dominating, and say that $non\text{-}dominating(p, p')$ holds iff:

$$\forall i \in \{1, \ldots, k\} \cdot F_i(p) = F_i(p') \vee \exists i, i' \in \{1, \ldots, k\} \cdot F_i(p) < F_i(p') \wedge F_{i'}(p) > F_{i'}(p')$$

Let $R \subseteq Q$ be such that for all $p, p' \in R$, we have $non\text{-}dominating(p, p')$. For the elements in R, a crowding distance function cd is defined that assigns a value to $p \in R$ based on the distance between p and other $p' \in R$. The definition of the relation $rank \subseteq Q \times Q$ is then extended as follows: For every $p, p' \in Q$, we have $rank(p, p')$ iff:

$$(\forall i \in \{1, \ldots, k\} \cdot F_i(p) \leq F_i(p') \wedge \exists i \in \{1, \ldots, k\} \cdot F_i(p) < F_i(p')) \vee (non\text{-}dominating(p, p') \wedge cd(p') < cd(p))$$

Having computed the *rank* partial order, the NSGAII algorithm then creates a new archive A of the best solutions found so far by selecting the best individuals from Q based on *rank* (Lines 9-11). Note that $BestRanked(Q, rank)$ returns an element $p \in Q$ such that no other $p' \in Q$ dominates p, i.e., $\neg rank(p', p)$ for every $p' \in Q \setminus \{p\}$. When there are more than one individual that are not dominated by any element in Q, $BestRanked(Q, rank)$ returns the one that appears first in the lexicographic ordering.

After computing the archive A, the algorithm breeds a new children population P from the parent archive A by calling the *Breed* procedure (Line 13). This is done by selecting $m/2$ individuals as parents from A using the tournament selection technique [Luke, 2013], and then creating m offsprings from the $m/2$ selected parents using the *crossover* and *mutation* genetic operators. The NSGAII algorithm uses the single-point crossover and bitwise mutation for binary-coded GA and the simulated binary crossover (SBX) operator and polynomial mutation [Deb and Agrawal, 1995] for real-coded GA. The algorithm terminates by returning the best solutions found within g generations.

2.1.2 Many-Objective Sorting Algorithm (MOSA)

Many-Objective Sorting Algorithm (MOSA) [Panichella et al., 2015] is a many-objective search optimization algorithm. MOSA generates test cases that separately covers individual test objectives (e.g., branches) rather than finding solutions capturing well-distributed and diverse trade-offs among the search objectives. MOSA is an extension of NSGAII that uses a preference sorting criterion to reward the best tests for each uncovered test objective, regardless of their dominance relation with other tests

in the population. MOSA uses an archive to store the tests that cover new test objectives, which aims to avoid losing the best individuals for a given test objective after each iteration.

The MOSA algorithm works as follows: similar to NSGAII, MOSA starts with an initial set of random solutions. Then, the offsprings are created by applying crossover and mutation. To select the best individuals from the combined set of parents and offsprings, MOSA replaces the *rank* partial order used in NSGAII with a new ranking algorithm based on a preference criterion. MOSA imposes an order of preference among non-dominated test cases. To do so, MOSA rewards test cases that cover at least one objective over those that yield a low value on several objectives without covering any. In particular, test cases are ranked as follows: First, MOSA determines the test cases with the lowest objective function for each uncovered test objective. All these test cases are assigned rank 0 (i.e., they are inserted into the first non-dominated front). Second, the remaining test cases are ranked according to the traditional non-dominated sorting algorithm used by NSGAII (i.e., the *rank* partial order defined for NSGAII). The preference criterion aims to focus the search effort on the test cases that are closer to one or more uncovered test objectives. After determining the order of preference of each test case, the archive that stores previously uncovered objectives is updated in order to yield the final test suite.

2.2 Supervised Learning techniques

Machine learning is a part of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from past experiences without being explicitly programmed [Alpaydin, 2010a]. Machine learning techniques can generally be divided into two broad categories, supervised and unsupervised. In supervised learning, the aim is to learn a mapping between a set of input objects and output variables from labeled training data. In contrast, in unsupervised learning, data is trained based on input data only.

In this dissertation, we use supervised learning techniques. Supervised learning techniques are divided into regression and classification techniques where the goal is to predict real-valued and categorical outputs, respectively. We use surrogate models (artificial neural networks) in Chapter 3, to predict simulation outputs within some confidence interval to bypass the execution of expensive simulations. We use classification models (decision trees) in Chapter 4, to guide the search-based generation of tests faster towards critical test scenarios (i.e., test scenarios leading to failures).

2.2.1 Surrogate models

Surrogate models are mathematical relations that aim to reduce computational cost by approximating high-fidelity but computationally expensive models of physical phenomena [Alpaydin, 2010a]. Common examples of surrogate models are artificial neural networks, support vector machines, bayesian networks, regression trees, and kriging models.

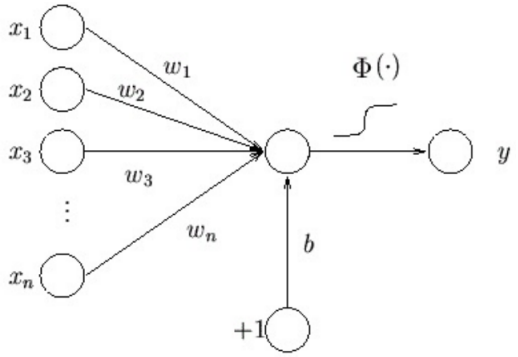

Figure 2.1. Simple artificial neuron. Here y is the output signal, ϕ is the activation function, n is the number of connections to the perceptron, w_i is the weight associated with the i^{th} connection, x_i is the value of the i^{th} connection, and b in the figure represents the threshold [Honkela, 2001].

In Chapter 3, we use neural networks to predict simulation outputs in order to reduce the execution time of our models. Artificial neural network models have been used as universal function approximators [Cybenko, 1989, Hornik, 1991]. Neural networks take their inspiration from the brain. The brain is composed of a very large number of cells, called *neurons*, that communicate via electrical signals. The interneuron connections are mediated by electrochemical junctions called synapses, which are located on branches of the cell referred to as dendrites. Each neuron receives thousands of connections from other neurons and is therefore constantly receiving a multitude of incoming signals, which eventually reach the cell body. In computer science, artificial neurons are processing units, which represent the smallest building block of an artificial neural network. An example of a simple artificial neuron is shown in Figure 2.1. Synapses are modelled by a single number or weight so that each input is multiplied by a weight before being sent to the equivalent of the cell body. The weighted signals are summed together by simple arithmetic addition to supply a node activation. If the combined input exceeds a threshold, it will activate and send an output (conventionally "1"), otherwise it outputs zero. The output it sends is determined by the activation function (ϕ in Figure 2.1).

Neural networks consist of a number of neurons. Figure 2.2 shows an example of neural network. Each node is represented by a circle. All connections have implicit weights. The nodes are organized in a layered structure. The first layer is the input layer followed by one or more hidden layers. The last layer is the output layer. Each signal emanates from an input and passes via at least two nodes before reaching an output beyond which it is no longer transformed.

Building a surrogate model by training neural networks requires a dataset that contains input values and known output values. The dataset is divided into training and test sets. The training dataset is used to modify the weights of the neural network using an error minimization objective function. The back-propagation algorithm is usually used to train the network. Using the back-propagation algorithm, the errors in the output layer are propagated backwards to the preceding hidden layers, and their weights are adjusted to reduce the error. The objective function used for training the neural network and updating its weights is the Mean-squared error (MSE), which is the most commonly used measure. The test dataset is used to evaluate the accuracy of the surrogate model.

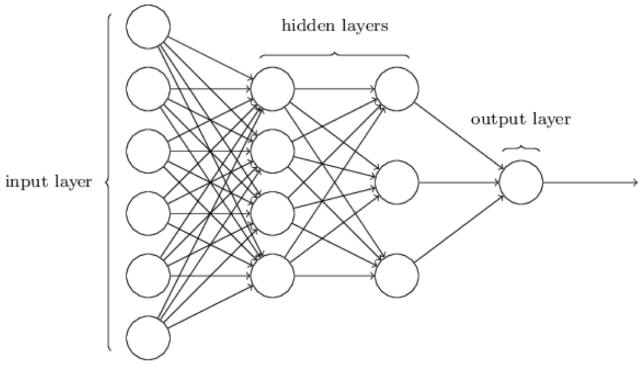

Figure 2.2. Simple example of neural network.

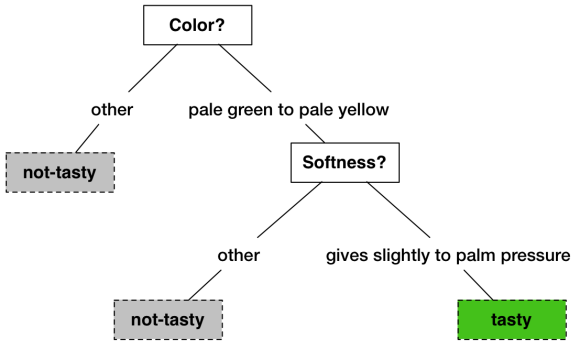

Figure 2.3. An example of decision tree for distinguishing papayas.

2.2.2 Decision Trees

A decision tree is a hierarchical tree structure implementing the divide-and-conquer strategy on the input objects [Alpaydin, 2010a]. Decision trees can be used for both regression and classification. In this dissertation, we use classification decision trees [Witten et al., 2011]. Decision trees classify instances by sorting them based on attributes (feature values). Each (non-leaf) node in a decision tree represents an attribute in an instance to be classified, and each branch represents a value that the node can assume (i.e., indicating the decision or prediction result). Decision trees classify an unknown instance by sorting it down the tree according to the values of the attribute tested in successive nodes, and when a leaf is reached the instance is classified according to the class assigned to the leaf. Decision trees generate classification rules [Hall et al., 2011], where a path from the root node of the tree to any leaf is considered as a rule. These classification rules are easy to understand.

An example of a decision tree for checking if a given papaya is tasty or not is shown in Figure 2.3 [Shalev-Shwartz and Ben-David, 2014]. This decision tree is built based on a training set that uses as attributes the color of the papaya (ranging from dark green, through orange and red to dark brown) and the softness of the papaya (ranging from rock hard to mushy). In this example, the training set is a sample of papayas that are examined for color and softness and then tasted to found out whether they were tasty or not.

To check whether a given papaya is tasty or not, the decision tree first checks the color of the papaya. If the color is not in the range from pale green to pale yellow, the tree will immediately predict

that the papaya is not tasty. Otherwise, the second step is to check the softness of the papaya. The decision tree predicts the papaya is tasty when the papaya gives slightly to palm pressure. Otherwise, the prediction result is "not-tasty". The resulting classifier is very simple to understand and interpret.

2.3 Program repair

The key idea of program repair techniques is to try to automatically repair software systems by localizing faults and modifying the software code to produce patches [Monperrus, 2018]. An overview of a classical programs repair is shown in Figure 2.4. Program repair takes as inputs a valid test suite and a faulty program. The test suite includes both passing and failing test cases. Program repair starts by locating the parts of the code that is responsible for the faults. This step is known as Fault Localization. Then, program repair tries to find patches for the faults to repair the program. This process is repeated until the patch program is found.

A well-known Fault Localization approach, used by classical program repair, is statistical debugging that uses coverage criteria as a heuristic to locate faults [Renieres and Reiss, 2003]. The idea is to execute all the given test cases and for each program element (i.e., statement s) in the code, keep track of how many failing and passing test cases execute them. Statistical debugging uses ranking formulas to compute the suspiciousness scores of each statement s. A well-known statistical ranking formula used for fault localization for source code is Tarantula [Jones et al., 2002]. Tarantula defines the suspiciousness score of s, denoted by $Score^{Ta}(s)$, as follows:

$$Score^{Ta}(s) = \frac{\frac{\#failed(s)}{\#failed}}{\frac{\#passed(s)}{\#passed} + \frac{\#failed(s)}{\#failed}} \tag{2.1}$$

such that, $\#passed(s)$ and $\#failed(s)$ are respectively the number of passing and failing test cases that execute s, and $\#passed$ and $\#failed$ represent the total number of passing and failing test cases, respectively. Using the suspiciousness score, statistical debugging, then, derives a statistical fault ranking, specifying an ordered list of statements likely to be faulty. Traditional programs repair consider such ranking to identify the fault, then tries to automatically identify patches for that fault.

Classical programs repair use a fitness function to evaluate each patch. The fitness function of a program p is based on the number of failing and passing test cases. For each faulty program p, the fitness function is defined as follows:

$$Fit(p) = W_{Neg} \times N_{Failed} + W_{Pos} \times N_{Passed} \tag{2.2}$$

where N_{Failed} is the number of failed test cases and N_{Passed} is the number of passed test cases. Each passed test is weighted by the global parameter W_{Pos}; each failed test is weighted by the global parameter W_{Neg}. The lower the value of the fitness function, the better the patch.

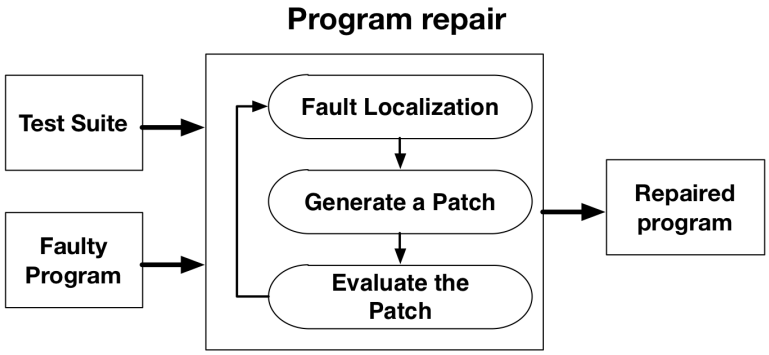

Figure 2.4. Overview of a classical program repair.

Traditional automated program repair techniques use Genetic Programming (GP) [Koza, 1992] to maintain a population of individual patches and uses a single fitness function to evaluate each patch. The pseudo-code of classical programs repair is given in Algorithm 2.

Algorithm 2: Classical Program Repair Algorithm

Input: m: Population size
Input: TS: Test suite
Result: Pop: Best patches found within the search time budget

```
1 begin
2     Pop ⟵ {p_1, ..., p_m}                    // Initial population (randomly selected)
3     ComputeFitness(Pop)
4     while not(Stop condition) do
5         O ⟵ ∅
6         O ⟵ LocalizeFaultAndGeneratePatch(Pop, TS)
7         ComputeFitness(O)
8         Pop ⟵ SelectBestPatches(Pop ∪ O)
9     return Pop
```

The algorithm works as follows: Initially, a random population Pop is generated (line 2). The algorithm then computes the fitness function for every individual $p \in Pop$ (line 3). Each iteration of the algorithm consists of the following steps: First, the algorithm generates m offsprings, denoted by O, by localizing the fault and applying mutation on each element $p \in Pop$ (line 6). Second, the offspring population is evaluated to compute the fitness function of every individual in O (line 7). Third, the individuals in Pop, as well as the individuals in O, are sorted based on the fitness function. Then, the algorithm updates Pop by selecting the best m individuals with the lowest fitness function (line 8). The algorithm terminates by returning the best solutions found within the search time budget. The best solutions contain the program versions with the least number of failing test cases.

Chapter 3

Testing Advanced Driver Assistance Systems

In this chapter, we provide a testing approach for ADAS by combining multi-objective search with surrogate models developed based on neural networks. We use multi-objective search to guide testing towards the most critical behaviors of ADAS. Given that executing simulation scenarios is computationally expensive, we use surrogate modeling to enable our testing approach to explore a larger part of the input search space within limited computational resources.

The combination of multi-objective search with surrogate modeling proposed in this chapter is not tied to our particular search-based testing algorithm and is applicable to any multi-objective search algorithm that computes a set of *Pareto optimal* solutions [Luke, 2013, Ferrucci et al., 2013]. A solution is called Pareto optimal if none of the fitness functions used by the search can be improved in value without degrading some of the other fitness values [Zitzler et al., 2000]. In our work, we use optimistic and pessimistic fitness function predictions computed based on surrogate models and a given confidence level to rank Pareto fronts during search. We identify and prune from the search space the candidate solutions that have a low probability to be selected in the best ranked Pareto front. We show that when actual fitness values are not better than their respective optimistic predictions, the search algorithm with surrogate modeling behaves the same as the original search algorithm without surrogate modeling. Specifically, under this condition and provided with the same set of candidate solutions at each iteration, search with and without surrogate modeling select the same solutions, but the search with surrogate modeling is likely to call less simulations per iteration than the search without surrogate modeling. Note that our proposed combination of multi-objective search with surrogate modeling is more accurate than existing alternatives [Magnier and Haghighat, 2010, Efstathiou et al., 2014] as it eventually uses the actual simulations instead of the predictions to compute Pareto optimal fronts.

This chapter highlights the following research contributions:

1. We formulate our testing approach as a *multi-objective search technique*. We use multi-objective search to obtain test scenarios that stress several critical aspects of the system and the environment at the same time.

2. We reduce the execution time of our search algorithm by proposing a *new* combination of multi-objective search with surrogate models built based on supervised learning techniques [Hall et al., 2011].
3. We evaluate our approach by applying it to an industrial ADAS.

Organization. This chapter is structured as follows. Section 3.1 motivates our work. Section 3.2 describes our case study system. Section 3.3 outlines our approach to developing surrogate models. Section 3.4 provides our multi-objective search algorithm that uses surrogate modeling. Section 3.5 tailors our search algorithm to the ADAS. Section 3.6 presents our empirical evaluation. Section 3.7 concludes this chapter.

3.1 Motivation and Challenges

We motivate our work using an ADAS case study. Our case study is a Pedestrian Detection Vision based (PeVi) system. Its main function is to improve the driver's view by providing proper warnings to the driver when pedestrians (people or animals) appear to be located in front of a vehicle in particular when the visibility is low due to poor weather conditions or due to low ambient light. PeVi consists of a CCD camera, and software components implementing image processing and object recognition algorithms as well as algorithms that determine when and which warning message should be shown to the driver. Since testing PeVi with real hardware and in the real environment is obviously dangerous, time-consuming, costly and infeasible, we use *physics-based simulation platforms*, in particular PreScan [TASS-International, 2019], to test PeVi. PeVi is developed in Simulink and is integrated into PreScan using inter-block connections between the Simulink models of the car and PeVi.

Provided with a physics-based simulation platform, test case execution is framed as executing models of the system under test and its environment. Existing simulation platforms are able to simulate ADAS behaviors with reasonable accuracy when provided with a set of input test data. However, they have two important limitations: The *first limitation* is that simulation platforms provide no guidance to engineers as to which test scenarios should be selected for simulation. Since test inputs are specified manually in current platforms, simulation is limited to a small number of scenarios hand-picked by engineers. Manual test generation is expensive and time-consuming, and further, manually picked test cases are unlikely to uncover faults that the engineers are not aware of a priori. Hence, it is important to augment these simulation platforms with some automated *test strategy* technique, i.e., a sampling strategy in the space of all possible simulation scenarios [Briand et al., 2016b], which attempts to build a sufficient level of confidence about correctness of the system under analysis through exercising only a small fraction of that space. The key question is how to choose an effective test strategy for testing ADAS.

We note that the space of all possible test scenarios for ADAS is very large. Traditional test coverage measures, which are common for small-scale, white-box testing, are infeasible and impractical for testing applications with large test spaces. Following the intuitive and common practice of system test engineers, we develop a *test strategy* that focuses on identifying high-risk test scenarios, that is

scenarios that are more likely to reveal critical failures [Briand et al., 2016b]. In particular, we rely on search techniques to devise a test strategy that focuses testing effort on an effective and minimal set of scenarios. The search is guided by heuristics that characterize high risk scenarios on which testing should focus. We develop meta-heuristics based on system requirements and critical environment conditions and system behaviors.

The *second limitation* is that physics-based simulations are computationally expensive because they often involve executing high-fidelity mathematical models capturing continuous dynamic behaviors of vehicles and their environment. To address this limitation, we rely on surrogate models [Jin, 2011] built based on machine learning techniques. Surrogate models are mathematical relations and aim to reduce computational cost by approximating high-fidelity but computationally expensive models of physical phenomena [Barton, 1994, Booker et al., 1999]. They are able to predict simulation outputs within some confidence interval allowing us, under certain conditions, to bypass the execution of expensive simulations.

3.2 The PeVi System

In this section, we provide some further background on the PeVi system, its important inputs and outputs and the fitness functions that we design to guide our test strategy to exercise PeVi's most critical behaviors.

PeVi Requirements. Based on the Pedestrian Detection Vision based (PeVi) specification, the cone-shaped space in front of a car that is scanned by the PeVi camera is divided into three *warning areas* illustrated in Figure 3.1. The size of the cone vertex, α, is a feature of the camera and is called *camera's field of view*. The warning areas are described as follows. (1) The *acute warning area (AWA)* is the red rectangle in Figure 3.1. (2) The *warning area (WA)* is the orange area in Figure 3.1. (3) The *cross warning area (CWA)* refers to the two yellow right-angled rectangles on the two sides of WA in Figure 3.1.

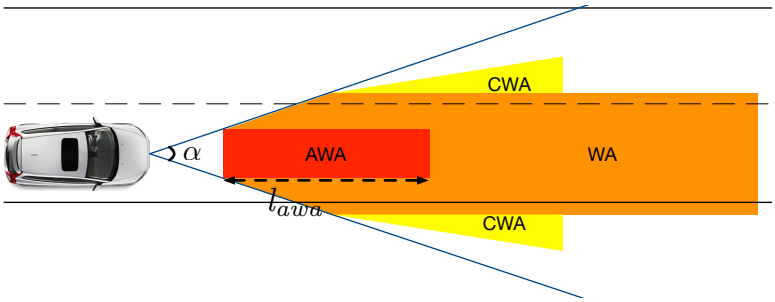

Figure 3.1. PeVi's warning areas.

The size and the position of the above three areas depend on the type of the car on which PeVi is deployed, the type of the camera used for PeVi, and the OEM (i.e., car maker) preferences. For

example, in our case study, the field of view α is set to 40°, and the length of AWA (l_{awa}) ranges between 60m to 168m depending on the car speed.

The main requirement of PeVi is stated as follows: R = "*The PeVi system shall generate a red, orange, or yellow alert when it detects an object in AWA, WA, and CWA warning areas, respectively. Further, PeVi shall fulfill this requirement under different weather conditions and while the car runs on different types of roads with different speeds*." The requirement R, although summing up the main function of PeVi, is still very broad. There are several scenarios where a pedestrian may end up being in one of the dangerous area in front of a car when crossing a road. To focus testing on the most high risk test scenarios among the numerous possibilities that requirement R characterizes, we identified the following specific situations after discussions with engineers at our partner company: *"It is more critical for PeVi to detect pedestrians in the warning areas when pedestrians are closer to the car and when the chance of collision is higher."*. We use these specific situations to define fitness functions for our multi-objective search algorithm.

PeVi Input and Output. In general, PeVi's function is impacted by several physical phenomena and environment factors. For example, road friction or wind may affect vehicle speed, which in turn, influences PeVi's behavior. However, given that the testing budget both in terms of manual and computational effort is limited, we identified, through our discussions with the domain expert, the most essential elements impacting the PeVi system. We developed a *domain model* to precisely capture these elements. This domain model essentially specifies a restricted simulation environment that is sufficient for testing PeVi. Further, this domain model characterizes the PeVi inputs and the outputs generated after simulating PeVi.

The domain model is shown in Figure 3.2. Based on this model a test scenario for PeVi contains the following input: (1) the value of the scene light intensity; (2) the weather condition that can be normal, foggy, rainy, or snowy; (3) The road type that can be straight, curved, or ramped; (4) the roadside objects, namely, trees and cars parked next to the road; (5) the camera's field of view; (6) the initial speed of the vehicle; and (7) the initial position, the orientation (θ) and the speed of the pedestrian. All these input elements except for the vehicle and the pedestrian properties are static (i.e., immobile) during the execution of a test scenario. The vehicle and the pedestrian (human or animal) are dynamic (i.e., mobile).

Our domain model makes some simplifying assumptions about PeVi's test scenarios. For example, we assume that the test scenarios contain only one pedestrian and one vehicle, and the vehicle and the pedestrian speeds are constant. These assumptions are meant to reduce the complexity of test scenarios and were suggested by the domain expert. However, we note that our search-based test generation approach is general and is not restricted by these assumptions.

Each of the input elements in Figure 3.2 (i.e., the elements related by a composition relation to the test scenario element in Figure 3.2) impacts PeVi's behavior. For example, the weather condition and the scene light intensity impact the quality of images and the accuracy of PeVi in detecting pedestrians. The camera field of view (α) and the road shape (e.g., straight, curved and ramped)

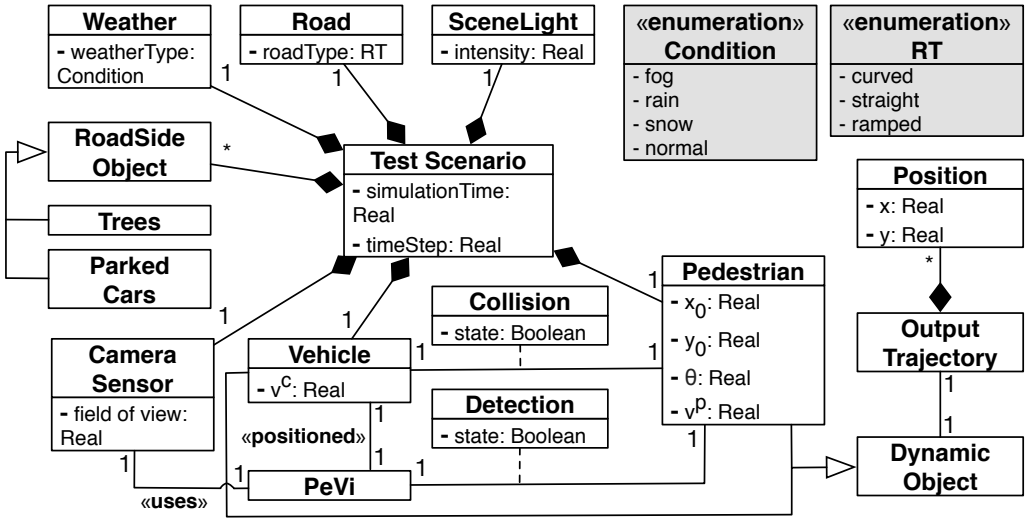

Figure 3.2. A fragment of the PeVi domain model.

impact the topological positions of the three warning areas, which in turn, impact the pedestrian detection function of PeVi. Roadside objects may block the camera's field of view, hence leaving PeVi with little time to detect a pedestrian and to react with a proper warning message. Finally, PeVi's function should be tested for various scenarios by varying the speed of the vehicle and the pedestrian, and the position and orientation of the pedestrian.

Each test scenario is associated with a simulation time denoted by T and a time step denoted by Δt. The simulation time T indicates the time we let a test scenario run. The simulation interval $[0..T]$ is divided into small equal time steps of size Δt. In order to execute a test scenario, we need to provide the simulator with the values of the input elements described in Figure 3.2 as well as T and Δt.

In this chapter, we limit the PeVi test input to the properties of the vehicle and the pedestrian. These properties can be directly manipulated by dynamically modifying the Simulink models implementing the vehicle and the pedestrian. We manipulate other properties (e.g., road types and weather conditions) manually. In Chapter 4, we will vary properties of all the input elements to generate test cases. We define the PeVi's test input as a vector $(v^c, x_0, y_0, \theta, v^p)$ where v^c is the car speed, x_0 and y_0 specify the position of the pedestrian, θ is the orientation of the pedestrian, and v^p is the speed of the pedestrian. We assume that the initial position of the vehicle (x_0^c, y_0^c) is fixed in test scenarios (i.e., $x_0^c = 0m$ and $y_0^c = 50m$). We denote the value ranges for v^c, x_0, y_0, θ and v^p, respectively, by R_{v^c}, R_{x_0}, R_{y_0}, R_θ and R_{v^p} such that R_{v^c}=[3.5km/h, 90km/h], R_{v^p}=[3.5km/h, 18km/h], R_θ=[40°, 160°], R_{x_0}=[x_0^c+20m, x_0^c+85m], and R_{y_0}=[y_0^c-15m,y_0^c-2m]. Note that variables v^c, x_0, y_0, θ and v^p are all of type float.

Having provided the input, PreScan simulates the behavior of PeVi, the vehicle and the pedestrian, and generates the following output elements: (1) output trajectory vectors: each dynamic object (i.e., the vehicle and the pedestrian) is associated with an output trajectory vector that stores the position of that object at each individual time step. The size of the trajectory vectors, denoted by k, is equal to the number of time steps within the simulation time, i.e., $k \approx \frac{T}{\Delta t}$. We denote the trajectory output of the pedestrian by the following two functions: $X^p, Y^p : \{0, \Delta t, 2 \cdot \Delta t, \ldots, k \cdot \Delta t\} \rightarrow \mathbb{R}$. We write $X^p(t)$

and $Y^p(t)$ to denote the pedestrian position on x- and y-axes at time t, respectively. Similarly, we define functions $X^c, Y^c : \{0, \Delta t, 2 \cdot \Delta t, \ldots, k \cdot \Delta t\} \rightarrow \mathbb{R}$ for the trajectory output of the car; (2) collision: this is a Boolean property indicating whether there has been a collision between the vehicle and the pedestrian during the simulation; and (3) detection: this is a Boolean property generated by PeVi indicating whether PeVi has been able to detect the pedestrian or not.

Fitness Functions. Our goal is to define fitness functions that can guide the search into generating test scenarios that break or are close to breaking the requirement R mentioned earlier. For example, PeVi test scenarios should exercise behaviors during which a pedestrian appears in front of a car in such a way that the possibility of a collision is high and the chance of detecting the pedestrian is low because the pedestrian is very close to the car or because the camera's field of view is blocked. Based on our discussions with the domain expert, we identify the following three fitness functions.

1. Minimum distance between the car and the pedestrian. The first function, denoted by $D^{min}(p/car)$, computes the minimum distance between the car and the pedestrian during the simulation time. We denote the Euclidean distance between the pedestrian and the car at time t by $D(p/car)(t)$. The function $D^{min}(p/car)$ is then defined as follows:

$$D^{min}(p/car) = Min\{D(p/car)(t)\}_{0 \leq t \leq T} \tag{3.1}$$

The test scenarios during which the pedestrian gets closer to the car are more critical. Hence, our search strategy attempts to minimize the fitness function $D^{min}(p/car)$.

2. Minimum distance between the pedestrian and AWA. The second function, denoted by $D^{min}(p/awa)$, computes the minimum distance between the pedestrian and AWA during the simulation time. We denote the distance between the pedestrian and AWA at time t by $D(p/awa)(t)$. This function depends on the shape of the road, the orientation of the pedestrian, and her position at time t. The value of $D(p/awa)$ for time steps in which the pedestrian is inside AWA is zero. The function $D^{min}(p/awa)$ is then defined as follows:

$$D^{min}(p/awa) = Min\{D(p/awa)(t)\}_{0 \leq t \leq T} \tag{3.2}$$

The goal of our search strategy is to minimize the second fitness function as well. This is because in order to test PeVi's function for AWA, we need to generate scenarios during which the pedestrian crosses AWA or gets close to it. To test PeVi for the two other warning areas, WA and CWA, we modify $D^{min}(p/awa)$ to compute the distances between the pedestrian and WA and CWA, respectively.

3. Minimum time to collision. The third fitness function is referred to as the minimum Time To Collision (TTC) and is denoted by TTC^{min}. The time to collision at time t, $TTC(t)$, is the time required for the car to hit the pedestrian if both the car and the pedestrian continue at their speed at time t and do not change their paths. The function TTC^{min} is then defined as the minimum value of $TTC(t)$ when t ranges from 0 to T. TTC^{min} has proven to be an effective measure to estimate the collision risk and to identify critical traffic situations [van der Horst and Hogema, 1993]. We are interested to generate scenarios that yield a small TTC^{min} since these scenarios are more risky.

We note that combining fitness functions into one function and using single-objective search is less desired in this situation because: First, our three fitness functions are about different concepts (i.e., time and distance). Second, engineers are typically interested in exploring interactions among critical factors of the system and the environment. For example, they might be particularly interested to inspect if PeVi is able to detect pedestrians when they are located on the borders of the camera's field of view. For this purpose, in our work, a multi-objective search algorithm that produces several test cases that exercise different and equally critical interactions of the system and the environment is preferred to a single-objective search algorithm that generates a single test case.

3.3 Surrogate Models

We use surrogate models in our work to mitigate the computation cost of executing physics-based ADAS simulations. Specifically, in order to compute the three fitness functions described in Section 3.2, we have to execute expensive physics-based simulations. We create a surrogate model for each fitness function to predict the fitness values without running the actual simulations. Such surrogate models are often developed using machine learning techniques such as classification, regression or neural networks [Alpaydin, 2010b]. Given that we are dealing with real-valued functions, *regression* or *neural network* techniques are more suitable for our purpose because classification techniques are geared towards functions with categorical outputs. Many studies have shown that neural networks perform better than regression techniques in particular when the input space under analysis is large and when the relationship between inputs and outputs is complex [Nguyen and Cripps, 2001, Emadi and Mahfoud, 2011]. Hence, we use neural networks to build surrogate models. Neural networks can be used with supervised or unsupervised training algorithms [Haykin, 1998]. In our work, we are able to obtain output values for training input data by running simulations. Hence, we use neural networks in a supervised training mode.

Recall from Section 2.2.1 that neural networks consist of a number of *neurons* connected via weighted links. The training process aims to synthesize a network by learning the weights on links connecting the neurons. Learning is carried out in a number of iterations known as *epochs*. In this work, we consider the following well-known training algorithms to develop our surrogate models: Bayesian regularization backpropagation (BR) [MacKay, 1992], Levenberg-Marquardt (LM) [Hagan and Menhaj, 1994], and Scaled conjugate gradient backpropagation (SCG) [Møller, 1993].

Given a fitness function F, we build a surrogate model of F by training a neural network. To do so, we use a set of observations containing input values and known output values [Witten et al., 2011]. We divide the observation set into a *training* set and a *test* set. The training set is used to infer a predictive function $\hat{F}$. This is done by training a neural network of $\hat{F}$ such that $\hat{F}$ fits the training data as well as possible, i.e., for the points in the training set, the differences between the output of F and that of $\hat{F}$ are minimized. The test set is, then, used to evaluate the accuracy of the predictions produced by $\hat{F}$ when applied to points outside the training set. Training neural networks requires tuning a number of parameters, particularly the number of (hidden) layers, the number of neurons in each hidden layer and the number of epochs. Further, we need to choose among the three training

algorithms (i..e, BR, LM, and SCG). Finding the best values for these parameters and selecting the best performing algorithm in our case is addressed in our empirical evaluation (Section 3.6).

In addition to building function $\hat{F}$ to predict the values of a fitness function F, we develop an error function $\hat{F}_{\varepsilon}^{cl}$ that estimates the prediction error based on a given confidence level *cl*. The value of *cl* is a percentage value between 0 and 100. For example, let $\hat{F}_{\varepsilon}^{cl}$ be the error function computed for $\hat{F}$ with respect to $cl = 95$. This implies that with a probability of 95%, the actual value of $F(p)$ lies in the interval of $\hat{F}(p) \pm \hat{F}_{\varepsilon}^{cl}$. We compute $\hat{F}_{\varepsilon}^{cl}$ based on the distribution of prediction errors obtained based on the test sets.

3.4 Search with Surrogate Model

We cast the problem of test case generation for ADAS as a multi-objective search optimization problem [Luke, 2013]. Specifically, we identified three fitness functions in Section 3.2 to characterize critical behaviors of the PeVi system and its environment. The solutions to our problem are obtained by minimizing these three fitness functions using a multi-objective *Pareto optimal* approach [Luke, 2013, Ferrucci et al., 2013] that states that "A solution p is said to dominate another solution p', if p is not worse than p' in all fitness values, and p is strictly better than p' in at least one fitness value". The solutions on a Pareto optimal front are non-dominating, representing best-found test scenarios that stress the system under analysis with respect to the three identified fitness functions.

In our work, we rely on population-based and multi-objective search optimization algorithms [Coello et al., 2007, Deb, 2001]. In this class of algorithms, the dominance relation over chromosome populations is used to guide the search towards Pareto-optimal fronts. In our work, we choose the NSGAII [Deb et al., 2002, Luke, 2013] algorithm which has been applied to several application domains and has shown to be effective in particular when the number of objectives is small [Sayyad and Ammar, 2013]. NSGAII is described in Section 2.1.1. Algorithm 1 illustrates the NSGAII algorithm. NSGAII uses an archive A of the best solutions. At each iteration, NSGAII breeds a new children population P from the parent archive A by calling the *Breed* procedure. NSGAII runs the simulation for every new element to compute the fitness functions. Then, NSGAII uses a partial order relation *rank* to sort elements in $Q = P \cup A$ based on the fitness functions $F_1, \ldots, F_k$. The *rank* is based on the non-domination relation, discussed in Section 2.1.1. Having computed the *rank* partial order, the NSGAII algorithm then creates a new archive A of the best solutions found so far by selecting the best individuals from Q based on *rank*.

In this chapter, we change the NSGAII algorithm in Algorithm 1 to use, instead of the actual fitness values, the predicted fitness values obtained from surrogate models to compute the partial order *rank* and to select the best individuals A. We refer to our algorithm as NSGAII-SM. The main goal of NSGAII-SM is to speed up the search by selecting an archive of best individuals A from the set Q without the need to run costly simulations for every element in the newly bred children population P. Specifically, we bypass execution of the simulation for any individual $p \in P$, if we can conclude using predicted fitness values that p has a low probability to be included in A.

Recall that the surrogate model for any fitness function F comprises a prediction function $\hat{F}$ and an error function $\hat{F}_{\varepsilon}^{cl}$ that estimates the prediction errors of $\hat{F}$ within the confidence level cl. Since our optimization problem aims to minimize fitness values, for any individual p, we have a most optimistic fitness value $\hat{F}(p) - \hat{F}_{\varepsilon}^{cl}(p)$ and a most pessimistic fitness value $\hat{F}(p) + \hat{F}_{\varepsilon}^{cl}(p)$. The gap between these two values widens by increasing the confidence level cl, and decreases by lowering cl.

Our NSGAII-SM algorithm is shown in Algorithm 3. The algorithm computes predicted fitness values for every individual $p \in P$ (Line 5). The algorithm, further, uses a set *Predicted* to keep track of elements for which only predicted fitness values are known, i.e., the elements for which the actual simulation has not yet been executed. The set *Predicted* is initially set to P since for the elements in P, actual fitness values have not yet been computed. Then, the algorithm computes two partial order relations $rank^{-}$ and $rank^{+}$. The relation $rank^{-}$ is computed based on optimistic fitness values ($\hat{F}(p) - \hat{F}_{\varepsilon}^{cl}(p)$) for individuals in *Predicted* and actual fitness values (F) for other individuals. Dually, the relation $rank^{+}$ is computed based on pessimistic fitness values ($\hat{F}(p) + \hat{F}_{\varepsilon}^{cl}(p)$) for individuals in *Predicted* and actual fitness values (F) for other individuals.

Algorithm 3: NSGAII-SM Algorithm

Input: m: Population and archive size `// |A| = |P| = m`
Input: g: Maximum number of search iterations
Result: *BestSolution*: The best solutions found in g iterations

1 **begin**
2 $A \longleftarrow \emptyset$ `// archive`
3 $P \longleftarrow \{p_1, \ldots, p_m\}$ `// Initial population (randomly selected)`
4 **for** *g iterations* **do**
5 *PredictFitness*(P) `// For every` $p \in P$ `and every fitness function` F_i `s.t.` $i \in \{1, \ldots, k\}$, `compute` $\hat{F}_i(p) - \hat{F}_{i,\varepsilon}^{cl}(p)$ `and` $\hat{F}_i(p) + \hat{F}_{\varepsilon,i}^{cl}(p)$
6 *Predicted* $\longleftarrow P$
7 $Q \longleftarrow P \cup A$ `// |Q| = 2m`
8 $rank^{-}, rank^{+} \longleftarrow$ *ComputeRanks*(Q)
9 $A \longleftarrow \emptyset$
10 **while** $|A| < m$ **do**
11 $p \longleftarrow$ *BestRanked*($Q, rank^{-}$)
12 **while** $p \notin$ *Predicted* $\wedge\, |A| < m$ **do**
13 $A \longleftarrow A \cup \{p\}$
14 $Q \longleftarrow Q \setminus \{p\}$
15 $p \longleftarrow$ *BestRanked*($Q, rank^{-}$)
16 **if** $|A| = m$ **then**
17 **break**
18 $p \longleftarrow$ *BestRanked*(*Predicted*, $rank^{+}$)
19 *ComputeFitness*($\{p\}$) `// Run simulation and compute actual fitness values` F_1, `...,` F_k `for` p
20 *Predicted* $\longleftarrow$ *Predicted* $\setminus \{p\}$
21 $rank^{-}, rank^{+} \longleftarrow$ *ComputeRanks*(Q) `// re-rank the remaining elements in` Q `after computing the actual fitness values for` p
22 *BestSolution* $\longleftarrow A$
23 $P \longleftarrow$ *Breed*(A) `// breeding a new population from the parent archive`
24 **return** *BestSolutionFound*

Let *rank* be the partial order over Q computed based on the actual fitness values for every element in Q (assuming that the actual fitness values are known for elements in Q). Then, we show the follow lemma.

Lemma. Let $BestRanked(Q, rank^-) \notin Predicted$. Suppose for every $p \in Predicted$ and every fitness function $F_i \in \{F_1, \ldots, F_k\}$, we have $\hat{F}_i(p) - \hat{F}^{cl}_{i,\varepsilon}(p) \leq F_i(p)$. Then, $BestRanked(Q, rank^-) = BestRanked(Q, rank)$.

Proof. By our assumption, the actual fitness value for any element $p \in Predicted$ is higher than their optimistic fitness value $\hat{F}_i(p) - \hat{F}^{cl}_{i,\varepsilon}(p)$, which is the value used to create $rank^-$. Hence, none of the elements in *Predicted* could be ranked higher than $BestRanked(Q, rank^-)$ when we use the partial order *rank*.■

The above lemma states that assuming that actual fitness values are not better than the optimistic predictions and if $BestRanked(Q, rank^-) \notin Predicted$, then $BestRanked(Q, rank^-)$ is equal to the best ranked element computed in NSGAII where we do not use predictions.

Given the above lemma, in NSGAII-SM, we first add the elements of Q that are ranked best by $rank^-$ and are not in *Predicted* to the archive of best elements A (Lines 11-15). After that, if A is still short of elements (i.e., $|A| < m$), we compute actual fitness values for the predicted element that is ranked highest by $rank^+$, i.e., the partial order based on pessimistic fitness values of predicted elements (Lines 18-20). We then recompute $rank^-$ and $rank^+$ (Line 21), and continue until we select m best elements from Q into A. For all the elements in A, the actual fitness values are already computed (i.e., $A \cap Predicted = \emptyset$).

Upon termination of the while loop (Lines 10-21) in Algorithm 3, the set $Predicted \subseteq Q$ contains those elements that NSGAII-SM was able to discard without the need to compute the actual fitness values for them. Hence, at each iteration, the size of *Predicted* indicates the number of simulation calls that our algorithm has been able to save. This is in contrast to the original NSGAII algorithm (Algorithm 1) where at each iteration, simulation is called for m times. According to the above Lemma, if actual fitness values are not better than the optimistic predictions, then NSGAII and NSGAII-SM behave the same. That is, assuming that NSGAII and NSGAII-SM are provided with the same set P at each iteration, they select the same candidate solutions (set A), but NSGAII-SM is likely to perform less simulations per iteration than NSGAII. The probability of actual fitness values being better than the optimistic predictions depends on the confidence level cl. For example, for $cl = 95\%$, with a probability of 2.5%, the actual fitness values are better than their optimistic predictions, and hence, NSGAII-SM might select less optimal solutions compared to NSGAII given the same set P. In Section 3.6, we empirically compare the quality of the solutions generated by NSGAII-SM and NSGAII in particular by accounting for the randomness factor in generating P and by executing the two algorithms within a limited and realistic time budget.

3.5 Tailoring Search to PeVi

The algorithms NSGAII and NSGAII-SM described in Section 3.4 are generic. We tailor them to our search-based test generation problem by specifying the search input representation, the fitness functions, and the genetic operators.

Input representation. The input space of our search problem consists of vectors $(v^c, x_0, y_0, \theta, v^p)$. The variables v^c, x_0, y_0, θ and v^p and their ranges are defined in Section 3.2. Each value assignment to the vector $(v^c, x_0, y_0, \theta, v^p)$ represents a *chromosome*, and each value assignment to the variables of this vector represents a *gene*.

Fitness functions. We use the three fitness functions, $D^{min}(p/car)$, $D^{min}(p/awa)$ and TTC^{min}, defined in Section 3.2 for our search algorithm. A desired solution is expected to minimize these three fitness functions.

Genetic operators. The *Breed*() procedure in the NSGAII and NSGAII-SM algorithms is implemented based on the following operators:

- *Selection.* We use a binary tournament selection with replacement that has been used in the original implementation of NSGAII algorithm [Deb et al., 2002].

- *Crossover.* We use the Simulated Binary Crossover operator (SBX) [Beyer and Deb, 2001, Deb and Agrawal, 1995]. SBX creates two offsprings from two selected parent individuals. The difference between offsprings and parents is controlled by a distribution index (η): The offsprings are closer to the parents when η is large, while with a small η, the difference between offsprings and parents will be larger [Deb and Beyer, 2001]. In this chapter, we chose a high value for η (i.e., $\eta = 20$) based on the guidelines given in [Deb and Agrawal, 1995].

- *Mutation.* Mutation is applied after crossover to the genes of the children chromosomes with a certain probability (mutation rate). Given a gene x (i.e., any of the variables v^c, x_0, y_0, θ and v^p), our mutation operator shifts x by a value x' selected from a normal distribution with mean $\mu = 0$ and variance σ^2. To avoid invalid offsprings, if the result of a crossover or a mutation is greater than the maximum, it is set to the maximum. If the result is below the minimum, it is clamped to the minimum.

3.6 Evaluation

In this section, we investigate the following Research Questions (RQs) through our empirical evaluation applied to the PeVi case study.

RQ1. (Comparing Random Search, NSGAII and NSGAII-SM) *How do NSGAII, NSGAII-SM and random search perform compared to one another?* We start by comparing the time performance and the quality of solutions obtained by our test generation strategy when we use NSGAII and

NSGAII-SM. Our goal is to determine whether NSGAII-SM is able to generate results with higher quality than those obtained by NSGAII within the same time period. We then compare the algorithm that performs better, between NSGAII and NSGAII-SM, with a random test generation algorithm (the baseline of comparison typically adopted in SBSE research [Harman et al., 2012a]).

RQ2. (Usefulness) *Does our approach help identify test scenarios that are useful in practice?* This question investigates whether the test scenarios generated by our approach were useful for the domain experts and how they compared with the test scenarios that have been previously devised by manual testing based on domain expertise.

Metrics. We evaluate the prediction accuracy of surrogate models using the coefficient of determination (R^2) [Witten et al., 2011] that measures the predictive power of a surrogate model by identifying how well a test set fits the model. Specifically, R^2 measures the proportion of the total variance of F explained by $\hat{F}$ for the observations in the test set where F is a fitness function and $\hat{F}$ is its corresponding predictive function. The value of R^2 ranges between 0 and 1. The higher the value of R^2, the more accurate the surrogate model is.

To assess and compare the quality of Pareto fronts obtained by our alternative search algorithms, we use two well-known quality indicators [Knowles et al., 2006a], hypervolume (HV) and generational distance (GD). HV [Zitzler and Thiele, 1999a] measures the volume in the solution space that is covered by members of a non-dominated set of solutions. The larger the volume (i.e., the higher the value of HV), the better the results of the algorithm. GD [Van Veldhuizen and Lamont, 1998a] compares the pareto front solutions computed by an algorithm with an *optimal pareto front* (or *true pareto front*), i.e., the best non-dominated solutions that exist in a given space of solutions for a given problem. In particular, GD [Van Veldhuizen and Lamont, 1998a] is the average distance between each point in a computed Pareto front and the closest optimal pareto front solution to that point. A value of 0 for GD indicates that all the obtained solutions by a search algorithm are optimal. The lower GD, the better the results of the algorithm. Computing an optimal pareto front is usually not feasible. As suggested in the literature [Wang et al., 2016], instead, we use a reference pareto front that is a union of all the non-dominated solutions computed by our search algorithms (i.e., NSGAII, NSGAII-SM and random search). The HV and GD are selected from the combination and convergence quality indicator categories, respectively. As discussed in [Wang et al., 2016], to assess the quality of computed pareto fronts with respect to combination and convergence indicators, it is sufficient to choose only one indicator from each of these two categories.

Experiment Design. We implemented the NSGAII and NSGAII-SM algorithms and the neural network surrogate models in Matlab. In addition, we implemented a test generation strategy based on random search. Random search [Luke, 2013] and our NSGAII-based search algorithms require to interact with PreScan to execute simulations of the Simulink models of the pedestrian, the car and the PeVi system embedded into the car (see Section 3.2). The NSGAII-SM algorithm, in addition to calling the PreScan simulator, calls neural networks that are developed to serve as surrogate models. We ran all the experiments on a laptop with a 2.5 GHz CPU and 16GB of memory. Based on our experiments, each PeVi simulation (i.e., each call to PreScan), on average, takes 2 min with a min value

of 1.2 min and a max value of 3.4 min. The simulation time variations are due to the variations in the car and the pedestrian speeds and positions. Further, we may stop simulations before completion at a point where we can conclusively determine the fitness function values.

To answer our research questions, we designed and performed the following experiment. First, we identified the training algorithm and the configuration values that lead to the most accurate neural network-based surrogate models for the PeVi case study. To do so, for each of our three PeVi fitness functions, we compared 18 different neural network configurations. The comparison is based on a k-fold cross validation with $k = 5$ [Alippi and Roveri, 2010, Efron, 1983]. Specifically, we first selected (using adaptive random search [Luke, 2013]) 1000 observation points from the input search space of the PeVi system. Adaptive random search was used to maximize diversity in our training and test sets. It is an extension of the naive random search that attempts to maximize the Euclidean distance between the points selected in the input space. Recall from Section 3.2 that each PeVi input point is a vector $(v^c, x_0, y_0, \theta, v^p)$ selected from a five dimensional space. We simulated each point to obtain the actual values for each fitness function. In the experiment, we refer to fitness function $D^{min}(p/car)$ by F_1, to $D^{min}(p/awa)$ by F_2, and to TTC^{min} by F_3. The 1000 observation points are randomly partitioned into five disjoint subsets with 200 points in each. We then randomly selected four subsets to create a training set with 800 points, and the remaining subset is used as the test set. This process is repeated for five times so that for each 5-fold cross validation, the R^2 values are computed on a test set containing the entire 1000 points. To account for randomness, we repeated the 5-fold cross validation ten times.

To develop neural networks with high prediction accuracy, we considered three training algorithms, BR, LM and SCG, and we set different values to the following parameters: the number of hidden layers (*nl*), the number of neurons in each hidden layer (*nn*) and the number of epochs (*ne*). Specifically, we set $nl = 2$ as is common in the literature [Goyal and Goyal, 2012, Karsoliya, 2012]. There are various recommendations for setting *nn*. In particular, *nn* is recommended to be less than twice or equal to $\frac{2}{3}$ of the size of the input vector [Behera, 2014, Karsoliya, 2012], or to be a number between the input and the output size [Behera, 2014, Karsoliya, 2012]. We considered the values 3 and 4 for *nn*. In addition, we considered the value 100 for *nn* because in some cases the accuracy may improve when *nn* is set to values considerably larger than the input size [Sheela and Deepa, 2013]. Finally, we set *ne* to 10 and 100.

In total, we developed and trained 18 different neural network configurations. We computed R^2 for 10 different repetitions of 5-fold cross validations of the 18 neural network configurations related to our three fitness functions. We selected the following neural network configurations with highest predictive accuracy (highest R^2) for our three fitness functions: For F_1, we selected the configuration that was developed by the BR algorithm with *nn* = 100 and *ne* = 100 (R^2 = 0.99). For F_2, we selected the configuration that was developed by the BR algorithm with *nn* = 100 and *ne* = 100 (R^2 = 0.84), and for F_3, we selected the configuration that was developed by the LM algorithm with *nn* = 3 and *ne* = 100 (R^2 = 0.89). Note that R^2 = 0.84 indicates that 84% of the variance in the test set is explained by the predictive model. The high R^2 values of the selected configurations indicate their high predictive accuracy.

Having obtained the most accurate surrogate models to be used by the NSGAII-SM algorithm, we now discuss value selection for the search algorithms' parameters. Since our experiments, which involve running physics-based simulations, are very time-intensive, we were not able to systematically tune the search parameters (e.g., based on the guidelines provided in [Arcuri and Fraser, 2011a]). Instead, we selected the parameters based on some small-scale preliminary experimentations as well as existing experiences with multi-objective search algorithms. Specifically, we set the crossover rate to 0.9, the mutation rate to 0.5 and the population size to 10. Our choice for the crossover rate is within the suggested range of $[0.45, .., 0.95]$ [Bowman et al., 2010]. Our preliminary experimentations showed that more explorative search may lead to better results. Hence, we set the mutation rate to 0.5 which is higher than the suggest value of $1/l$ where l is the length of chromosome [Arcuri and Fraser, 2011a]. Finally, we chose a relatively small population size to allow for more search iterations (generations) within a fixed amount of time.

Results. Next, we discuss our RQs:

RQ1 (Comparing Random Search, NSGAII and NSGAII-SM). To answer RQ1, we ran Random search, NSGAII and NSGAII-SM (with cl =.95) 40 times for 150 min. We computed the HV and the GD values for the pareto front solutions obtained by these alternative search algorithms at every 10 min interval from 0 to 150 min. We used the resulting HV and GD values and the changes in these values over time to first compare NSGAII and NSGAII-SM by focusing on their performance when the algorithms are executed within a practical execution time budget (i.e., 150 min). Second, we compare the better algorithm between NSGAII and NSGAII-SM with Random search. To statistically compare the HV values, we performed the non-parametric pairwise Wilcoxon Paired Signed Ranks test [Capon, 1991], and calculated the effect size using Cohen's d [Cohen, 1977]. The level of significance (α) was set to 0.05, and, following standard practice, d was labeled "small" for $0.2 \leq d < 0.5$, "medium" for $0.5 \leq d < 0.8$, and "high" for $d \geq 0.8$ [Cohen, 1977].

Comparing NSGAII and NSGAII-SM. Figure 3.3(a) shows the HV values obtained by 40 runs of NSGAII and NSGAII-SM up to 150 min. As shown in the figure, at the beginning both NSGAII and NSGAII-SM are highly random. After executing these two algorithms for 50 min, the degree of variance in HV values across NSGAII-SM runs reduces faster than the degree of variance in HV values across NSGAII runs. Further, the average HV values obtained by NSGAII-SM grows faster than the average HV values obtained by NSGAII. After executing the algorithms for 120 min, both search algorithms converge towards their pareto optimal solutions and the difference in average HV values between the two algorithms tends to narrow.

We note that the differences between the HV distributions of NSGAII and NSGAII-SM are not statistically significant. This is likely because the number of runs is rather small (40), thus yielding low statistical power. However, as shown in Figure 3.3(a), the medians and averages of the HV values obtained by NSGAII-SM are higher than the medians and averages of the HV values obtained by NSGAII. Given the large execution time of our test generation algorithm, in practice, testers will likely have the time to run the algorithm only once. With NSGAII, certain runs really fare poorly, even after the initial 50 min of execution. Figure 3.3(c) shows the HV results over time for the worst

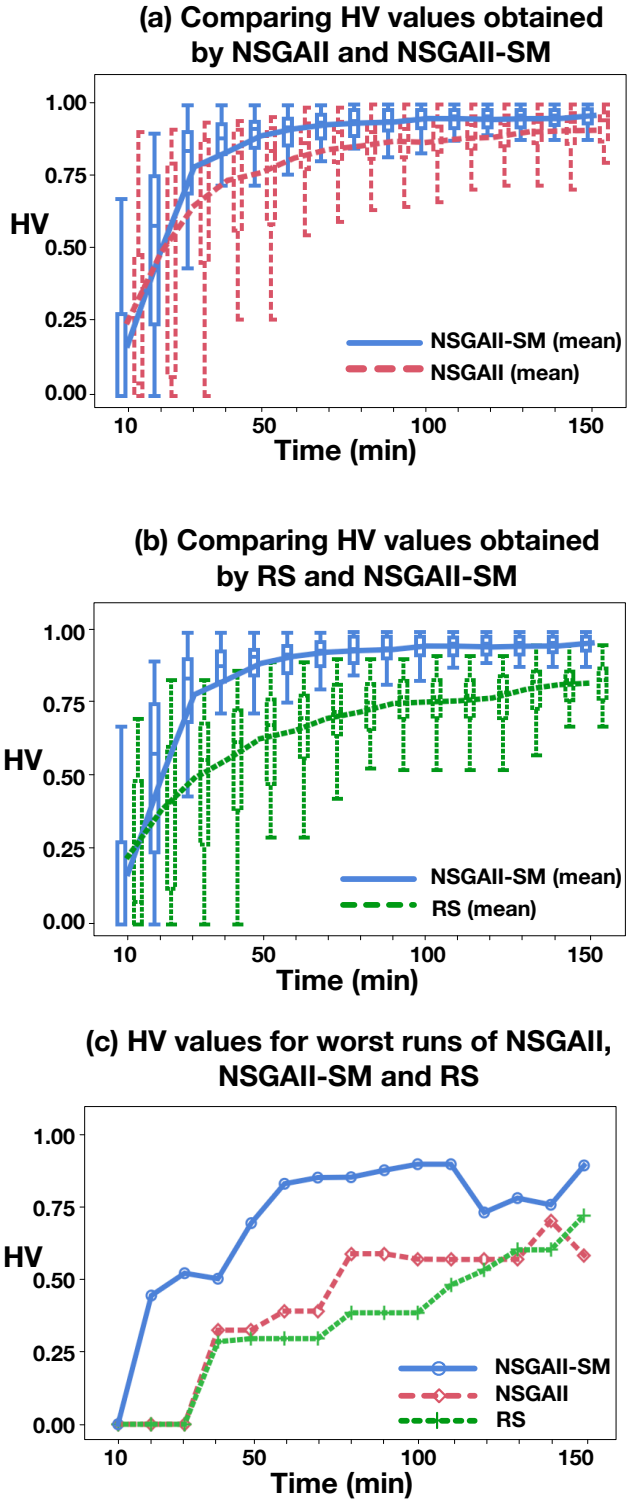

Figure 3.3. Comparing HV values obtained by (a) 40 runs of NSGAII and NSGAII-SM (*cl*=.95); (b) 40 runs of random search and NSGAII-SM (*cl*=.95); and (c) the worst runs of NSGAII, NSGAII-SM (*cl*=.95) and random search.

run of NSGAII, NSGAII-SM and Random search among our 40 runs. As shown in the figure, the worst run of NSGAII yields remarkably lower HV values compared to the worst run of NSGAII-SM. With NSGAII, the tester might be unlucky, and by running the algorithm once, obtain a run similar

to the worst NSGAII run in Figure 3.3(c). Since the worst run of NSGAII-SM fares much better than the worst run of NSGAII, we can consider NSGAII-SM to be a safer algorithm to use, especially under tight time budget constraints. We note that as shown in Figure 3.3(c) the HV values do not necessarily increase monotonically over time. This is because HV may decrease when, due to the crowding distance factor, solutions in sparse areas but slightly away from the reference pareto front are favored over other solutions in the same pareto front rank [Peng and Tang, 2011]. We further compared the GD values obtained by NSGAII and NSGAII-SM for 40 runs of these algorithms up to 150 min. Similar to the HV results, after 50 min executing these algorithms, the average GD values obtained by NSGAII-SM is better than the average GD values obtained by NSGAII.

In addition, we compared the average number of simulations per generation performed by NSGAII and NSGAII-SM. As expected, the average number of simulations per generation for NSGAII is equal to the population size (i.e., ten). For NSGAII-SM, this average is equal to 7.9. Hence, NSGAII-SM is able to perform more generations (iterations) than NSGAII within the same execution time. As discussed in Section 3.4, for $cl = 95\%$, at a given iteration and provided with the same set P, NSGAII-SM behaves the same as NSGAII with a probability of 97.5%, and with a probability of 2.5%, NSGAII-SM produces less accurate results compared to NSGAII. Therefore, given a fixed execution time, NSGAII-SM is able to perform more iterations than NSGAII, and with a high probability ($\approx 97.5\%$), the solutions generated by NSGAII-SM at each iteration are likely to be as accurate as those would have been generated by NSGAII. As a result and as shown in Figure 3.3(a), given the same time budget, NSGAII-SM is able to produce more optimal solutions compared to NSGAII.

Finally, we compared NSGAII-SM with three different confidence levels, i.e., cl = 0.95, 0.9 and 0.8. The HV and GD results indicated that NSGAII-SM performs best, and better than NSGAII, when cl is set to 0.95.

Comparing with Random Search. Figure 3.3(b) shows the HV values obtained by Random search and NSGAII-SM. As shown in the figure, after 30 min execution, NSGAII-SM yields better HV results compared to Random search. The HV distributions obtained by running NSGAII-SM after 30 min and until 150 min are significantly better (with a large effect size) than those obtained by Random search. Similarly, we compared the GD values obtained by NSGAII-SM and Random search. The GD distributions obtained by NSGAII-SM after 30 min and until 150 min are significantly better than those obtained by Random search with a large effect size at 100 min and 110 min and otherwise a medium effect size at other times.

To summarize, when the search execution time is larger than 50 min, NSGAII-SM outperforms NSGAII. With less than 50 min execution time, both algorithms show a high degree of randomness. When engineers cannot afford to run the test generation algorithm for a long time, for example because they make a change to the PeVi system and need to rerun the test execution procedure frequently, NSGAII-SM is more likely to provide close to optimal solutions compared to NSGAII. Further, as shown in Figure 3.3(c), the worst run of NSGAII-SM performs considerably better than the worst run

of NSGAII. Finally, NSGAII-SM is able to find significantly better solutions compared to Random search.

RQ2 (Usefulness). To demonstrate practical usefulness of our approach, we have made available at [Ben Abdessalem, 2018b] some test scenario examples obtained by our NSGAII-based test generation algorithms. We presented these test scenarios as well as other scenarios to domain experts at our partner company. The scenarios were generated for various stressful weather conditions (e.g., fog, snow and rain) and for situations where roadside objects block the camera's field of view or when ramped and curved roads may interfere with the pedestrian detection function of PeVi. In all the example scenarios at [Ben Abdessalem, 2018b] either PeVi fails to detect a pedestrian that appears in the red warning area (AWA) in front of a car, or the detection happens very late and very close to the collision time. As confirmed by our domain expert, such scenarios had not been previously developed by manual testing based on domain expertise. These scenarios particularly helped engineers identify particular car speed and pedestrian speed ranges and pedestrian orientations for which the PeVi's detection function is more likely to fail. In addition, the light scene intensity, the light orientation and reflection may impact the detection capabilities of pedestrian detection algorithms. However, due to the current imitations of PreScan (the PeVi simulation tool) discussed earlier, we were not able to define fitness functions related to the scene light intensity.

To summarize, our NSGAII-based test generation algorithms are able to identify several critical behaviors of the PeVi systems that have not been previously identified based on manual and expertise-based simulations.

3.7 Conclusions

Physics-based simulation tools provide feasible and practical test platforms for control and perception software components developed for self-driving cars. We identified the following two key challenges that hinder systematic testing based on these simulation tools: (1) These tools lack the guidance and automation required to generate test cases that would be likely to uncover faulty behaviors, and (2) executing individual test cases is very time-consuming. In this chapter, we proposed an approach based on combination of multi-objective search and neural networks. We developed meta-heuristics capturing critical aspects of the system and its environment to guide the search towards exercising behaviors that are likely to reveal faults. Our proposed search algorithm relies on neural network predictions to bypass actual costly simulations when predictions are sufficient to conclusively prune certain solutions from the search space. Our evaluation performed on an industrial system shows that (1) our search-based algorithm outperforms random test generation, (2) combining our search algorithm with neural networks improves the quality of the generated test cases under a limited and realistic time budget, and (3) our approach is able to identify critical system and environment behaviors.

In this Chapter, we relied on a subset of the PeVi input elements, i.e., the properties of the vehicle and the pedestrian, to generate test cases. To account for various critical properties of the environment,

in the next chapter, we rely on properties of all the input elements, where we use a different ADAS case study.

Chapter 4

Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms

Vision-based control systems (i.e., ADAS) are key enablers of many autonomous vehicular systems, including self-driving cars. Testing such systems is complicated by complex and multidimensional input spaces. Search-based techniques are best suited for testing at the system level [Zeller, 2017]. They provide effective and flexible guidance for test generation, going beyond test generation based on structural coverage that is not often effective or scalable for system testing. Even though evolutionary search algorithms often scale well to large input spaces, their ability to effectively identify critical test scenarios may diminish as the search space increases in size and dimensions. This is mostly because the search may be stuck in local optima in less critical parts of the input space [Luke, 2013].

In this chapter, we provide an algorithm to improve effectiveness of the evolutionary search for large and multidimensional input spaces. Our algorithm builds on *learnable evolution models*, a machine learning-guided form of evolutionary computation [Michalski, 2000, Wojtusiak and Michalski, 2004]. Specifically, we propose to use the set of scenarios generated at intermediary search iterations to build *decision tree classification models* [Witten et al., 2011]. Decision trees learn the characteristics of the critical test scenarios and identify critical regions in an input space (i.e., the regions of a test input space that are likely to contain most critical test scenarios). We then focus the subsequent search iterations on the critical regions, generating and evolving more critical test scenarios within those regions using genetic operators. We iteratively build decision trees followed by search iterations focused on critical regions identified by the trees. The process stops when we run out of our search time budget. Our algorithm, in addition to guiding the search towards the critical test scenarios faster, produces a decision tree model that identifies the critical regions of the system under test. The critical region characterizations help engineers understand the conditions on input variables that may lead to failures.

This chapter highlights the following research contributions:

1. We propose a lightweight formalism for vision-based control systems used in self-driving cars (i.e., ADAS). Our formalism specifies ADAS input and output variables and their critical behaviors. Our formalism is developed based on our analysis of different ADAS examples (see [data, 2017]) as well as the characteristics of a widely-used, industrial ADAS simulation tool [TASS-International, 2019].
2. We propose a system testing algorithm that combines evolutionary search algorithms and decision tree classification models. Our algorithm has two main objectives, which are important in the context of testing ADAS systems: First, classification models guide the search-based generation of tests faster towards critical test scenarios. Second, search algorithms refine classification models so that the models can accurately characterize critical regions.
3. We evaluate our approach on an industrial automotive system.

Organization. This chapter is structured as follows. Section 4.1 motivates our work. Section 4.2 provides an ADAS formalization. Section 4.3 describes our approach. Section 4.4 evaluates our approach, and Section 4.5 concludes this chapter.

4.1 Motivating Case Study

Figure 4.1 shows an overview of an ADAS example referred to as the Automated Emergency Braking (AEB) system. Its main function is to identify pedestrians in front of a vehicle and to avoid collision by applying the brake when it is necessary. AEB has three main components: (1) *The Sensor Component.* This component identifies the position and speed of objects in a cone-shaped area in front of a vehicle (i.e., the field of view). It also computes the *time to collision (TTC)* that measures the time required for a vehicle to hit an object if both continue with the same speed and do not change their paths [van der Horst and Hogema, 1993]. When an object is detected in front of a vehicle and when the TTC is below a defined threshold, the object position is sent to the vision component. (2) *The Vision (Camera) Component.* This component detects object types and shapes after receiving their positions from the sensor component. Specifically, they determine whether the object is a pedestrian (human or animal), a car, a traffic-sign, etc. Then, the system is able to decide whether braking is needed and sends a command to the brake control component when it is necessary. (3) *The Braking Control Component.* This component applies the braking request.

To simulate AEB, we use the PreScan simulator. PreScan allows us to define and execute scenarios capturing various road traffic situations and different pedestrian-to-vehicle and vehicle-to-vehicle interactions. In addition, using PreScan, one can vary road-topologies, weather conditions and infrastructures in test scenarios.

Figure 4.2 shows a domain model capturing the test input space and the output of AEB. Based on our analysis, we categorize the AEB input variables into two categories:

I. *Static input variables*. The values of these variables are fixed during ADAS simulation and they include: (1) Different road types (e.g., straight, curved or ramped). For the curved and ramped roads,

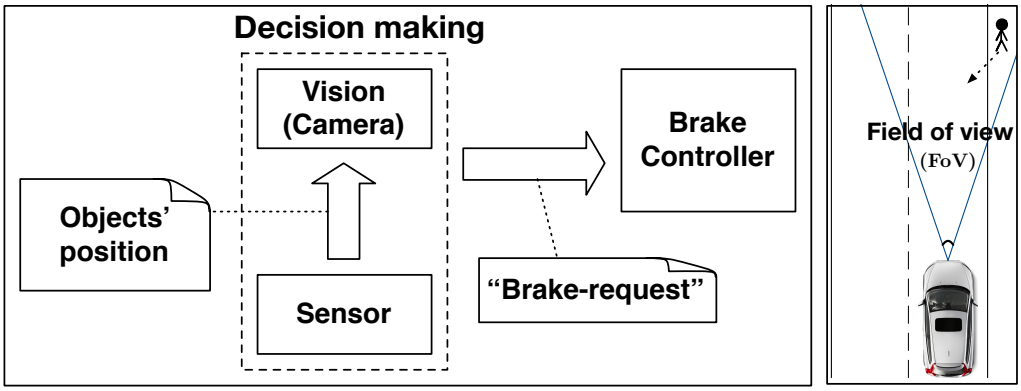

Figure 4.1. An example of a vision-based control system: Automated Emergency Braking (AEB) system.

we specify the curve radius and the ramp height, respectively. (2) Different weather types: normal, rainy and snowy. For each of the snowy and rainy weather types, we specify the level of precipitation. For each weather type, we may or may not have fog with different density levels. Finally, we specify a visibility range, i.e., the distance at which the objects can be clearly seen. As Figure 4.2 shows, we have defined enumerations for the road radius and height, the level of precipitation for rain and snow, fog density, and visibility range. According to the domain experts in IEE, these enumerations provide a desired level of granularity for analysis, and hence, static variables do not need to be real or integer.

II. *Dynamic (mobile) objects*. They indicate objects that change their positions during ADAS simulation, i.e., pedestrians and vehicles. For AEB, we consider two mobile objects: one pedestrian and one vehicle, and assume linear trajectories for them. These assumptions are meant to reduce the complexity of test scenarios and were suggested by the domain experts. For the vehicle, we require to know its initial speed (v_0^c). The pedestrian has four variables characterizing its initial position along x and y axes and relative to the position of the vehicle (x_0^p, y_0^p), its orientation angle (θ_0^p) and its initial speed (v_0^p). The dynamic objects variables are float. Figure 4.3 shows the ranges for the pedestrian initial position and orientation variables when the road is curved, ramped and straight, respectively. The ranges for vehicle and pedestrian speed variables are $[1km/h..90km/h]$ and $[1km/h..18km/h]$, respectively.

In addition to variable ranges, the valid inputs of ADAS are determined by constraints defined over the input variables. These constraints are either defined on static input variables, or they specify how value assignments to static variables impact the ranges of the mobile object variables. An example of the constraints defined over AEB static variables is shown using the OCL language [Group, 2017] in Figure 4.2 (see WeatherC-OCL). The constraint states that when there is no fog, the visibility range is set to maximum. The constraints that relate static variables of AEB to ranges of mobile object variables are captured in Figure 4.3. Specifically, the figure specifies the valid ranges for pedestrian position and orientation variables corresponding to different road topologies.

ADAS simulations have two outputs: **I.** Position vectors for mobile objects (i.e., position vectors for the vehicle and the pedestrian in the AEB case study). The position vector related to each mobile

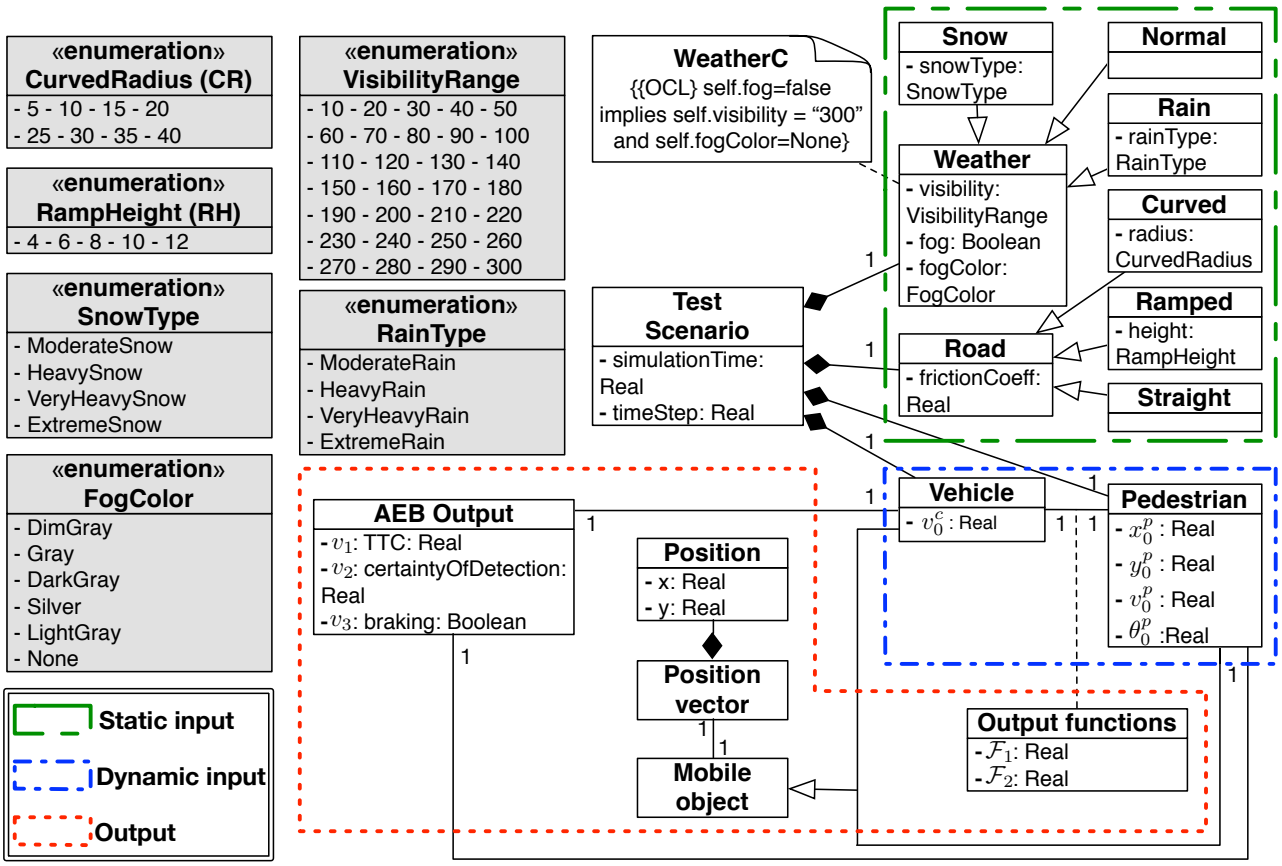

Figure 4.2. The AEB domain model.

Curved road	Ramped road	Straight road
Range θ_0^{P} = [120..250]	Range θ_0^{P} = [40..160]	Range θ_0^{P} = [40 ..160]
Range $\mathrm{x}_0^{\mathrm{P}}$= [32..50]	Range $\mathrm{x}_0^{\mathrm{P}}$= [60..95]	Range $\mathrm{x}_0^{\mathrm{P}}$= [30 ..85]
Range $\mathrm{y}_0^{\mathrm{P}}$= [50..76]	Range $\mathrm{y}_0^{\mathrm{P}}$= [2..16]	Range $\mathrm{y}_0^{\mathrm{P}}$= [24 ..36]

Figure 4.3. The ranges of the pedestrian position (x_0^p, y_0^p) and orientation (θ_0^p) for different road topologies.

object stores the position of that object at each simulation time step. **II.** Function specific output variables: Each ADAS, depending on its function, produces some outputs. For example, AEB produces three outputs corresponding to its three main components: (1) *Time to collision (TTC)* generated by the sensor component and discussed earlier. (2) *certaintyOfDetection* generated by the vision component which is a percentage value indicating the probability that the detected object is a pedestrian. (3) *Braking* that indicates whether a braking request has been triggered.

The following describes the main AEB critical (or failure) behavior extracted from the AEB requirements: "*AEB detects a pedestrian in front of the car with a high degree of certainty, but an accident happens where the car hits the pedestrian with a relatively high speed (i.e., more than* $30km/h$*)*".

We denote this critical behavior by *CB*, and refer to any AEB simulation scenario exhibiting this behavior as a *critical test scenario* of AEB.

The test input space of AEB is large and multidimensional. As we will specify in Section 4.2, it consists of four enumeration (static) and five float (dynamic) variables. Considering only the static AEB variables, their total number of value assignments is $11,242$. Further, AEB simulations (and in general ADAS simulations) are computationally expensive. This is because the underlying simulator (e.g., PreScan) builds on high-fidelity mathematical models and takes a relatively large amount of time to run (e.g., on average, each AEB simulation takes 1.2 min). Our goal is to provide an effective algorithm that, within a reasonable testing time budget: (1) generates AEB critical test scenarios (i.e., those exhibiting *CB*), and (2) identifies under what conditions on the AEB input variables such critical scenarios are more likely to occur. The latter will provide engineers with critical region characterizations, allowing them to better understand the conditions under which AEB fails to behave correctly.

4.2 ADAS Formalization

In this section, we formalize an ADAS system and its environment. Our formalization is meant to help define our algorithm precisely, and to demonstrate how our work can be applied to other ADAS systems. Our formalization is developed based on our analysis of different ADAS examples [data, 2017] and the input and configuration variables of the PreScan tool [TASS-International, 2019]. Generic descriptions of our ADAS examples can be found on the Bosch website [Bosch, 2017].

Definition 4.2.1. We define an ADAS as a tuple (S, O, I, D, C), where

- $S = \{s_1, \ldots, s_n\}$ is a set of *variables* specifying (immobile) static environment aspects.

- O is a set of *mobile objects* (pedestrians and vehicles).

- $I = \{i_1, \ldots, i_m\}$ is a set of variables specifying *initial states* of the mobile objects in O. Each variable in I is related to one mobile object in O, while each mobile object $o \in O$ is related to one or more variables in I.

- D is a set of domains of values for variables in $S \cup I$. In particular, D is partitioned into D_S and D_I ($D = D_S \cup D_I$) such that $D_S = \{D_1, \ldots, D_n\}$ is a set of finite value sets to variables in S, while $D_I = \{D'_1, \ldots, D'_m\}$ is a set of infinite value sets to variables in I. Specifically, D_j is the set of values for $s_j \in S$, and D'_j is an interval $[min...max]$ of real values specifying the values that $i_j \in I$ can take.

- C is a set of Boolean propositional constraints over $S \cup I$. The set C is partitioned into C_S and C_I such that constraints in C_S are defined on finite-domain variables in S, and constraints in C_I relate finite-domain variables in S to infinite-domain variables in I.

Example 4.2.1. We formalize AEB in Figure 4.1 as follows:

- Static variables (S): s_1 (precipitation), s_2 (fogginess), s_3 (road shape) and s_4 (visibility range).

- Mobile objects (O): o_1 (vehicle) and o_2 (pedestrian).

- Dynamic variables (I): v_0^c (initial speed of vehicle), v_0^p (initial speed of pedestrian), x_0^p (initial position of pedestrian on the x-axis), y_0^p (initial position of pedestrian on the y-axis) and θ_0^p (the orientation of pedestrian).

- The domain of s_1 is the union of RainType and SnowType enumerations in Figure 4.2 as well as a value for normal weather. Variable s_2 takes values from the FogColor enumeration. The domain of s_3 is the union of RampedHeight and CurvedRadius enumerations and a value for the straight road. Variable s_4 takes values from the VisibilityRange enumeration. The ranges for dynamic variables were discussed in Section 4.1.

- The constraints over static variables (C_S) relate the level of fog (s_2) to the visibility range (s_4). An example of a C_S constraint is: $(s_2 = \text{“}DimGray\text{”} \Rightarrow s_4 = 10 \vee \ldots \vee s_4 = 100)$. The constraints over static and dynamic variables (C_I) relate the shape of the road (s_3) to different ranges for x_0^p, y_0^p and θ_0^p (see Figure 4.3). An example of a C_I constraint is: $(s_3 = \text{“}RH4\text{”} \vee \ldots \vee s_3 = \text{“}RH12\text{”} \Rightarrow D_{x_0^p} = [60..95] \wedge D_{y_0^p} = [2..16] \wedge D_{\theta_0^p} = [40..160])$.

We denote by $Z \subseteq D_1 \times \ldots \times D_n \times D'_1 \times \ldots \times D'_m$ the set of value assignments to variables in $S \cup I$ satisfying all the constraints in C. An ADAS simulation function Σ takes as input a value assignment $z \in Z$ and a value $T \in \mathbb{N}$ indicating the simulation duration (i.e., the number of simulation steps). The output of Σ is (1) a set U of *output vectors* indicating the position and speed of mobile objects at each simulation time step, and (2) a set V of (time-independent) output variables. Specifically, U captures the dynamic behavior of ADAS and the environment (i.e., how mobile objects move over time). Each position vector $u \in U$ corresponds to one and only one mobile object $o \in O$ and is a function $u : \{0, 1, \ldots, T\} \rightarrow \mathbb{R}^3$ where $u(t)$ ($t \in \{0, \ldots, T\}$) is a triple (x, y, v) indicating the position (x, y) and the speed v of the mobile object related to u at time t. The set V determines the function-specific outputs produced by decision-making components of an ADAS.

Example 4.2.2. AEB generates two position vectors (U): u_1 (for vehicle) and u_2 (for pedestrian); and three decision-making outputs (V): (1) TTC denoted by v_1, (2) *certaintyOfDetection* denoted by v_2, and (3) *braking* denoted by v_3 (see Figure 4.2).

To specify critical behaviors of ADAS, we define (auxiliary) functions over the dynamic system outputs U. For example, let u_1 and u_2 be position vectors generated for AEB over simulation time T. We define two functions: (1) $\mathcal{F}_1(u_1, u_2)$ that computes the minimum distance between the pedestrian (u_2) and the field of view of the vehicle (u_1), and (2) $\mathcal{F}_2(u_1, u_2)$ that computes the speed of the car at the time of collision, and returns -1 if collision does not occur.

We formalize the AEB critical behavior *CB* described in Section 4.1. Given AEB outputs $U = \{u_1, u_2\}$ and $V = \{v_1, v_2, v_3\}$, we define $CB(U,V)$ as follows:

$$CB(U,V) = (\mathcal{F}_1(u_1,u_2) < 50cm) \wedge (v_2 > 0.5) \wedge (\mathcal{F}_2(u_1,u_2) > 30km/h) \tag{4.1}$$

The *CB* property states that: a pedestrian is in front of a car ($\mathcal{F}_1(u_1,u_2) < 50cm$), is detected by AEB with a high certainty ($v_2 > 0.5$), and the car hits the pedestrian with a speed higher than $30km/h$ ($\mathcal{F}_2(u_1,u_2) > 30km/h$). The constant values $50cm$, 0.5 and $30km/h$ are taken from the AEB specification. An AEB test scenario generating U and V is critical if and only if $CB(U,V)$ is true.

4.3 Search Guided by Classifiers

In this section, we describe our ADAS testing algorithm that combines multi-objective search and decision tree classification models.

4.3.1 Multi-objective search

The formalization of ADAS critical behaviors depends on several ADAS outputs. For example, formalizing the *CB* behavior (see equation 4.1) relies on three AEB outputs $\mathcal{F}_1$, $\mathcal{F}_2$ and v_2. We cast the problem of computing ADAS critical test scenarios as a *multi-objective search optimization problem* where the ADAS outputs specifying its critical behaviors act as the search fitness functions. We use the NSGAII algorithm, which is discussed in Section 2.1.1. NSGAII algorithm generates a set of solutions forming a *Pareto nondominated front*. In our work, NSGAII generates a number of ADAS critical test scenarios by maximizing or minimizing the ADAS outputs characterizing its critical behavior. In the following, we discuss how we tailor NSGAII to ADAS testing:

Representation. A feasible solution is a vector of values to static variables $s_1, \ldots, s_n$ and dynamic variables $i_1, \ldots, i_m$ of the ADAS under analysis such that each vector satisfies the constraints in C. Each such vector defines an ADAS test scenario. Simulating each vector generates outputs U and V that can be used to compute fitness functions.

Initial population. An initial population for our search algorithm is a set P consisting of vectors of ADAS test scenarios. We aim to generate P by selecting a diverse set of vectors from the input space. We generate P with size q as follows: First, we generate q vectors of value assignments to static variables $s_1, \ldots, s_n$ using t-wise combinatorial testing [Kuhn et al., 2004] such that (1) the C_S constraints hold, and (2) the pairwise coverage of variables s_1 to s_n is maximized. We use the PLEDGE tool [Henard et al., 2013] for this purpose. Second, we use an adaptive random search algorithm [Luke, 2013] to generate a large number ($> q$) of value assignments to dynamic variables $i_1, \ldots, i_m$. Adaptive random search is an extension of the naive random search that attempts to maximize the Euclidean distance between the points selected in the input space. Third, for each static

variable vector, we select a dynamic variable vector such that the constraints C_I (i.e., constraints between static and dynamic variables) hold. If for some static variable vector v we cannot find such dynamic variable vector among the existing randomly generated pool, we perform some more iterations of (adaptive) random search within the value ranges accepted by the C_I constraint for v. The initial population set P is complete when every static variable vector is matched to one dynamic variable vector. Note that in our ADAS formalization, we do not have any constraint among the dynamic variables.

Fitness Functions. Fitness functions are defined based on the ADAS outputs specifying its critical behavior. For the AEB case study, fitness functions are the two functions $\mathcal{F}_1$ and $\mathcal{F}_2$, and the output variable v_2. These are used to formalize the critical behavior of AEB (the *CB* behavior in Section 4.2). To generate critical test scenarios, we maximize $\mathcal{F}_2$ and v_2, and minimize $\mathcal{F}_1$. This is because for scenarios exhibiting *CB*, the values of $\mathcal{F}_2$ and v_2 should be larger than a threshold, and $\mathcal{F}_1$ should be smaller than a threshold.

Genetic operators. The genetic operators of NSGAII should be defined such that the generated test scenario vectors satisfy the C_S and C_I constraints. Here, we provide crossover and mutation operators that respect pairwise C_S constraints, and C_I constraints relating one static variable to one or more dynamic variables. The constraints of the ADAS systems we have studied in our work [data, 2017] conform to these conditions. Specifically, in all of these systems, the C_S constraints relate the weather properties (e.g., fog-level (s_2) to visibility range (s_4)), and the C_I constraints relate different road shape types (s_3) to the ranges of dynamic variables x_0^p, y_0^p, and θ_0^p.

Selection: We use a binary tournament selection with replacement that has been used in the original implementation of NSGAII [Deb et al., 2002].

Crossover: To avoid violating the C_S constraints, crossover is not applied to the static segments of the vectors. That is, our crossover operator is applied to dynamic segments of the vectors only (i.e., $(i_1, \ldots, i_m)$). To avoid violating the C_I constraints, we match pairs of vectors with the same value for the static variables participating in C_I (e.g., the same value for s_3 in the AEB case study). If we do not find any match for some parent vector, we match two vectors with the smallest Euclidean distance between the variables participating in the C_I constraints. We then use Simulated Binary Crossover operator (SBX) [Beyer and Deb, 2001, Deb and Agrawal, 1995] that has been previously applied to vectors of float variables. The difference between offsprings generated by SBX and their parents is controlled by a distribution index (η): The offsprings are closer to the parents when η is large, while with a small η, the difference between offsprings and parents will be larger [Deb and Beyer, 2001]. In this chapter, we chose a high value for η (i.e., $\eta = 20$) based on existing guidelines [Deb and Agrawal, 1995]. Given that η is large, even when parents do not have the same values for the static variables in C_I, the values of the dynamic variables in each of the two offsprings are likely to fall within the valid ranges of their respective parent vector. Hence, the C_I constraints are likely to still hold after applying SBX in such situations. If the resulting values are out of variable ranges after crossover, we cap them at the max or min of the ranges when they are closer to the max or min, respectively.

Mutation: Mutation is applied after crossover to static and dynamic variables with a probability (mutation rate). To avoid violation of the C_I constraints, we do not mutate static variables participating in the C_I constraints. Note that since the initial population is generated by maximizing pairwise coverage of static variables, different value combinations of the static variables in C_I are already present in the initial population. Except for static variables in C_I, all other static and dynamic variables can be mutated. We mutate a static variable not appearing in C_S by randomly changing its value within its valid range. For a pair s_i and s_j of static variables appearing in a C_S constraint we define a closed mutation operator as follows: after mutating s_i (respectively s_j), we identify the set of values for s_j (respectively s_i) consistent with the new value of s_i (respectively s_j), and randomly change s_j (respectively s_i) to one of those values. To mutate a dynamic variable, we shift the variable by a value selected from a normal distribution with mean $\mu = 0$ and a small variance. Similar to the crossover operator, if the resulting values are out of variable ranges, we cap them at the max or min of the ranges when they are closer to the max or min, respectively.

4.3.2 Decision tree learning

Decision tree learning is a supervised learning classification technique [Witten et al., 2011, Alpaydin, 2010a]. They are divided into *regression* and *classification* techniques where the goal is to predict real-valued and categorical outputs, respectively. In this chapter, we use classification decision trees. Recall from Section 2.2 that supervised learning techniques are trained based on labeled data. In this chapter, we use Boolean functions such as *CB* (see equation 4.1 in Section 4.2) to label each ADAS test scenario as critical or non-critical. Alternatively, we could characterize the critical behavior as a real-valued function and use regression trees instead.

In contrast to other learning techniques (e.g., SVM), decision tree boundaries are parallel to the dimensions of the input space and expressible in terms of linear conditions over input variables. This makes decision tree boundaries understandable by practitioners, and has been a main reason why we selected them in our work.

Figure 4.4 shows two decision trees generated for the AEB case study. The input data for building decision trees is a set of AEB test scenario vectors. The label for each scenario is computed by first simulating the scenario and then labeling it either as critical or non-critical by applying the *CB* function to the scenario simulation outputs. A decision tree model is built by partitioning the set of labeled test scenarios in a stepwise manner aiming to create partitions with increasingly more homogeneous labels (i.e., partitions in which the majority of scenarios are labeled either as critical or non-critical). For example, the tree in Figure 4.4(a) shows that out of the 636 scenarios that use a straight, ramped or curved (with **CR = 5**) road, 98% were not critical (did not exhibit *CB*).

The tree leaves containing more than 50% critical scenarios (i.e., leaves **A** to **D** in Figure 4.4) are critical regions. For example, out of the total of 1200 scenarios, 230 of them are classified in the critical region (**A**), and 69% of them are critical. Each critical region is specified by conjoining the conditions appearing on the path from the root to the critical region. For example, the critical region **A** is characterized as follows: $v_0^P \geq 7.2\text{km/h} \wedge \theta_0^P < 218.6° \wedge (s_3 = \text{“}CR10\text{”} \vee \ldots \vee s_3 = \text{“}CR40\text{”})$.

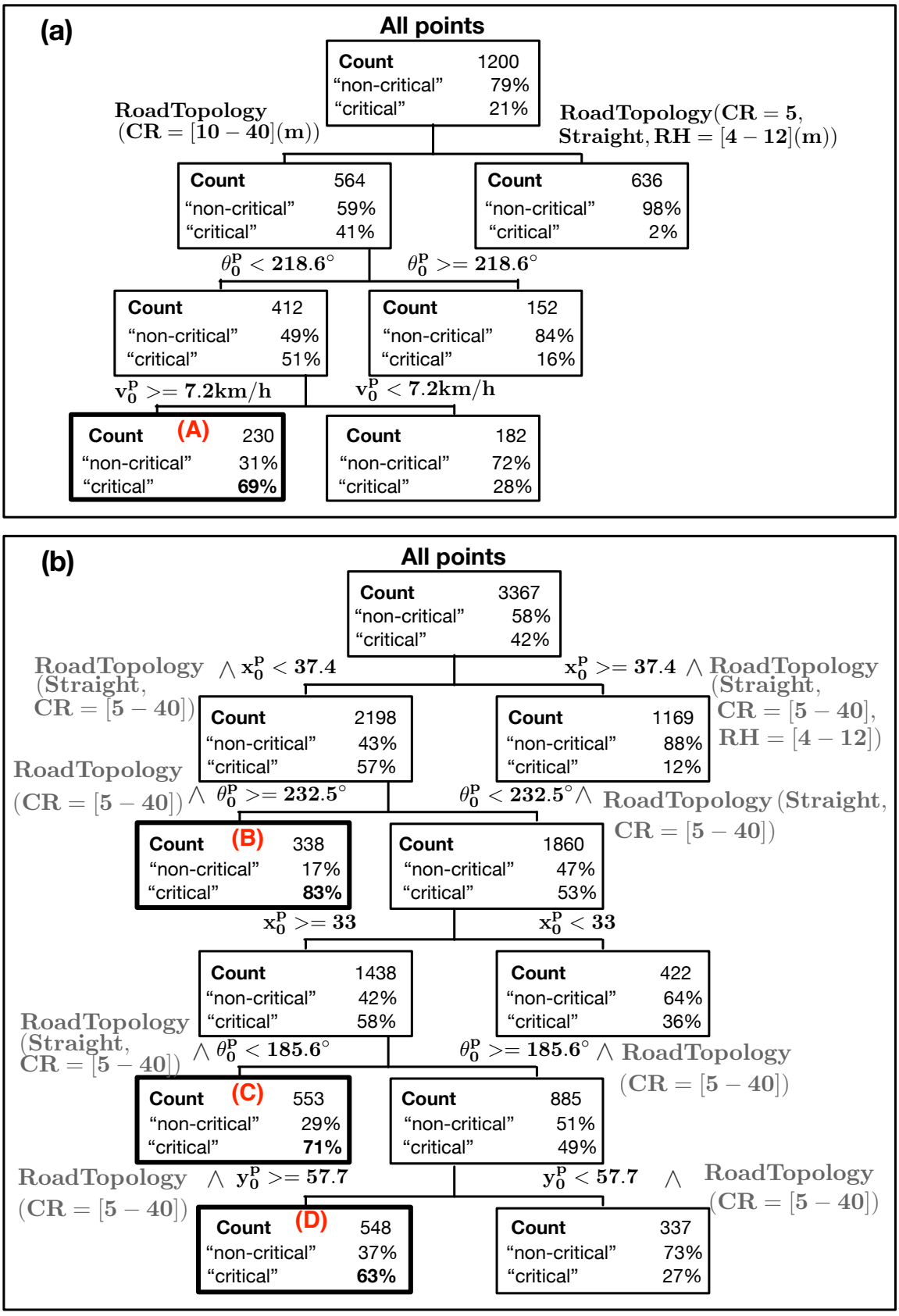

Figure 4.4. Decision trees generated our approach for the AEB system: (a) An initial decision tree, and (b) A decision tree obtained after some iterations of the NSGAII-DT algorithm.

At each (non-leaf) node, a decision tree partitions the data in that node based on a condition on only one variable. However, due to the C_S and C_I constraints, a decision tree condition on a variable v may additionally constrain variables other than v but related to v via C_S or C_I. As discussed in Section 4.3.1, our genetic operators respect the C_S and C_I constraints. Hence, when our search algorithm applies these operators to a specific critical region (as we will discuss in Section 4.3.3), the operators automatically handle both the constraints explicitly identified by the tree and the additional constraints implied by C_S or C_I. However, as critical region characterizations are among outputs of our approach, we explicate these additional constraints in outputs presented to engineers. For example, in Figure 4.4(b), the conditions in gray color are not generated by the decision tree but are implied by the AEB constraints.

We note that, in this chapter, we do not use decision trees to predict whether a given ADAS scenario is critical or not (i.e., the decision trees are not used as predictor models). We exclusively use the decision trees: (1) to better guide the search, and (2) to characterize the critical regions of the ADAS input space. Further, to avoid overfitting in the trees generated by our approach, in Section 4.4.3, we define a stopping criterion to control the tree expansion such that the number of vectors in each tree leaf does not fall below a certain threshold.

4.3.3 NSGAII guided by decision trees

Algorithm 4 shows our proposed algorithm, NSGAII-DT, that generates critical test scenarios and critical regions for ADAS. NSGAII-DT receives as input an ADAS specification, a set of (quantitative) fitness functions, a Boolean *label* function indicating whether a test scenario is critical or not, and a parameter g indicating the number of search iterations we perform in each critical region. The output of NSGAII-DT is a set of the critical test scenarios and the critical regions $R_1, \ldots, R_k$ of the ADAS input space.

NSGAII-DT starts with an initial and randomly selected population set P (line 2). Each iteration of NSGAII-DT consists of the following main steps: *First*, it performs a number of (genetic) search iterations using NSGAII in critical regions of the input space (lines 6–10). Specifically, for each critical region R_i, the set Q of the elements inside R_i is passed as the initial population to NSGAII along with the parameter g (i.e., the number of search iterations to be applied in R_i). We further pass R_i and C to NSGAII. In particular, R_i specifies the ranges of input variables valid for the critical region under search. Provided with the variable ranges and the constraints (i.e., the set C), our mutation and crossover operators described in Section 4.3.1 can generate new vectors within the region R_i. Note that in the first iteration, the only critical region is the entire input space (R_1 in line 4). NSGAII returns two sets: Q' and B where Q' is the set of all scenarios and B is the set of most critical scenarios computed by NSGAII.

Second, NSGII-DT identifies the scenarios on the best Pareto front rank computed so far (lines 11–12). In particular, it identifies the best Pareto front rank in set *Best* (i.e., the set of all best solutions generated by all invocations of NSGAII). *Third*, NSGII-DT builds a decision tree based on the labeled set of all the scenarios generated up to that point (lines 13–14). Specifically, it computes the label for

Algorithm 4: NSGAII-DT

Input: (S, O, I, D, C): An ADAS specification
Input: $F_1, \ldots, F_l$: Search fitness functions
Input: *label*: A Boolean function to label scenarios as critical/non-critical
Input: g: Number of search iterations to be applied at each critical leaf

Result: *criticalScenarios*: A set of critical test scenarios
Result: $R_1, \ldots, R_k (\subseteq D_1 \times \ldots \times D_n \times D'_1 \times \ldots \times D'_m)$: A set of critical regions

1 **begin**
2 Select an initial population set P randomly.
/*Each $p \in P$ is a vector of values for $(s_1, \ldots, s_n, i_1, \ldots, i_m)$ */
3 $k \leftarrow 1$; $Best \leftarrow \emptyset$
4 $R_1 \leftarrow D_1 \times \ldots \times D_n \times D'_1 \times \ldots \times D'_m$
/*R_1 is the entire search space and includes all elements in P*/
5 **repeat**
6 **for** $i = 1$ **to** k **do**
7 $Q \leftarrow P \cap R_i$
8 $B, Q' \leftarrow$ NSGAII$(g, Q, F_1, \ldots, F_l, R_i, C)$
/*Inputs passed to NSGAII:
g: the number of search iterations applied to each critical leaf;
Q: the set of scenarios used as the initial population of NSGAII;
$F_1, \ldots, F_l$: search fitness functions;
R_i: the critical leaf in which we want to run NSGAII; and
C: the ADAS constraints.
Outputs received from NSGAII:
Q': all the solutions generated during search; and
B: best solutions generated by NSGAII.*/
9 $P \leftarrow P \cup Q'$
10 $Best \leftarrow B \cup Best$
11 $rank_1, \ldots, rank_t \leftarrow ComputeRanks(Best)$
12 $criticalScenarios \leftarrow rank_1$
13 $(P^+, P^-) \leftarrow ComputeLabel(label, P)$ /*P^+ : *non-critical*, P^- : *critical**/
14 Build a decision tree *Tree* based on (P^+, P^-)
15 Let $R_1, \ldots, R_k$ characterize the leaves of *Tree* where P^- has a higher probability than P^+
/* For each region $R_i = d_1 \times \ldots \times d_n \times d'_1 \times \ldots \times d'_m$,
we have: $\forall j \in \{1, \ldots, n\} \Rightarrow d_j \subseteq D_j$, and $\forall j \in \{1, \ldots, m\}$,
$\exists min \in D'_j, \exists max \in D'_j$ s.t. $min < max \wedge d'_j = [min..max]$*/
16 **until** *search time has run out*

each $p \in P$, partitions P into P^+ (the non-critical set) and P^- (the critical set), and builds a decision tree based on the labeled data. *Fourth*, it updates the set of desired input space regions in which subsequent search iterations are performed (line 15). Specifically, it identifies the *critical* leaves $R_1, \ldots, R_k$ of the tree such that the probability of failure P^- is higher than that of non-failure P^+. Each R_i is a sub-region of the ADAS input space and is specified as the conjunction of the conditions on the tree paths leading to the leaves containing more elements from P^- than from P^+.

NSGAII-DT can be stopped when we run out of time. Alternatively, we can stop NSGAII-DT when all the tree leaves classify critical scenarios with a high probability (e.g., more than 95%) or when the fitness functions do not improve for the scenarios in the *criticalScenarios* set compared with the previous iteration.

For example, the decision trees in Figure 4.4 are computed by applying NSGAII-DT to the AEB case study. Figure 4.4(a) shows an initial tree, and Figure 4.4(b) shows a tree after a few search iterations. The tree in Figure 4.4(b) contains more conditions, and identifies three critical regions B, C and D, instead of one such region in Figure 4.4(a). Further, the regions B, C and D are considerably more specific than region A as they prune the domains of the input variables more.

We note three important aspects of NSGAII-DT: (1) In ADAS testing, the most time-consuming part of the search is running simulations to compute fitness functions. NSGAII-DT does not increase the number of simulations compared to NSGAII. The fitness values computed by the NSGAII search (line 8) are reused at line 13 to label the new elements. (2) To rebuild the tree in line 14, we use all the scenarios generated and simulated by NSGAII (i.e., Q'). Since computing simulations is expensive and to build more accurate trees, we try to reuse as much as possible the simulation outputs computed by NSGAII. (3) In our work, we run NSGAII in leaves that classify critical scenarios with a probability lower than 95%. This is to use search time budget exploring the critical parts in the input space about which we have less certainty regarding criticality. These are parts of the space where the tree may need to be refined.

4.4 Evaluation

In this section, we present the result of our evaluation performed on the AEB case study.

4.4.1 Research Questions

RQ1. *Does the decision tree technique help guide the evolutionary search and make it more effective?* The most important criterion for a search algorithm to be effective in the context of ADAS testing is that it should be able to generate critical test scenarios, in particular, in large and multidimensional search spaces. To answer this question, we determine whether NSGAII-DT (i.e., our proposed algorithm that is guided by both decision trees and genetic operators) is able to generate scenarios that are more critical compared to those obtained by the NSGAII algorithm (i.e., the baseline evolutionary search algorithm).

RQ2. *Does our approach help characterize and converge towards homogeneous critical regions?* After evaluating the ability of NSGAII-DT in generating critical test scenarios in **RQ1**, we evaluate the critical regions. In particular, in **RQ2**, we investigate whether the decision trees generated by NSGAII-DT are able to precisely characterize critical regions in ADAS input spaces and increasingly do so better over NSGAII-DT iterations.

At the end of Section 4.4.4, we provide qualitative insights into the benefits of our approach from the perspective of practitioners.

4.4.2 Metrics

To answer **RQ1**, we compare the Pareto fronts generated by NSGAII-DT and NSGAII using three well-known quality indicators for evaluating multi-objective search results [Knowles et al., 2006b]: Hypervolume (HV), Generational Distance (GD), and Spread (SP). To compute the quality indicators, following existing guidelines in the literature [Wang et al., 2016], we compute a *reference* Pareto front as the union of all the non-dominated solutions obtained from all runs of NSGAII-DT and NSGAII. The HV quality indicator [Zitzler and Thiele, 1999b] measures the size of the space covered by the members of a Pareto front generated by a search algorithm. The higher this size, the better the results of the algorithm. The GD quality indicator [Van Veldhuizen and Lamont, 1998b] measures the Euclidean distance between members of a Pareto front and the nearest solutions on a reference Pareto front. The lower the value of GD, the more optimal the Pareto front solutions. The SP quality indicator [Deb et al., 2002] measures the extent of spread among the members of a Pareto front generated by a search algorithm [Deb et al., 2002]. The lower the SP values, the better spread out the search outputs.

To answer **RQ2**, we use the *RegionSize*, the *GoodnessOfFit* and the *GoodnessOfFit-crt* metrics defined below.

RegionSize measures the size of the critical regions as a percentage of the size of the entire input space. It is used to determine whether the critical regions become smaller and more specific over NSGAII-DT iterations. Let D_1 to D_n and D'_1 to D'_m be the dimensions of the input space (as defined in Section 4.2). Recall from the NSGAII-DT algorithm (line 15 in Algorithm 4) that the dimensions of a region R_i are characterized by $d_1 \times \ldots \times d_n \times d'_1 \times \ldots \times d'_m$ such that d_1 to d_n are respectively (finite) subsets of D_1 to D_n, and d'_1 to d'_n are respectively sub-intervals of the intervals D'_1 to D'_m. We define $RegionSize(R_i)$ as follows:

$$RegionSize(R_i) = \prod_{j=1}^{n} \frac{|d_j|}{|D_j|} \times \prod_{j=1}^{m} \frac{max(d'_j) - min(d'_j)}{max(D'_j) - min(D'_j)} \tag{4.2}$$

RegionSize for the entire input space is equal to one, and the lower $RegionSize(R)$, the smaller the region R. For example, for the tree in Figure 4.4(a), we have $RegionSize(\mathbf{A}) = 0.25$, and for that in Figure 4.4(b), we have $RegionSize(\mathbf{B}) = 0.02$, $RegionSize(\mathbf{C}) = 0.03$ and $RegionSize(\mathbf{D}) = 0.03$, implying that the size of critical regions are reduced over subsequent iterations of NSGAII-DT. As discussed in Section 4.3.2, in our work, input variable domains are reduced in two ways: by the explicit conditions on tree edges and due to ADAS constraints, i.e., the gray conditions in Figure 4.4(b). In order to accurately compute *RegionSize* (e.g., for **B**-**D** in Figure 4.4(b)), we consider both explicit and implicit domain reductions.

GoodnessOfFit is used to determine how well the trees generated during the search fit to the set of scenarios sampled during the search. Similarly, *GoodnessOfFit-crt* determines goodness of fit for critical scenarios only. Each decision tree is built based on a labeled set $P^+ \cup P^-$ of elements (see line 14 in Algorithm 4). The *GoodnessOfFit* of each decision tree is the number of elements in

$P^+ \cup P^-$ that are correctly classified by the tree (either as critical or non-critical) divided by $|P^+ \cup P^-|$. Similarly, the *GoodnessOfFit-crt* for each tree is the number of elements in P^- that are correctly classified by the tree (as critical) divided by $|P^-|$. Note that since we do not use the classification trees as prediction models, we do not evaluate them based on cross validation with test sets. Instead, we assess how well the trees characterize critical scenarios, while avoiding overfitting as discussed in Sections 4.4.3 and 4.4.4.

4.4.3 Experiment Design

We applied both NSGAII-DT and NSGAII to the AEB case study introduced in Section 4.1. For both algorithms, we set the (initial) population size to 100, the mutation rate to 0.11, and the crossover rate to 0.6. Specifically, the mutation rate is $1/l$ where l is the chromosome size (nine in our work). The search parameter values are consistent with existing guidelines [Arcuri and Fraser, 2011b].

We set the search time to 24 hours. Based on our experiments, the HV, GD and SP quality indicator values for both NSGAII and NSGAII-DT start to stabilize and reach a plateau within the search time budget of 24h. Further, according to domain experts, longer search time budgets are not practical in the context of ADAS testing.

To build NSGAII-DT decision trees, we use the Classification and Regression Trees (CART) [Breiman et al., 1984] algorithm. We control the decision tree size (depth) by setting the value of *minimum split parameter (msp)* to 10% of the size of the underlying data set. Our goal is thus to avoid overfitting and obtain reasonable estimates in ADAS critical regions captured by the tree leaves labeled critical. Moreover, we require that splitting a node reduces the miss-classification error of decision trees by at least 1%.

Within the 24h search time budget, NSGAII performed, on average, 22 search iterations (generations). Note that the NSGAII-DT algorithm consists of two nested loops: The outer loop that generates decision trees (lines 5–16 in Algorithm 4), and the inner loop that invokes NSGAII for critical input space regions (lines 6–10 in Algorithm 4). We refer to each iteration of the outer loop as *tree generation*. Corresponding to each tree generation, NSGAII is invoked one or more times depending on the number critical tree leaves. We set the number of search iterations performed by each NSGAII invocation to five (i.e., we set $g = 5$ in Algorithm 4). By setting $g = 5$, NSGAII-DT performed between five to seven *tree generations* in 24h (i.e., each run of NSGAII-DT generated between five to seven trees). Further, on average, NSGAII-DT performed 30 search iterations (i.e., NSGAII iterations) in 24h. Note that in our experiments, each run of NSGAII-DT performed more search iterations than each run of NSGAII. This is because, in our experiments and within the 24h search time budget, most search iterations of NSGAII-DT are applied to population sets smaller than the initial population set, while all search iterations of NSGAII are applied to a fixed-size population set equal to the size of the initial population. We reran each of the NSGAII and NSGAII-DT algorithms for 15 times to account for their randomness. We have made our experimental results available at [data, 2017].

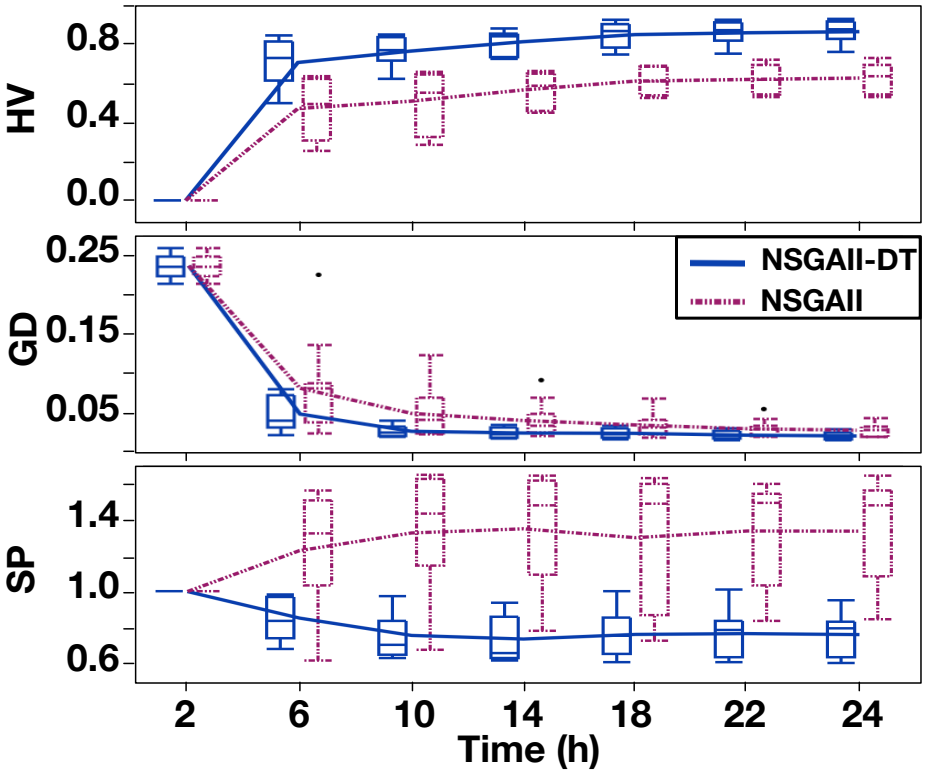

Figure 4.5. Comparing HV, GD and SP values obtained by NSGAII and NSGAII-DT.

4.4.4 Results

RQ1. Figure 4.5 shows the HV, GD and SP values computed based on the outputs of NSGAII-DT and NSGAII. We show the results at every four-hour time interval starting at 2h as well as the results at the end of the search time limit (i.e., at 24h). Note that, on average, simulating the elements in the initial population takes about 2h. Hence, the results at 2h are those obtained for the randomly selected initial population and prior to any search iteration. As shown in the figure, the HV, GD and SP values for NSGAII-DT are consistently better than those for NSGAII. Further, after executing the algorithms for about 22h, both NSGAII-DT and NSGAII converge towards their respective Pareto optimal solutions (i.e., for each algorithm, the differences in HV, GD and SP average values between 22h and 24h are negligible).

Following existing guidelines [Arcuri and Briand, 2014], to statistically compare HV, GD and SP values, we use the nonparametric pairwise Wilcoxon rank sum test [Capon, 1991] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [Vargha and Delaney, 2000]. The level of significance (α) is set to 0.05. Table 4.1 reports the statistical test results comparing NSGAII-DT and NSGAII at 24h. For HV and SP comparisons, the p-values are less than 0.05, and the $\hat{A}_{12}$ values show large effect sizes. The differences between the GD distributions of NSGAII-DT and NSGAII are not statistically significant, although the effect size value is in the medium range. However, as shown in Figure 4.5, the medians and averages of the GD values obtained by NSGAII-DT are lower (i.e., better) than the medians and averages of the GD values obtained by NSGAII.

Table 4.1. Statistical test results for NSGAII-DT and NSGAII at 24h (the format is: metric (p-value / $\hat{A}_{12}$)).

HV (0.01 / 0.9), GD (0.07 / 0.3), SP (0.01 / 0.1)

Finally, we evaluate the results of NSGAII-DT and NSGAII based on the number of *distinct, critical* test scenarios generated by each algorithm. Recall that an AEB test scenario is a vector in the AEB input space (i.e., a vector of values to four static and five dynamic variables). Also, an AEB test scenario is critical if its simulation outputs satisfy the *CB* property (equation 4.1 in Section 4.2). Two AEB test scenarios are distinct if they differ in the value of at least one static variable or in the value of at least one dynamic variable with a significant margin.

Over the 15 runs, NSGAII generates 708 distinct AEB test scenarios among which 411 are critical. In contrast, over the 15 runs, NSGAII-DT generates 1045 distinct AEB test scenarios among which 731 are critical. This result shows that, within the same search time budget, on average, NSGAII-DT provides 78% more distinct, critical test scenarios compared to NSGAII, enabling the engineers to better identify the limitations of AEB.

The answer to ***RQ1*** is that, NSGAII-DT significantly outperforms NSGAII. Further, on average, NSGAII-DT generates 78% more distinct, critical test scenarios compared to NSGAII.

RQ2. To answer this question, we focus on assessing the critical regions characterized by NSGAII-DT (i.e., the algorithm that is shown, in **RQ1**, to outperform NSGAII). We further note that NSGAII, or any search algorithm for that matter, has never been used to characterize critical regions and cannot be used as a baseline of comparison for this research question.

Figure (a) shows the *RegionSize* values for the critical regions obtained from the decision trees generated by NSGAII-DT. Recall that NSGAII-DT performs five to seven *tree generations* within the 24h search time limit. In our experiments, each decision tree generated by NSGAII-DT had between one and three critical leaves (i.e., critical regions). As shown in the figure, the critical regions generated by NSGAII-DT become monotonically smaller (i.e., more specific) over successive tree generations. In particular, the critical regions obtained from the first decision trees are on average 17.2% of the size of the entire input space, while the final trees generated by NSGAII-DT are on average 3.5% of the input space.

Figures (b) and (c) show the *GoodnessOfFit* and the *GoodnessOfFit-crt* values for NSGAII-DT decision trees, respectively. As shown in the figure, *GoodnessOfFit* increases from 57% to 77%, and *GoodnessOfFit-crt* increases from 50% to 89% over the maximum seven tree generations of NSGAII-DT. These results show that the decision trees generated by NSGAII-DT, compared to those generated based on random initial populations, accurately classify on average 20% more critical and non-critical scenarios, and almost 40% more *critical* scenarios. Hence, NSGAII-DT, over its successive tree generations, produces decision trees that fit noticeably better to *critical* scenarios.

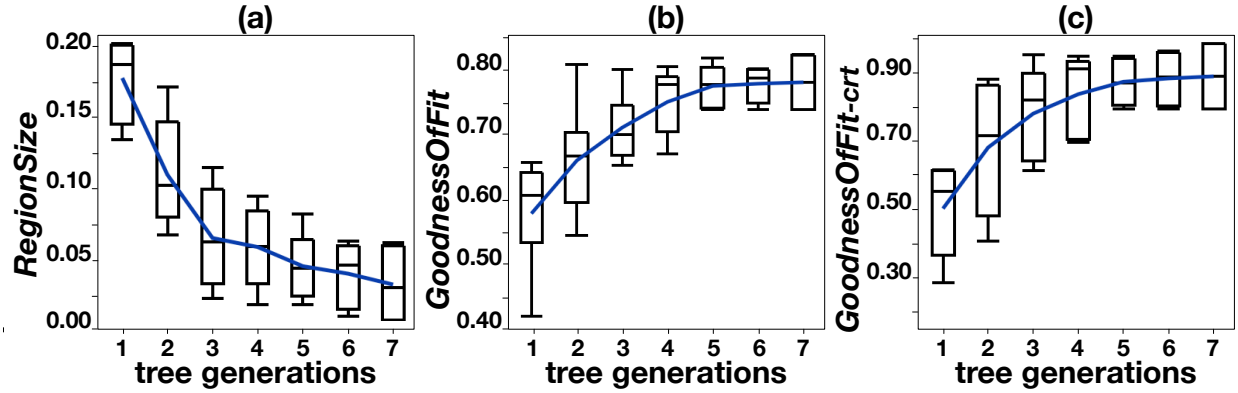

Figure 4.6. Evaluating the critical regions: (a) the *RegionSize*, (b) the *GoodnessOfFit*, and (c) the *GoodnessOfFit-crt* values.

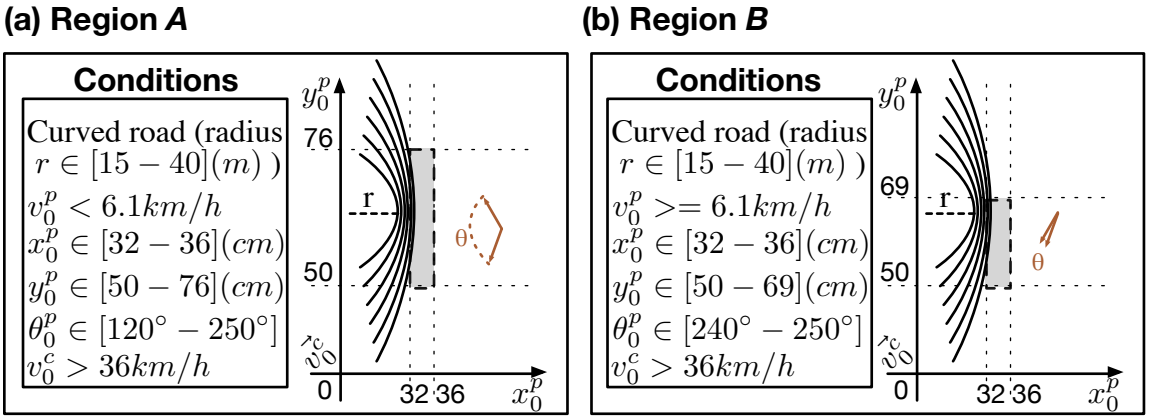

Figure 4.7. Examples of critical regions for the AEB case study

The answer to ***RQ2*** is that the *RegionSize*, *GoodnessOfFit* and *GoodnessOfFit-crt* values monotonically improve across different tree generations, confirming that the generated critical regions consistently become smaller, more homogeneous and more precise over successive tree generations of NSGAII-DT. In particular, the trees generated by NSGAII-DT, compared to those generated based on the initial randomly selected populations, fit on average to 40% more *critical* AEB test scenarios.

Benefits from a practitioner's perspective. Here, we investigate whether practitioners are able to use and benefit from our approach. In particular, we intend to know whether the critical regions computed by our approach are understandable, informative, and useful to practitioners. To do so, we draw on the qualitative reflections of two *semi-structured interview* [Wohlin et al., 2012] sessions that we conducted with three senior engineers at IEE. The reflections are based on the comments the engineers made in two two-hour meetings with the researchers. The engineers were selected from three different groups at IEE working on different aspects of ADAS development and testing. We have been collaborating with one of these engineers on the research and the case study presented in

this chapter. The two other engineers, however, did not have any interaction with the researchers prior to the interview sessions. Further, they were not involved in our research nor in the development of the AEB case study or any of our other ADAS examples.

To perform the interviews, we selected, among the decision trees generated by NSGAII-DT in our experiments, the one with the highest goodness of fit. The selected tree characterized three critical regions in the AEB input space. We created visually-enhanced representations of the three regions, showing the regions individually without any reference to the tree structure. Figures (a) and (b) illustrate the representations for two of the regions. The conditions specifying each region are shown on the left side of each region diagram. Furthermore, some of these conditions (i.e., those on the road type, and the initial position and orientation of the pedestrian) are visually shown on the right side of each diagram. Region **A** specifies the AEB input scenarios where a car (speed $>$ 36.6km/h) drives on a curved road with a radius between 15m to 40m, while a pedestrian starts walking from a point inside the dashed gray rectangle with a trajectory between 120° and 250° and crosses the road with a low speed ($<$ 6.1km/h). Region **B** specifies similar scenarios as those in **A** except that the pedestrian walks with a high speed ($>=$ 6.1km/h) within a much narrower trajectory and starts crossing the road from a slightly smaller area compared to the one in **A**.

During the meetings, we presented the critical regions to the engineers, and asked the following questions: (1) Are you able to understand the conditions specifying the regions? (2) Based on your domain knowledge, do you think the regions specify situations where AEB is more likely to fail (i.e., exhibits *CB*)? (3) How can you utilize the knowledge you gain from the characterizations of the regions to analyze AEB? These questions aim to assess, respectively, comprehension, intuitiveness and usefulness of the critical region characterizations generated by our approach.

Regarding comprehension and intuitiveness, the engineers noted that the characterizations of the regions are understandable and consistent with their intuition. For example, regions **A** and **B** indicate that scenarios containing curved roads are more likely to exhibit *CB*. This is because, on such roads, pedestrians appear relatively late in the camera's field of view and will be detected late by AEB, hence leaving little time to apply the brake. The regions further show that the probability of *CB* is higher when the car speed is higher than 36.6km/h. Finally, the regions show that, in addition to the road and vehicle characteristics, *CB* likely happens due to pedestrian dynamics. Specifically, critical scenarios are more likely when pedestrians walk from particular areas on the sidewalk, or as shown in **B**, running pedestrians with a particular trajectory (θ) are more likely to escape accidents if they do not run towards the car.

Regarding the usefulness of our approach, the engineers noted that the information captured by these regions can help them in the following ways: (1) *Debugging the system or the simulator.* The region characterizations, particularly when they do not match the domain knowledge, may point to errors in the system or the simulator. For example, in our early results, curved roads did not appear as critical regions. Our investigation showed that due to an error in the AEB sensor output (i.e., the TTC output), which resulted in some scenarios that actually led to collision to be wrongly labeled as non-critical. Further, the pedestrian dynamic situations identified as critical may point to weaknesses

in pedestrian tracking algorithms [Philomin et al., 2000] typically used in ADAS. (2) *Identifying changes to hardware components to help increase ADAS safety.* For example, in our work, we assume AEB contains one camera located at the front center of the car with a specific value for its field of view. Regions **A** and **B** indicate that a different type of camera with a larger field of view or two cameras, although more expensive, may help detect pedestrians faster and better on curved roads. (3) *Identifying proper warnings to drivers.* Some of ADAS critical behaviors may not be avoidable due to real world and physical constraints. Nevertheless, our approach enables car makers to be aware of such situations and consider mitigation strategies. For example, regions **A** and **B** indicate that AEB may not be fully trusted on curved roads in residential zones where it is more likely for pedestrians to cross roads. In such situations, a warning message may be shown to drivers to reduce their speed (e.g., to lower than 30km/h).

4.4.5 Threats to validity

To mitigate the *Internal validity* risks caused by confounding factors, we compared NSGAII-DT and NSGAII under identical parameter settings. Further, we present a detailed formal description of our case study and search algorithm, and provide all the parameter settings to facilitate reproducibility. Our case study is a real ADAS. The simulation data is obtained based on an industrial and widely-used ADAS simulation tool. To assess usefulness of our approach, we conducted two semi-structured interview sessions with three engineers from different groups at IEE with varying types of expertise related to ADAS development.

Conclusion validity is related to random variations and inappropriate use of statistics. To mitigate these threats, we have followed standard guidelines in search-based software engineering [Arcuri and Fraser, 2011b] and ran the search algorithms 15 times. Further, we use the non-parametric pairwise Wilcoxon Paired Signed Ranks test and Vargha and Delaney's $\hat{A}_{12}$ for statistical testing and effect sizes.

The main threat to *construct validity* concerns unsuitable or incorrect metrics. To compare multi-objective search algorithms we use standard quality indicators (i.e., HV, GD, SP). Further, we assess the decision trees generated by our approach using our formally defined *RegionSize* and the standard *GoodnessOfFit* metrics.

Regarding the *external validity* threats, we note that we provide in Section 4.2 a precise formalization of the ADAS systems to which our testing approach is applied. Our ADAS formalism builds on our experiences of studying different ADAS systems as well as the characteristics of the PreScan simulator. Our testing approach applies to any ADAS system that conforms to our formalism presented in Section 4.2. Finally, we note that ADAS systems comprise an important and growing industry sector with pressing needs regarding testing and verification.

4.5 Conclusions

We proposed a simulation-based testing algorithm for vision-based control systems (i.e., ADAS). Our algorithm builds on learnable evolution models and uses classification decision trees to guide the generation of new test scenarios within complex and multidimensional input spaces. Our approach is evaluated on an industrial ADAS. The results indicate that our classification-guided search algorithm outperforms a baseline evolutionary search algorithm and generates 78% more distinct, critical test scenarios compared to the baseline algorithm. Our approach, further, characterizes critical regions of the ADAS input space. Based on our interviews with domain experts, such characterizations are accurate and help engineers debug their systems. They further help engineers identify environment conditions that are likely to lead to ADAS failures as well as hardware changes that can increase ADAS safety.

Chapter 5

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

In this chapter, we focus on identifying feature interaction failures in self-driving systems. Complex systems such as autonomous cars consist 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 [Fisler and Krishnamurthi, 2005, Prehofer, 1997, Jackson and Zave, 1998]. 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. 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 [Jackson and Zave, 1998, Calder et al., 2003, Braithwaite and Atlee, 1994, Apel et al., 2013a]. Some techniques focus on identifying feature interactions at the requirements-level by analysis of formal or semi-formal requirements models [Zave, 1993, Bredereke, 2000, Blom et al., 1994]. Several techniques detect feature interaction errors in implementations using test cases derived from feature models capturing features and their dependencies [Oster et al., 2011, Patel et al., 2013, Ferber et al., 2002, Apel et al., 2013b]. Other approaches devise design and architectural resolution strategies to eliminate at runtime undesired feature interactions identified at the requirements-level [Hay and Atlee, 2000, van der Linden,

1994, Jackson and Zave, 1998, Zibaeenejad et al., 2017]. 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.

In this chapter, 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 [Zander et al., 2017, Wainer, 2009, Nise, 2004]. 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 chapter are specified in the Matlab/Simulink language [Matlab, 2019].

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 [Matinnejad et al., 2016, Matinnejad et al., 2015, Bühler and Wegener, 2008, Abbas et al., 2013] as well as various other testing problems such as unit testing [McMinn, 2004a, Tonella, 2004, Fraser and Arcuri, 2013a], regression testing [Li et al., 2007, Yoo and Harman, 2007] and optimizing machine learning components [Suttorp and Igel, 2006].

This chapter highlights the following research contributions:

1. 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* [McMinn, 2004a] 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.
2. 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.

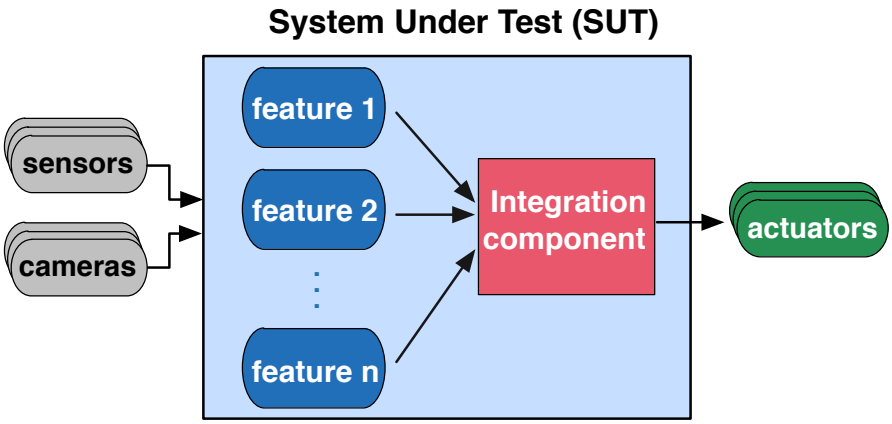

Figure 5.1. Overview of a typical function model capturing the software subsystem (SUT) of a self-driving car.

3. We evaluate FITEST using two industrial self-driving systems from our partner company IEE. Both systems represent a (partial) self-driving car consisting of four features.

Organization. This chapter is structured as follows. Section 5.1 motivates our work. Section 5.2 presents our approach. Section 5.3 describes our evaluation, and Section 5.4 concludes this chapter.

5.1 Motivation

Figure 5.1 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.

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 chapter.

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 5.1) such that the interactions do not lead to failures (e.g., using existing feature interaction resolution techniques [Jackson and Zave, 1998, Zibaeenejad et al., 2017]). 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 devised by engineers satisfy system safety requirements under different environment conditions. We note that in Section 5.2.3, 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.

5.2 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 5.2.1 describes how we integrate the function models into a high-fidelity,

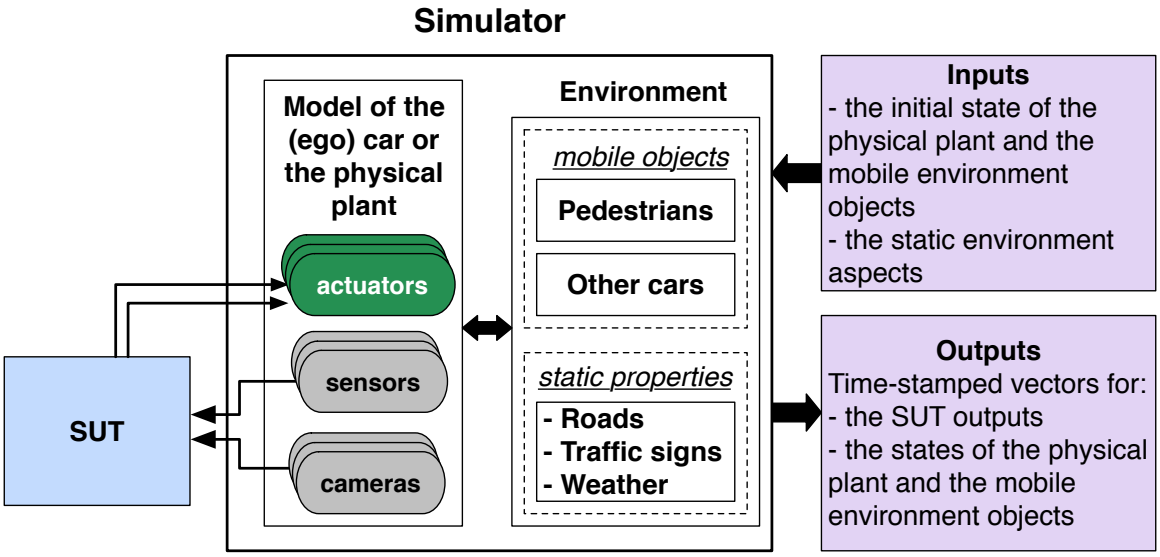

Figure 5.2. Early testing of control system function models using simulators.

physics-based simulator for self-driving systems. Section 5.2.2 characterizes the test inputs and outputs for self-driving systems. Section 5.2.3 introduces our hybrid test objectives. Section 5.2.4 presents FITEST, our proposed many-objective test generation algorithm that utilizes our test objectives to generate test inputs revealing feature interaction failures.

5.2.1 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 5.1, 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 5.2). 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 the PreScan simulator. Some examples of *SafeDrive* simulations are available online [Ben Abdessalem, 2018a].

5.2.2 Test Inputs and Outputs

The test inputs for a self-driving system are the inputs required to execute the simulation framework in Figure 5.2. For example, to 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

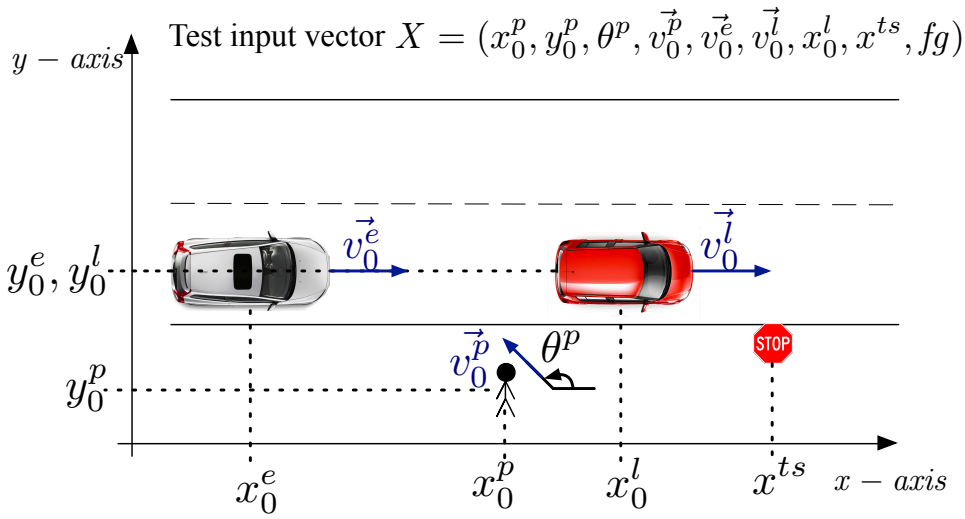

Figure 5.3. Test inputs required to simulate *SafeDrive*, our case study system.

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 5.3. 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: $sd - \varepsilon \leq x_0^l - x_0^e \leq sd + \varepsilon$ and $|v_0^e - v_0^l| \leq \varepsilon'$ 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 5.2, 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

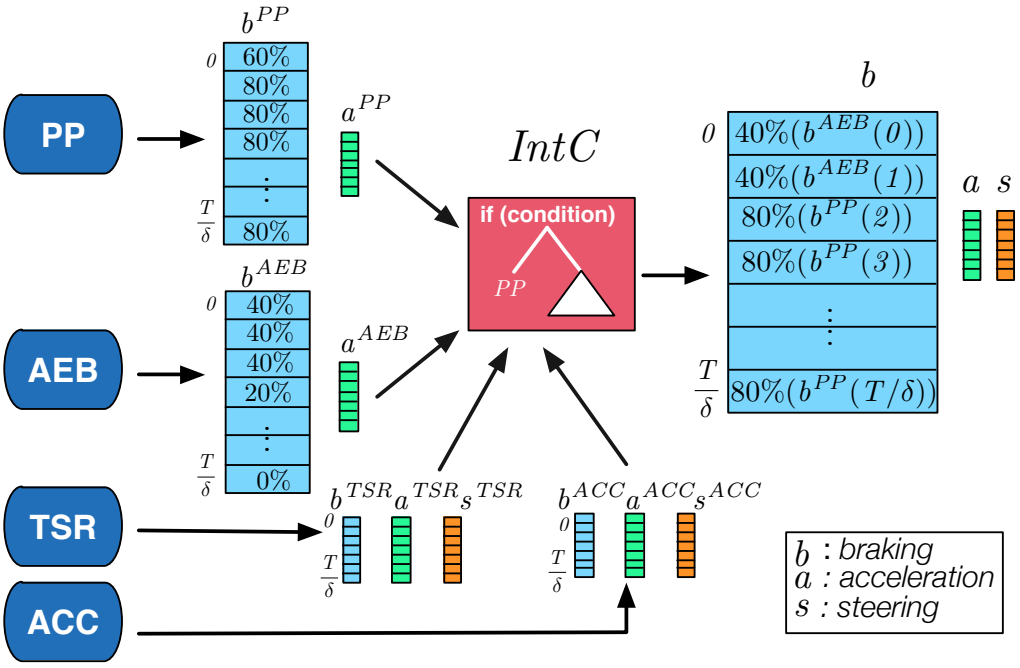

Figure 5.4. Actuator command vectors generated at the feature-level and at the system-level by simulating *SafeDrive*. Vectors b^f, a^f and s^f indicate command vectors generated by feature f for the braking, acceleration and steering actuators, respectively. The *IntC* component analyzes the command vectors generated by all the features and issues the final command vectors b, a and s to the braking, acceleration and steering actuators, respectively.

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 5.4 illustrates the SUT outputs generated by simulating *SafeDrive*. Specifically, the 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.

5.2.3 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 chapter 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

Table 5.1. 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 $FD_2(i)$.

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 5.1 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 5.2). A test case for $\mathcal{F}$ is a vector X of inputs required to execute the simulation framework into which $\mathcal{F}$ is embedded (e.g., Figure 5.3 shows the test input vector for *SafeDrive*). The test output of $\mathcal{F}$ includes: (1) a vector v_{act}^{f} 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 [McMinn, 2004a, Harman et al., 2012b] 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 [McMinn, 2004a]. 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 [McMinn, 2004a] and the normalized *branch distance* [McMinn, 2004a, Fraser and Arcuri, 2013a]. 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 5.1 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 5.1 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 5.4, 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 5.1 compared to a firm and early braking. Dually, the requirements in Table 5.1 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 [Korel, 1990] 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:

$$UOD^{act}_f(i) = \begin{cases} v_{act}(i) - v^f_{act}(i), & \text{if } v^f_{act}(i) < v_{act}(i) \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

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 [Korel, 1990], we define UOD^{act}_f for this kind of actuators as follows:

$$UOD^{act}_f(i) = \begin{cases} v^f_{act}(i) - v_{act}(i), & \text{if } v_{act}(i) < v^f_{act}(i) \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

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 [Arcuri, 2013] 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:

$$\Omega_{j,l}(i) = \begin{cases} \overline{BD}_j(i) + \overline{UOD}_{max} + \overline{FD}_{max} & \text{(1) If } j \text{ is not covered } (\overline{BD}_j(i) > 0) \\ \overline{UOD}_f(i) + \overline{FD}_{max} & \text{(2) If } j \text{ is covered, but } f \text{ is not unsafely overridden } (\overline{BD}_j(i) = 0 \wedge \overline{UOD}_f(i) > 0) \\ \overline{FD}_l(i) & \text{(3) Otherwise } (\overline{BD}_j(i) = 0 \wedge \overline{UOD}_f(i) = 0) \end{cases} \tag{5.3}$$

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.

5.2.4 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 $\Omega = \left\{\Omega_{1,1}, \ldots, \Omega_{k,n}\right\}$ *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

Algorithm 5: 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
```

a class of search algorithms suitably defined for problems with more than three objectives. Various many-objective meta-heuristics have been proposed in the literature, such as NSGA-III [Deb and Jain, 2014], HypE [Bader and Zitzler, 2011]. These algorithms are designed to produce different alternative trade-offs that can be made among the search objectives [Li et al., 2015].

Recently, Panichella et al. [Panichella et al., 2015, Panichella et al., 2018] 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 [Panichella et al., 2015], which is discussed in Section 2.1.2. MOSA (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 [Panichella et al., 2015, Panichella et al., 2018].

In this chapter, 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 5. We then discuss the differences between FITEST and MOSA.

Main loop. As Algorithm 5 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 5.3). After simulating each test $X \in P$, the test objectives $\Omega_{j,l}$ for X are computed based on the simulation results (see Section 5.2.3). Next, tests are evolved through subsequent iterations (loop in lines 7-16), called *generations*. In each generation, the *binary tournament selection* [Deb et al., 2002] 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 [Luke, 2013], 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 5.3), we use two widely-used genetic operators proposed for real number solution encodings: the *simulated binary crossover* [Deb, 1995] (SBX) and the *gaussian mutation* [Deb and Deb, 2014]. Prior studies [Herrera et al., 2003, Deb and Deb, 2014] 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 [Fraser and Arcuri, 2013a] to handle different types of test data, e.g., strings, Java objects, etc.

Correction operator. Recall from Section 5.2.2 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 5). 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 5.2.2, 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* [Panichella et al., 2015, Panichella et al., 2018, Rojas et al., 2017] (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 5) 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* [Panichella et al., 2015]). 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* [Deb et al., 2002, Deb, 2014] (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.

5.3 Evaluation

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

5.3.1 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 5.2.3, *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 [McMinn, 2004a, Fraser and Arcuri, 2013a, Fraser and Arcuri, 2015]. For example, Fraser and Arcuri [Fraser and Arcuri, 2015] 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 [Briand et al., 2006a, Afzal et al., 2009]. 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 5.2.3 and are computed as the sum of the approach level [McMinn, 2004b] and the normalized branch distance [McMinn, 2004b]. 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 5.2.3. 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

$$Fail_{j,l}(i) = \begin{cases} \overline{BD}_j(i) + \overline{FD}_{max} & \text{if } j \text{ is not covered} \\ \overline{FD}_l(i) & \text{otherwise} \end{cases} \tag{5.4}$$

In this chapter, 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 5.2.4, our changes to MOSA are primarily motivated by the practical needs of (1) using genetic operators for numerical vectors (often called real-coded operators [Herrera et al., 2003, Deb and Deb, 2014]) 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 [Panichella et al., 2015] 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 [Campos et al., 2017], whole suite search [Campos et al., 2017, Panichella et al., 2015], and other many-objective evolutionary algorithms [Panichella et al., 2018].

5.3.2 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 5.1. 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 chapter. We refer to these systems as *SafeDrive1* and *SafeDrive2*.

5.3.3 Experimental Settings

For the genetic operators used in FITEST, we use the parameter values suggested in the literature [Cobb and Grefenstette, 1993, Briand et al., 2006b, Deb et al., 2002]: We use the *simulated binary crossover* (SBX) with a crossover probability 0.60, as the recommended interval is $[0.45, 0.95]$ [Cobb and Grefenstette, 1993, Briand et al., 2006b]. 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 [Deb et al., 2002], 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 5.2.4, 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 5.3.4, 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) $\times$ 2 (systems) $\times$ 3 (test objectives) $\times$ 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 5.1 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.

5.3.4 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 5.5(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, 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,

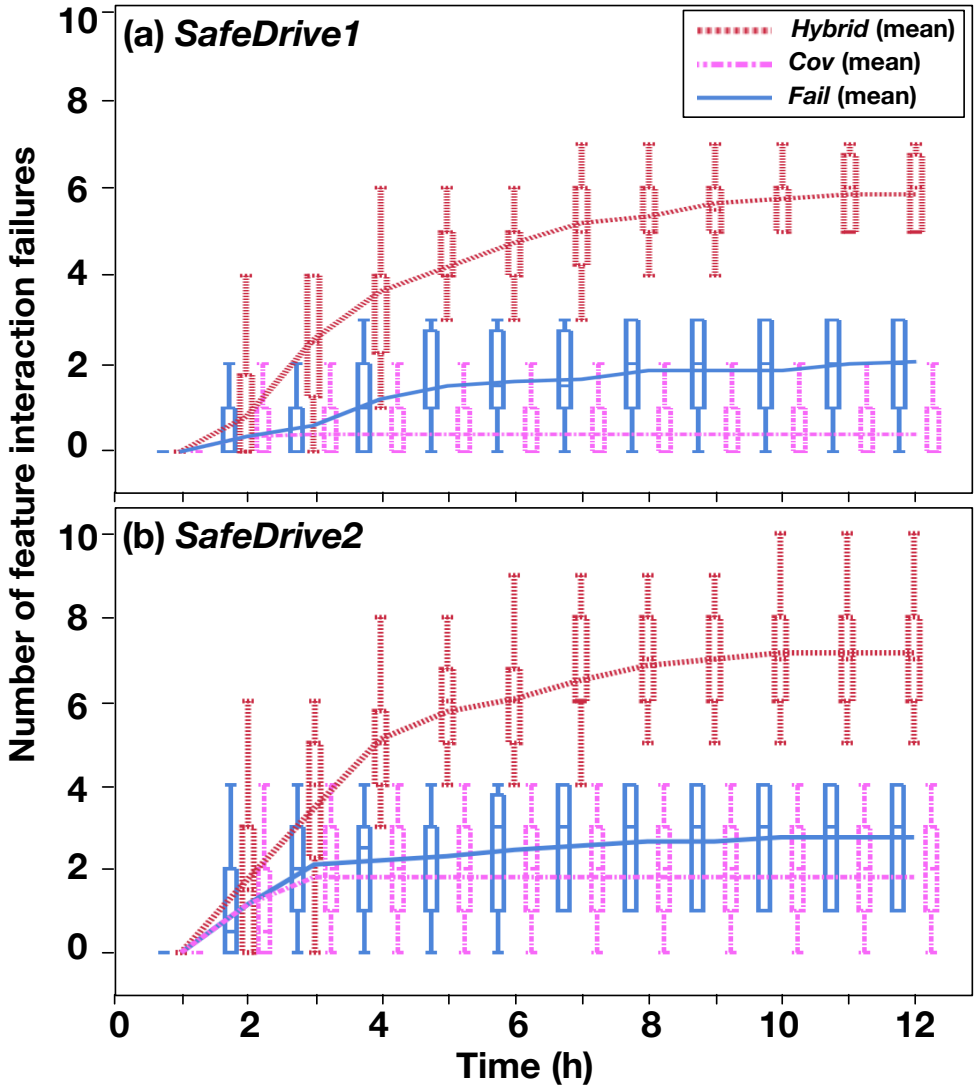

Figure 5.5. The number of feature interaction failures found by *Hybrid*, *Fail* and *Cov* over time for (a) *SafeDrive1* and (b) *SafeDrive2* systems.

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 5.2.4, *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

Table 5.2. 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 5.5).

	SafeDrive1				***SafeDrive2***			
	Hybrid* vs. *Cov		***Hybrid* vs. *Fail***		***Hybrid* vs. *Cov***		***Hybrid* vs. *Fail***	
time	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$	**p-value**	$\hat{\mathbf{A}}_{\mathbf{12}}$
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)

hence, we include all of them in our search.

We compare the results in Figure 5.5 using a statistical test. Following existing guidelines [Arcuri and Briand, 2014], we use the non-parametric pairwise Wilcoxon rank sum test [Capon, 1991] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [Vargha and Delaney, 2000]. Table 5.2 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*.

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 chapter. 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 5.1 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 [Ben Abdessalem, 2018a].

5.4 Conclusions

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 [Panichella et al., 2015, Panichella et al., 2018] 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. 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.

Chapter 6

Automatic Localization and Repair of Feature Interaction Failures

In the previous chapter, we identified feature interaction failures. These failures may occur when the feature interaction resolution rules or their implementation are erroneous. In this chapter, we propose a strategy to identify errors in the feature interaction resolution rules for self-driving systems and to automatically repair these errors. Our approach localizes errors by focusing on the rules that are related to the most severe failures. Then, our approach generates patches by using a mutation-based evolutionary algorithm.

We cast the problem of repairing decision rules into a search-based problem. Search-based methodologies have been successfully applied to repair faults in software code [Le Goues et al., 2012, Kim et al., 2013, Arcuri and Yao, 2008, Qi et al., 2014] . Most of the existing approaches [Le Goues et al., 2012, Kim et al., 2013, Arcuri and Yao, 2008] use Genetic Programming (GP) and rely on a single fitness function to evaluate each patch. GP maintains a population of individual patches, where each patch evaluation requires to simulate the test suite. In our work, executing simulation scenarios is computationally expensive. Hence, a population-based algorithm is not efficient for our context purpose. Correct decision rules should satisfy the safety requirements of self-driving systems. In our work, we define many objectives, where each objective is related to one safety requirement. In this chapter, we propose a many-objective single-state search algorithm, which uses an archive to keep track of partial patches.

This chapter reports on the following research contributions:

1. We propose new mutation operators that are specifically designed to address the specificities of the errors in the decision rules for self-driving systems.
2. We introduce RUFI (Repair Undesired Feature interaction Interactions), a new many-objective repair algorithm to repair decision rules of self-driving systems, and hence, to avoid undesired feature interactions. RUFI localizes the errors and then uses the mutation operators to generate patches.

3. We evaluate RUFI on an industrial automotive system consisting of four features.

Organization. This chapter is structured as follows. Section 6.1 motivates our work. Section 6.2 discusses why existing program repair techniques are unlikely to effectively repair errors in decision rules of self-driving systems. Section 6.3 presents our approach. Section 6.4 describes our evaluation, and Section 6.5 concludes this chapter.

6.1 Motivation

We motivate our work using the self-driving system case study, *SafeDrive*, which is presented in the previous chapter (Chapter 5). Recall that SafeDrive contains four self-driving features: Autonomous Cruise Control (ACC), Traffic Sign Recognition (TSR), Pedestrian Protection (PP), and Automated Emergency Braking (AEB). All the features generate braking and acceleration commands to respectively control the brake and the throttle actuators. TSR and ACC, additionally, generate steering commands.

Each SafeDrive feature receives its inputs from sensors/cameras and sends its outputs to actuators. Both inputs and outputs are sequences of timestamped values. The system 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. For example, 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. In this scenario, ACC and PP send conflicting outputs to actuators. We denote this scenario by *Scenario-1*.

Since features may issue conflicting outputs, engineers have to develop algorithms to resolve potential conflicts between feature outputs and decide the most appropriate actuator command at each time step. This requires developing complex rules that determine what feature output should be prioritized at each time step based on the environment factors as well as other conditions. Figure 6.1 shows a small subset of such rules for the SafeDrive system. The left side of each rule shows a conjunction of conditions on input or other system variables, and the right side shows the feature whose command should be applied when the conditions on the left hold. For example, rule 1 states that feature PP should be applied when a detected objective is a pedestrian, and the time to collision (TTC) and the distance between a vehicle and a pedestrian ($Dist(P/Car)$) variables are respectively less than their corresponding thresholds, denoted by TTC_{th} and $Dist_{th}$. Note that TTC 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 [van der Horst and Hogema, 1993], and $Dist(P/Car)$ is the distance between the car and the pedestrian during the simulation time. Dually, rule 4 in Figure 6.1 states that if a detected object is not a pedestrian, but there is still a risk of collision (i.e., TTC is less than its threshold), then AEB should be applied. Note that all the rules in Figure 6.1 are expected to be

1. if $(TTC < TTC_{th}) \wedge (Dist(P/Car) < Dist_{th}) \wedge$ (object detected is pedestrain)	$\Longrightarrow$	PP
2. if (none of TSR rules are activated) $\wedge$ (speed of the car $<$ speed of the leading car)	$\Longrightarrow$	ACC
3. if (speed of the car $>$ speed limit)	$\Longrightarrow$	TSR
4. if $(TTC < TTC_{th}) \wedge$ (object detected is not pedestrain)	$\Longrightarrow$	AEB
5. if $Dist(P/Car) < Dist_{th}$	$\Longrightarrow$	PP
6. if (none of PP rules are activated) $\wedge$ (there is a STOP sign)	$\Longrightarrow$	TSR

Figure 6.1. Decision rules that determine which feature output should be applied at each time step depending on the environment and other conditions.

executed periodically at regular time steps, and all the variables in the rules such as TTC, $Dist(P/Car)$ and car-speed specifically refer to the values of these variables at the current time instant.

Developing rules to resolve conflicts is a difficult task and requires extensive domain expertise and a thorough analysis of system requirements. In general, we can never be sure if the set of conflict resolution rules is complete as we can never know if we have a complete set of system requirements [van Lamsweerde, 2009]. But apart from the requirements incompleteness challenge, the developed rules or their implementation might be erroneous as well. In particular, we identify two general ways where the rules may be wrong, and hence lead to unsafe self-driving systems: (1) The conjunctive conditions on the left side of the rules may be wrong. Specifically, there might be missing clauses in the conditions, the thresholds used in each clause may not be accurate and there might be errors in mathematical or relational operations of the clauses (i.e., using $<$ instead of $>$ or $+$ instead of $-$). (2) The second issue arises in determining the order of conflict resolution rules. As the rules in Figure 6.1 show, there is already a partial order over such rules implied by the logical implication. For example, the condition of rule 1 implies the condition of rule 5 (i.e., rule 1 is more specific than rule 5). Hence, rule 1 has to be checked before rule 5 since otherwise, rule 1 would be unreachable. Some rules are also mutually exclusive (e.g., rule-1 and rule-4), and hence, they can be applied in any order. However, for some other rules, different rule orderings lead to different system behaviors and it is not clear what order should be used. For example, rule-1 and rule-2 can be applied in two different orders leading to two different system behaviors, one of which prioritizes PP and the other prioritizes ACC when both of their left conditions hold. In such situations, engineers have to opt for one option based on their domain knowledge. But the selected order may not be safe and may lead to unknown of unforeseen errors. For example, for Scenario-1, when rule 2 is checked before rule 1, the output of ACC is sent to the actuator. Hence, the car accelerates and hits the pedestrian. There are two cases where rule 2 is checked before rule 1: either (1) rule 2 is placed before rule 1 in the ordered set of rules, or (2) rule 1 is placed before rule 2, but one the thresholds in rule 1 makes the rule invalid (e.g., TTC_{th} is too small).

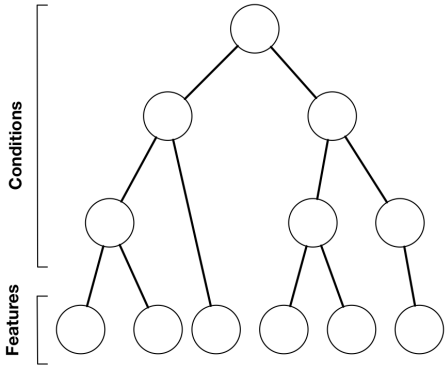

Figure 6.2. Decision rules structure.

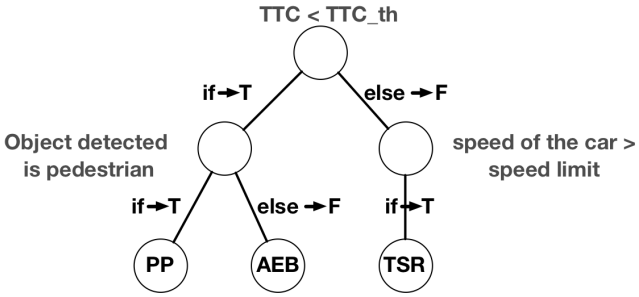

Figure 6.3. Example of decision rules structure.

The implementation of the decision rules (referred to as *IntC* is Chapter 5) is a sequence of ordered conditional statements (e.g., if-then-else statements). The structure of the rules can further be represented as a tree. For example, Figure 6.2 shows the structure of the rules as a tree. Each leaf node in the tree is a feature. The non-leaf nodes represent the conditions under which a feature should be applied. We represent the rules as a set of paths $\Pi = \{\pi^1 \dots \pi^m\}$. The last node of each path is the leaf representing the feature applied by that path. The middle nodes of each path correspond to the prerequisite conditions that should hold to apply that feature. Figure 6.3 shows a simplified example of *IntC* tree structure. The path on the left shows that when $TTC < TTC_{th}$ and the object detected is a pedestrian, PP should be applied.

6.2 Program Repair: State-of-the-art

Studies have shown that most software bug fixing tasks are manual, time-consuming, and tedious [Tom et al., 2014, Britton et al., 2019, Software, 2014]. Various techniques related to automated program repair have been proposed in the literature to help automate these tasks [Weimer et al., 2009, Kim et al., 2013, Qi et al., 2014, Gazzola et al., 2017]. The key objective of these techniques is to try to automatically repair software systems with minimal human intervention. The inputs to these techniques are a faulty program, a set of passing test cases that encode desired program behavior, and a

failing negative test case that demonstrates a defect. These techniques consist of three steps: (1) Fault localization [Xie et al., 2013, Jin and Orso, 2013] that identifies the locations where a patch could be applied. (2) Patch generation that modifies the software in the code locations returned by the fault localization step, and (3) Patch validation that checks if the synthesized patch has actually repaired the software.

Traditional automated program repair techniques use GP to search for a correct version of the faulty program. Fault Localization techniques, used by classical program repair, rely on statistical debugging [Renieres and Reiss, 2003] to localize faults. A well-known fault localization approach (Tarantula [Jones et al., 2002]) and GP are discussed in Section 2.3. Most of traditional program repair approaches are based on the following assumptions:

- A faulty program includes a single fault representing one program element (i.e., statement s) in the code. They do not support multi-location bug fixing [Qi et al., 2015].
- Test execution time is usually negligible (i.e., fitness evaluation is inexpensive).
- All failures are equally critical. Traditional fault localization techniques rely on the number of statements executed by passing and failing test cases to localize faults.
- Fitness evaluation requires executing the faulty program once.

Traditional program repair cannot be used in our context because (1) *IntC* includes multiple failures in different locations of the code. (2) Fitness evaluation is computationally expensive. Specifically, each system simulation takes around 2 min to run. This is because the physics-based simulator for self-driving systems builds on high-fidelity mathematical models. Using GP is not adequate in our context since for a population of 30 test cases, one single fitness evaluation requires 1 hour to execute. (3) The system runs iteratively at regular time steps. Every statement is executed multiple times by the same test case. When we run a test case, one decision rule may be executed at each time step. Therefore, each test case generates several execution traces, one per each time step. When we run a test suite, we thus obtain a large number of traces at each time step. Notice that many of these traces can execute the same statements. As a result, traditional statistical debugging, discussed in Section 2.3, may lead to many statements having the same suspiciousness scores. Therefore, in our context, traditional statistical debugging techniques do not provide effective guidance to localize faults.

6.3 Approach

In this section, we present our approach, called RUFI, to locate and repair faults in the decision rules of self-driving systems. The overview of our approach is shown in Figure 2.4. The inputs to RUFI are (1) a faulty *IntC* and (2) a test suite *TS* verifying the safety requirements of the system. Section 6.3.1 describes in more detail the inputs of our approach. The output is a repaired *IntC*. To address the specificities of the faults described in Section 6.1, the steps in Figure 2.4 are different from the state of the art in program repair as detailed in the following subsections.

6.3.1 Inputs

6.3.1.1 Faulty *IntC*

The goal of *IntC* is to select a feature output f to be executed by the car at every time step. Examples of these feature outputs are a braking command issued by PP or an acceleration command issued by ACC. A faulty IntC wrongly selects the feature output such that some safety requirement of feature f is violated. In most cases, such faults are created because an engineer wrongly defines some rules or makes mistakes in defining the order of the rules, i.e., either a conjunctive condition in a non-leaf node is wrong or the tree topology is not correct and two leaf nodes should be swapped.

6.3.1.2 Test suite (TS)

The test suite *TS* should thoroughly exercise *IntC*, i.e., by covering all paths in the tree. TS includes oracle's assertions to determine whether the test cases in TS are revealing feature interaction failures or not. Specifically, each test case $tc \in TS$ is composed of a set of input parameters, discussed in section 5.2.2, and an assertion. An assertion $fail(tc)$ is a Boolean function that is true when a test case fails. To compute $fail(tc)$, we define a distance function ϕ_l^{tc} that determines whether feature interactions failures are revealed. In the following, we describe how we define ϕ_l^{tc}.

Recall from Chapter 5 that a feature interaction failure is revealed when: (1) Some safety requirement r is violated because (2) *IntC* overrides the output of the feature responsible for *r*. As discussed in Section 5.2.3, we defined two distance functions, failure distance (*FD*) and unsafe overriding distance (*UOD*) to respectively capture the conditions (1) and (2) above. FD_l evaluates how close SafeDrive is from violating its safety requirement l, and UOD_f computes how close SafeDrive is from causing *IntC* to unsafely override f. To define ϕ^{tc}, we combine the two distance functions *FD* and *UOD*. For every safety requirement l of SafeDrive where f is the feature responsible for l, and for every simulation time step i, we define a distance $\phi_l^{tc}(i)$ as follows:

$$\phi_l^{tc}(i) = \begin{cases} \overline{UOD}_f(i) + \overline{FD}_{max} & \text{If } f \text{ is not unsafely overridden} \\ & (\overline{UOD}_f(i) > 0) \\ \overline{FD}_l(i) & \text{Otherwise } (\overline{UOD}_f(i) = 0) \end{cases}$$

where $\overline{FD}$ and $\overline{UOD}$ are the normalized form of *FD* and *UOD*, respectively. To normalize FD_k, we rely on the well-known rational function $f_1(x) = x/(x+1)$ [Arcuri, 2013]. $\overline{FD}_{max} = 1$ and $\overline{UOD}_{max} = 1$, indicating the maximum value of the normalized functions. Each distance function $\phi_l^{tc}(i)$ is defined for each simulation step i. We define a distance ϕ_l^{tc} for the entire simulation time interval as follows:

$$\phi_l^{tc} = Min\{\phi_l^{tc}(i)\}_{0 \leq i \leq \frac{T}{\delta}} \tag{6.1}$$

where, T is the duration of the simulation and δ is the simulation step size.

If there exists l such that ϕ_l^{tc} is equal to zero, $fail(tc)$ is true, otherwise $fail(tc)$ is false. After running each tc in TS and computing $fail(tc)$, TS is partitioned into TS_p and TS_f (i.e., $TS = TS_p \cup TS_f$), where TS_p is a set of passing test cases and TS_f is a set of failing test cases.

6.3.2 Fault Localization

As discussed in Section 6.2, existing statistical debugging techniques assume that faulty programs contain one single fault that can be confined in one individual statement. The premise of statistical debugging techniques is that they rank individual statements, and then pick the top statements in the ranked list and try to fix them. However, the two high-level errors, discussed in Section 6.1, cannot be pinpointed at the level of statements. This is because our errors are at the level of rules whose implementation spans several lines of code. Hence, the failures are related to different locations in the code and not to one single statement.

In our context, each test case covers a large portion of *IntC* (one path in each time step). As a consequence, both failing and non-failing tests cover the same (large) set of paths of the code albeit with different data values. Indeed, each test case executes one path at each time step, which leads to cover multiple paths through different subsequent steps (within the simulation window). Recall from Section 6.2 that the system is executed within a continuous loop over time.

Existing statistical debugging techniques assign a suspicious score (suspiciousness) to each statement in the program to repair. Since each test (either failing or passing) covers multiple paths, Tarantula will assign the same high suspiciousness to most of the statements in *IntC* in a way that is proportional to the total number of failing tests. This leads to a very large search space as patches can be applied almost everywhere in *IntC*.

Based on this information, we modify the Tarantula formula (Equation 2.1) by (1) focusing on the path/trace covered at the time of failure, and (2) considering the "severity" of the failure. In the following, we describe how we rank test cases and how test cases can guide us to the faulty path in the tree.

Localizing the faulty path. The goal is to determine the path executed by each $tc_j \in TS_f$ that is more likely to be associated with the fault. Recall that each test case in *TS* is executed over the entire simulation time T. At each simulation time step, one tree path is executed, and only one decision rule is selected. We denote by $tf_{tc_j} \in [0 \dots T]$ the time when the failure detected by tc_j has happened. Recall from section 6.1 that a fault in *IntC* may be due to (1) the conjunctive conditions on the left side of the rules are not correctly defined (i.e., the conditions on the input variables at the non-leaf nodes on a tree path were not adequately defined) or (2) the order of the rules is not correct (i.e., the leaf nodes on a path are not properly ordered). We assume that the rule selected at tf_{tc_j}, is wrong (i.e, the conditions on the left side of the rule is wrong, or this rule should not have been selected at tf_{tc_j}). Hence, given a test case tc_j, the faulty path is the one selected at tf_{tc_j}. We denote it by π_{tc_j}.

The severity of the failure. In state-of-the-art statistical debugging techniques, failing tests have the same weight (i.e., one) in the formula for computing the statement suspiciousness. However, in case of multiple faults, this strategy does not provide priorities to the (likely) faulty statements to repair first. Focusing on faulty statements that are related to the most severe failures can lead to patches with the largest potential benefits (gain) to the overall fitness function. To this aim, we rely

on the failure distance function discussed in Section 5.2.3. For each test case, SafeDrive has several safety requirements to fulfill. Let *SR* be the set of safety requirement of SafeDrive. For every safety requirement $l \in SR$, we have a failure distance function FD_l that shows how close SafeDrive is from violating l. A failing test case exposes a failure whose severity is inversely proportional to the failure distance function FD_l. For example, let us consider the safety requirement: "*the minimum distance between the ego and the leading cars* must be larger than a certain threshold D_{th}". A failure happens when the distance between the two cars becomes lower than D_{th} within the simulation time, and its severity is $1/(d+1)$.

In general, a failing test case tc_j exposes a violation to the safety requirement l whose failure severity within the entire simulation time interval is defined as:

$$\omega_l(tc_j) = \frac{1}{Min\{\overline{FD}_l(i)\}_{0 \leq i \leq \frac{T}{\delta}} + 1} \tag{6.2}$$

where $\omega_l(tc_j)$ takes a value between 0 and 1. The lower the value of $\overline{FD}_l$, the higher the severity of the failure for the requirement l. We note that when a test case is passing, $\forall l \in SR, \omega_l(tc_j) = 0$.

For each failing test case $tc_j \in TS_f$, we aggregate the severity of the failure related to each safety requirement $l \in SR$ as follows:

$$\omega(tc_j) = \sum_{l \in SR} \omega_l(tc_j) \tag{6.3}$$

We refer to $\omega(tc_j)$ as *the severity of failures exposed by a test case* tc_j. If the test case passes then the severity $\omega(tc_j)$ is zero otherwise it takes value in $[0,1]$.

Our fault localization formula. We define the suspiciousness of each statement s by considering both failure severity and faulty paths as follows:

$$Score(s) = \sum_{tc_j \in TS_f} I_{tc_j}(s) \tag{6.4}$$

where,

$$I_{tc_j}(s) = \begin{cases} 0 & \text{if } s \notin \text{faulty path } \pi_{tc_j} \\ \omega(tc_j) & \text{otherwise} \end{cases} \tag{6.5}$$

Selecting a statement. Instead of systematically selecting the single worst statement (top-ranked), we select in a randomized manner a statement among the most suspicious ones. First, fault localization is probabilistic. The tree path matching the worst statement might not be the one that requires change. Second, in our work, we have multiple failures in multiple locations of the code. To make a patched version, we therefore must mutate in multiple locations, which are associated with different failures. Third, if we keep selecting the same worst statement and keep focusing on repairing faults exposed by the corresponding test cases, we are less likely to improve than if we consider other statements as well.

To select a statement among the most suspicious ones, we use Roulette Wheel Selection (RWS) [Holland, 1992] that assigns a probability for each statement. The probability is based on the score of each statement. The higher the score of a statement, the higher its probability of being selected. We note that by using RWS, though we dedicate more execution time to fixing the most suspicious failures, we consider all failures. RWS defines the probability as follows:

$$\textbf{prob(s)} = \frac{Score(s)}{\sum_{s_i} Score(s_i)} \tag{6.6}$$

Let $F(s)$ be the set of failing test cases that cover the selected statement s. For each test case $tc \in F(s)$, it exists a path π_{tc} that has covered s at tf_{tc}. We select one of these paths randomly and we denote it π, where π covers s at the time of failure tf. The outputs of the fault localization step are s and π.

6.3.3 Generating a Patch

Having located a faulty path and statement, the goal is to generate a patch for that faulty path or statement by mutating it. Algorithm 6 describes how to generate a patch. Algorithm 6 receives a faulty IntC and a set *TS* of test cases. Algorithm 6 starts by localizing the fault (line 2), and then applies mutation operators to mutate the faulty IntC (generate a patch). Fault localization is described in Section 6.3.2. Fault localization produces the faulty path π and the faulty statement s. In this section, we first describe how patch generation is done (line 3-9 in Algorithm 6) and then we describe in details the mutation operators that are used for patch generation (line 6).

Algorithm 6 iteratively applies a mutation operator to *IntC*. At each iteration, a mutation operator is selected randomly. The number of iterations is determined by a probability. At each iteration, the loop condition compares the probability with a threshold, which is reduced subsequently. Specifically, at each iteration, the threshold is equal to σ^i, where i is the iteration. Using the probability σ, mutation can be applied many times but it is at least applied once, since the counter is initially equal to 0 ($\sigma^0 = 1$). The reason of applying a sequence of mutation operators is to increase the probability of fixing faults in *IntC*. We describe below the two mutation operators **modify** and **shift** that are used in line 6 of Algorithm 6. Note that these mutation operators receive π and s that are produced by the fault localization step.

6.3.3.1 Mutation Operators

We define two search operators **modify** and **shift** based on the error types in the decision rules defined in Section 6.1. Specifically, **modify** aims to modify conditions at the non-leaf nodes and **shift** aims to modify the tree structure of *IntC* (i.e., the order of the decision rules) to correct the fault. The two operators **modify** and **shift** are defined as follows:

- Operator **modify**: The statement s is represented by a node in the tree. Figure 6.4 illustrates an example of node to be modified by the **modify** operator. In the fault localization step, the statement s

Algorithm 6: GENERATE-PATCH

Input: P: A faulty *IntC* version
Input: TS: Test suite
Input: σ: A probability value
Input: Σ: Severity of failure exposed by all test cases
Result: O: A mutant of P

```
1 begin
2     {π, s} ⟵ FL(P, TS, Σ)                    // FL is explained in section 6.3.2
3     counter ⟵ 0
4     threshold ⟵ (σ)^counter
5     while rand(1) ≤ threshold do
6         O ⟵ APPLY-MUTATION({modify, shift}, π, s)
7         counter ⟵ counter + 1
8         threshold ⟵ (σ)^counter
9     return O
```

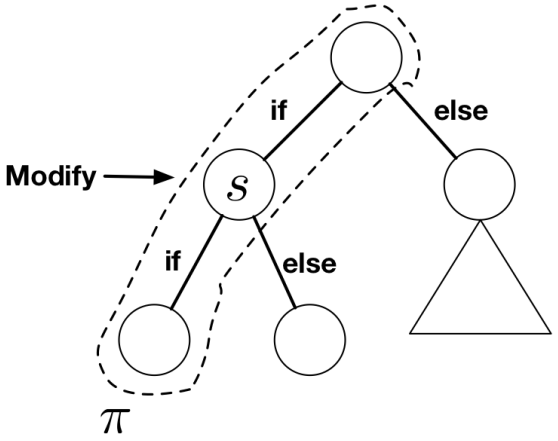

Figure 6.4. modify operator.

is selected in such a way it is a non-leaf node. Therefore, the node (i.e., statement s) has conditions. Operator **modify** modifies the statement s, by performing the following types of changes: changing a threshold value, altering the direction of a relational operator (e.g., $\leq$ to $\geq$), or swapping arithmetic operations (e.g., $+$ to $-$).

- Operator **shift**: This operator aims to change the order of the decision rules. Changing the order of rules may correct the errors since, as discussed in Section 6.1, the rule ordering selected by engineers may not be correct. Recall from Section 6.1 that the decision rules are represented by a tree. The idea of **shift** operator is to change the order of rules by changing the structure of the tree such that the path π is not executed at the time of failure.

Recall from Section 6.3.2 that s is in π. To change π and swap s, we randomly select one node among all the nodes in the tree that are either an ancestor of s, or have s as an ancestor, and we refer to it as b^s. Figure 6.5 shows an example of possible nodes to be selected for the **shift** operator. As shown in Figure 6.5, the ancestor nodes of s or the nodes that have s as ancestor, in gray, are the non-leaf

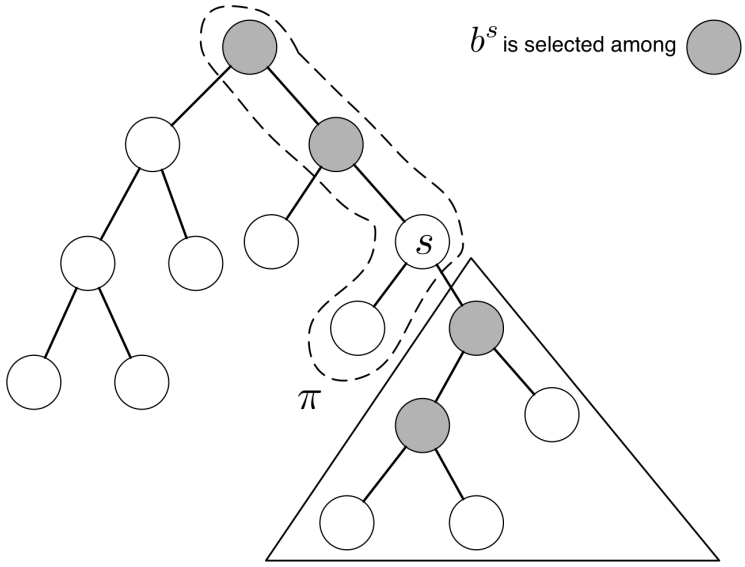

Figure 6.5. Example of selecting b^s (path condition) to be used for the **shift** operator.

nodes in π and the non-leaf nodes in the sub-tree below s.

When b^s is an ancestor of s, the shift action consists of removing (shifting) the left subtree of s, and placing it on top of b^s such that the first node in the right subtree of s becomes b^s (i.e., s is the parent of b^s). Figure 6.6(a) shows an example of this case. Dually, when s is an ancestor of b^s, the shift action consists of removing (shifting) the left subtree of b^s, and placing it on top of s such that b^s becomes the parent of s. Figure 6.6(b) shows an example of this case.

The reason of selecting b^s as an ancestor of s or selecting b^s in such a way that s is an ancestor of b^s, is that we want b^s to be control dependent on s, which ensures to avoid meaningless rules (because the conditions on the non-leaf nodes of π, except s, also hold for the nodes that have s as ancestor).

As discussed in section 6.1, each feature has a set of rules that are partially ordered. Hence, when we apply **shift**, the partial order among each feature should be respected.

6.3.4 Evaluating a Patch

This step takes as input the patch (i.e., IntC version) generated in the previous step and evaluates it. The pseudo-code of this step is shown in Algorithm 7. To evaluate the patch, each test case $tc \in TS$ needs to be simulated using the new patch. We simulate all test cases and not only the ones impacted by the mutation since (1) we need to guarantee that the applied mutations (modifications) do not inject new errors and (2) since each test case in TS is executed over the entire simulation time T, the IntC code is executed within a loop (i.e., at each simulation time step, one tree path and its corresponding leaf node (feature) are selected. Thus, every change in iteration i is going to impact all the paths covered in the next iterations.

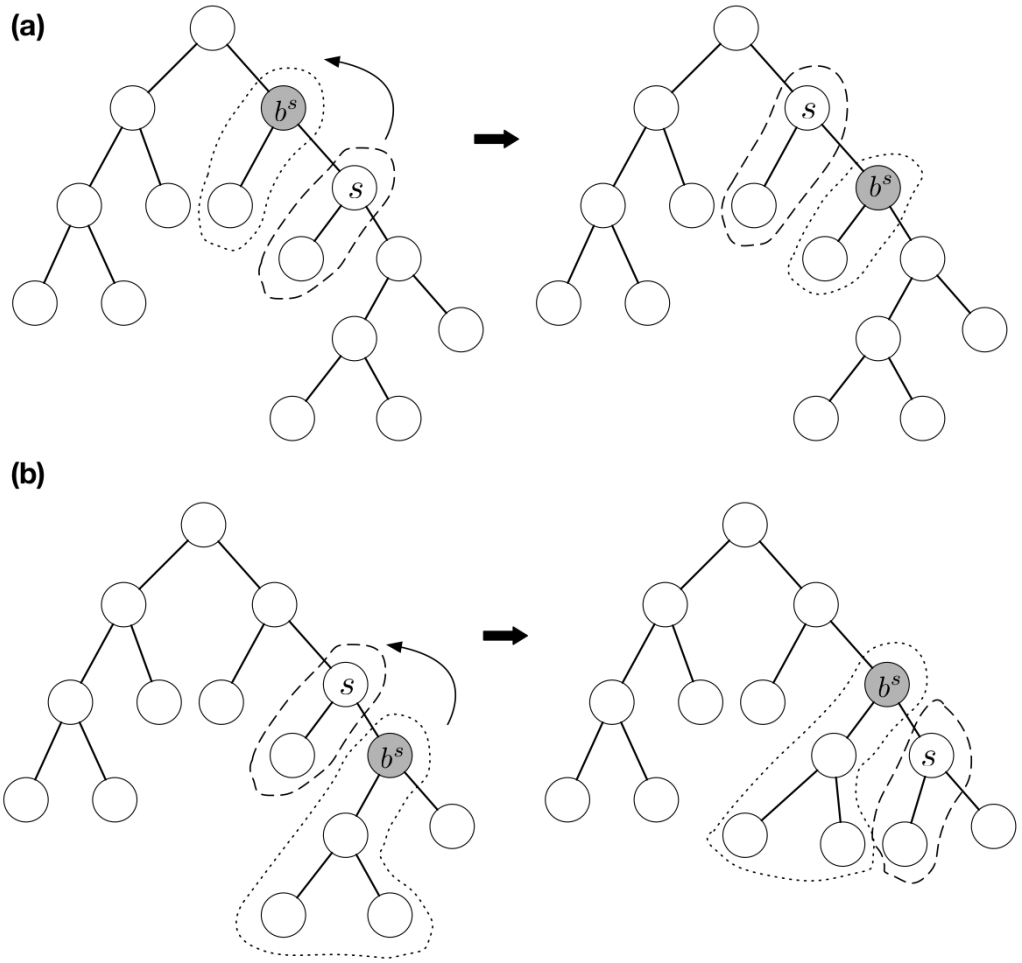

Figure 6.6. Examples of applying the **shift** operator.

After simulating all test cases in TS, we compute fitness functions, and we use an archive to keep the best patches found at each iteration of the search. In the following, we describe how we define the fitness functions and discuss how the archive is updated at each iteration.

6.3.4.1 Fitness Function

Our goal is to define fitness functions that can guide the search into repairing *IntC*. Computing only the number of violated safety requirements or the number of test cases passed by *IntC* (i.e., number of passing test cases), similar to classical programs repair, will not provide enough gradient to the search (i.e., will not provide guidance to correct faults). Thus, we use a quantitative measure formalizing the *severity of the failures* exposed by failing test cases. Specifically, for every safety requirement l and each test case $tc \in TS$, this measure computes the failure severity exposed by the violation of the safety requirement l ($\omega_l(tc)$) discussed in Section 6.3.2 (Equation 6.2).

Our goal is to repair *IntC* by minimizing the severity of failures exposed by test cases until eventually they all pass. At the same time, we want to be sure that the passing test cases remain correct.

Algorithm 7: RUN-EVALUATE

Input: O: A mutant of *IntC*
Input: TS: Test suite
Result: Ω: Fitness functions
Result: Σ: *DistanceFromViolations* of test cases

```
1  begin
2      for tc ∈ TS do
3          SIMULATE-SYSTEM(O, tc)
4          for l ∈ SR do
5              ω_l(tc) ⟵ CALCULATE-SEVERITY-OF-FAILURE-TC(tc)        // Equation 6.2
6
7          ω(tc) ⟵ CALCULATE-SEVERITY-OF-FAILURE(ω_l(tc))            // Equation 6.3
8      for l ∈ SR do
9          Ω_l(TS) ⟵ CALCULATE-FITNESS({ω_l(tc_1), ..., ω_l(tc_n)})   // Equation 6.7, n = |TS|
10     Σ ⟵ {ω(tc_1), ..., ω(tc_n)}
11     Ω ⟵ {Ω_1, ..., Ω_m}                                            // m = |SR|
12     return (Ω, Σ)
```

Hence, for every safety requirement l, we define a fitness function Ω_l by taking the maximum value of the *severity of failure* ω_l exposed by all test cases in TS. For every safety requirement l of SafeDrive, we define Ω_l as follows:

$$\Omega_l = \max_{tc \in TS} \omega_l(tc) \tag{6.7}$$

To repair *IntC*, our goal is to minimize Ω_l for all $l \in SR$ (SR is the set of safety requirements of SafeDrive).

6.3.4.2 Archive

At the beginning, the faulty IntC is stored in an archive. Every time a new patch is created and evaluated, we compare it with the patches stored in the archive. Specifically, to compare between two patches and determine which one is optimal with respect to the identified fitness functions, we use the notion of dominance, defined in Pareto optimal approaches [Luke, 2013] "A solution x dominates another solution y, if x is not worse than y in all fitness values, and x is strictly better than y in at least one fitness value".

This step takes as input the values of the fitness functions of the new patch. Then, the new patch is compared with every element in the archive. If the new patch dominates an element in the archive, we replace that element with the new patch. Otherwise, if there is no element in the archive that dominates the new patch, the new patch is added to the archive. The pseudo-code of this step is shown in Algorithm 8.

The archive at the end includes the found non-dominated solutions (i.e., the best-found patches with respect to the fitness functions). Since the number of non-dominated solutions may be extremely large, we define a maximum size for the archive, denoted by s_A. The value of s_A is twice the number of safety requirements. If the size of the archive exceeds s_A, we evaluate each patch in the archive by computing the *severity of failure* exposed by a test suite TS, denoted by D. Then, we select the s_A

patches that have the lowest D values. To compute D, we aggregate the *severity of failure* exposed by test cases (defined in Equation 6.3):

$$D = \sum_{tc \in TS} \omega(tc) \tag{6.8}$$

Algorithm 8: UPDATE-ARCHIVE

```
   Input: A: Archive
   Input: O: A mutant of IntC
   Input: Ω: Fitness funtions
   Result: A': An updated archive
1  begin
2      A' ⟵ A
3      isDominated ⟵ false
4      for a ∈ A' do
5          if O dominates a                          // the dominance uses Ω
6          then
7              remove a
8              A' ⟵ A' ∪ {O}
9          else
10             if a dominates O then
11                 isDominated ⟵ true
12     if isDominated = false then
13         A' ⟵ A' ∪ {O}
14     return A
```

6.3.5 Search Algorithm

As discussed in Section 6.2, the GP used by traditional program repair techniques is not adequate in our context. Therefore, we opt for a single solution algorithm instead of a population-based algorithm. Our approach is applied iteratively to repair *IntC*. We use a many-objective search optimization algorithm to guide the repair of IntC. Algorithm 9 shows our proposed algorithm, RUFI, that repairs *IntC*. RUFI receives as input a faulty *IntC*, and a test suite TS. The output is a repaired *IntC* version. Given the faulty IntC and the set of failing test cases, RUFI starts by localizing the fault and creating a patch for that fault (line 3). This step is discussed in section 6.3.2. Then, RUFI simulates each test case $tc \in TS$ using the new patch and computes the fitness functions (line 4), as explained in section 6.3.4. At the beginning of the search, the faulty IntC is stored into an archive. After evaluating the new patch, RUFI updates the archive (line 5) by comparing the new patch with the faulty IntC. The update archive step is discussed in section 6.3.4. Our algorithm, then, searches for the best solution (*IntC* version) through subsequent iterations (loop in lines 6-10). In each iteration, one of the elements (i.e., patches) from the archive is selected randomly (line 7). Then, RUFI localizes the fault and creates a new patch (line 8). Next, the new patch is evaluated by computing the fitness functions (line 9). Finally, RUFI updates the archive by comparing the new patch with each element in the archive (line 10). At each iteration, the archive contains the best-found patches with respect to the fitness

functions. The search stops when the archive includes one patch such that the number of failing test cases is equal to zero.

Algorithm 9: RUFI

```
   Input: P_f: Faulty IntC
   Input: TS: Test suite
   Result: A: A patch of P_f satisfying all tc ∈ TS
1  begin
2      A ⟵ P_f
3      O ⟵ GENERATE-PATCH(P_f, TS)
4      {Ω, Σ} ⟵ RUN-EVALUATE(O, TS)
5      A ⟵ UPDATE-ARCHIVE(A, O, Ω)
6      while not(One element in A satisfies all tc ∈ TS) do
7          P ⟵ SELECT-A-PARENT(A)                    // P is selected randomly
8          O ⟵ GENERATE-PATCH(P, TS, Σ)
9          {Ω, Σ} ⟵ RUN-EVALUATE(O, TS)
10         A ⟵ UPDATE-ARCHIVE(A, O, Ω)
11     return A
```

6.4 Evaluation

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

6.4.1 Research questions

RQ1: *How effective and efficient is our approach in localizing and repairing faults?* This question aims at evaluating the extent to which RUFI is able to effectively and efficiently repair the decision rules of self-driving systems. An effective approach should be able to fix all the bugs in the decision rules. An approach is deemed efficient if it can find a correct version of the faulty rules in a practical amount of time.

RQ2: *How does our approach compare to a traditional program repair approach?* This question aims at evaluating the benefits obtained from our many-objective single-state search algorithm (RUFI), which uses an archive to keep track of partial patches, compared to the baseline relying on GP with single-objective optimization. To answer this question, we need to compare the performance in terms of execution time.

6.4.2 Experiment Design

We evaluate our approach by applying it to the *SafeDrive* case study system introduced in Chapter 5 (Section 5.1). We use the common GP algorithm used for program repair [Weimer et al., 2009, Le Goues et al., 2012], described in Section 2.3, as a baseline of comparison for answering **RQ2**. We note that based on the state of the art, this is the most commonly used algorithm for program repair.

However, to address the specificities of the faults described in Section 6.1, we must slightly adapt the GP algorithm to our context by using our proposed fault localization and mutation techniques.

Test cases. As suggested in the literature, we use a small number of test cases [Weimer et al., 2009, Le Goues et al., 2012, Qi et al., 2014], which allows for more search iterations within a fixed amount of time (i.e., the more test cases are used, the more computational resources are required by fitness evaluations). However, if too few test cases are used, the repair may overlook some of the system functionalities. The test suite should include both passing and failing test cases. To select failing test cases, we use our approach proposed in Chapter 5 that automatically detects conflicts between feature outputs by generating failing test cases. The detected failures were related to the violation of four safety requirements, discussed in Section 5.2.3 (Table 5.1). For each safety requirement, we selected one failing test case. As for the passing test cases, we selected four test cases that satisfy these four safety requirements. Therefore, we set the test suite size to eight.

Parameters. The mutation probability σ is set to 0.5 as suggested in the literature [Fraser and Arcuri, 2013b]. In RUFI, we do not need to set the archive size arbitrarily since, as described in Section 6.3.4.2, it is dynamically updated at each iteration. For the Baseline, we set the (initial) population size to 40, which represents the recommended value used in the literature [Weimer et al., 2009, Le Goues et al., 2012]. For computing fitness (discussed in Section 2.3), we set $W_{Neg} = 10$ and $W_{Pos} = 1$ as suggested in the literature [Weimer et al., 2009, Le Goues et al., 2012].

The search stops when a patch passes all the test cases or when the timeout of 16 hours is reached. We set a timeout of 16 hours because as we will discuss in Section 6.4.3, it is sufficient to repair our decision rules. We ran all the experiments on a laptop with a 2.5 GHz CPU and 16GB of memory.

6.4.3 Results

RQ1. To answer RQ1, we ran RUFI 20 times for 16 hours to account for the randomness of the search algorithm. The search stops when a repaired program is found. Our experiments show that our approach can successfully repair IntC within the search time budget. Hence, our approach is effective for repairing faults in the decision rules of self driving-cars.

To evaluate the efficiency of our approach, we report the time needed by our approach to repair IntC. The boxplot in Figure 6.7 shows the variation of the time needed for RUFI to repair IntC across 20 independent runs. The average amount of time spent by our approach to repair IntC is five hours. This time is significantly less than the time required (Chapters 4 and 5) for testing self-driving features using a simulation environment, i.e., 12 hours and 20 hours, respectively, to test self-driving systems. More importantly, such repairs, in practice, take place over night, thus leading to a fixed set of rules the next day. Such results indicate that RUFI is efficient for repairing decision rules (in our case, for self-driving cars).

RQ2. To answer RQ2, we would need to compare the time needed for each algorithm to repair IntC. However, we were not able to run the baseline because computing the fitness of a population

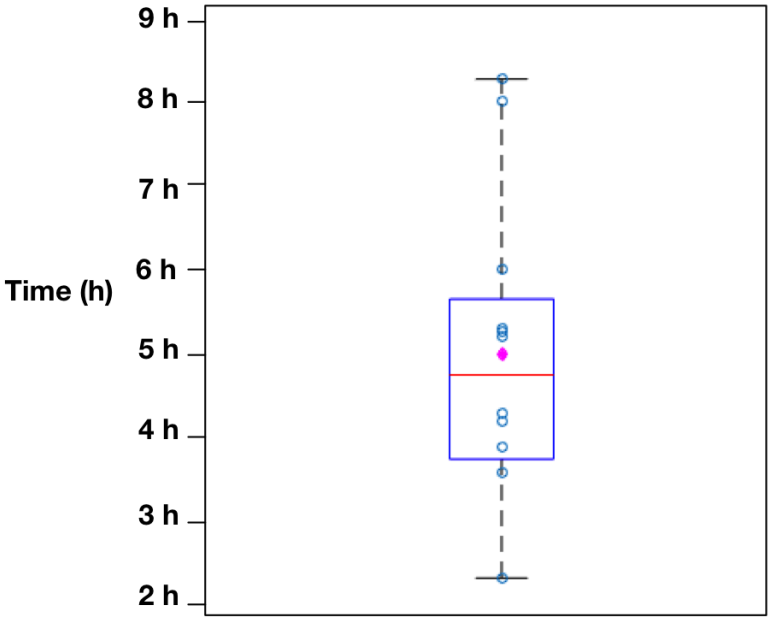

Figure 6.7. Time required for RUFI to repair *IntC*

of 40 individuals takes on average 10 hours. Each test case takes on average 2 minutes to simulate SafeDrive (2 minutes (test case) $\times 8$ (test suite size) $\times 40$ (population size) $= 10$ hours). Therefore, not only it is impossible for us to experiment with the baseline in our context but also such computation times make it impractical. We conclude RUFI is clearly more adequate in our context, and it is more efficient than repair algorithms based on population solutions in terms of the time it takes to repair a faulty program.

Our approach effectively repairs the faulty IntC. Hence, the repaired version of IntC can be used by engineers to avoid conflicts between feature outputs. Further, our approach can help engineers to extend IntC for including new features. Specifically, when a new feature f_a is added to the system, engineers develop a new list of rules that specifies under what conditions the new feature output should be prioritized. Then, they integrate the developed rules based on their domain knowledge into IntC to specify how f_a interacts with existing features. However, the order of the rules or the conjunctive conditions of the newly developed rules may not be correct. Our approach can help order the rules and correct the conjunctive conditions of each new rule if they are wrong. Our approach requires engineers to add additional test cases to the test suite. The new test cases should include at least one passing test case that satisfies the safety requirement of f_a, and at least one failing test case that violates the safety requirement of f_a. We note that the proposed is general and is applicable to any system that resolves feature interactions through a dedicated integration component.

6.5 Conclusions

The resolution rules that are developed to resolve conflicts between features require extensive domain expertise and a thorough analysis of system requirements. When the rules or their implementation are

erroneous, this may result in undesired feature interactions. Therefore, faulty decision rules should be repaired and preferably automatically so when a system test suite is available.

In this chapter, we presented an automated technique for identifying and repairing errors in the feature interaction resolution rules for self-driving systems. We developed a search-based repairing algorithm that localizes faults and mutates the faulty decision rules to generate patches. To localize faults, we defined a fault localization formula that aims to compute the suspiciousness of each statement by considering both the severity of failures and faulty paths. To mutate decision rules, we proposed mutation operators that are designed to address the specificities of the errors in the decision rules for self-driving systems. Our approach is evaluated on an industrial ADAS. The results indicate that our approach efficiently and effectively repairs decision rules of self-driving cars. Our approach, further, can help engineers with extending their systems for including new features.

Chapter 7

Related Work

This chapter provides an overview of existing work related to the approaches researched and developed in this dissertation: search-based testing, surrogate modeling, testing autonomous cars, analysis of feature interactions, and program repair techniques.

7.1 Search-based testing

Search-based testing has largely focused on unit testing and has rarely been used for system testing. Exceptions include GUI testing [Gross et al., 2012, Mariani et al., 2014] and the generation of system test cases to exercise non-functional behaviors such as quality-of-service constraints [Shams et al., 2006], computational resources consumption and deadline misses [Briand et al., 2006a]. Embedded software systems and their environments are prevalently captured and simulated using physics-based models such as those captured by MATLAB/Simulink. Some of the test automation techniques for MATLAB/Simulink use meta-heuristic search to guide testing towards the maximisation of coverage criteria [Windisch, 2010, Zhan and Clark, 2006], for example path coverage [Zhan and Clark, 2006], or towards the generation of input signals that satisfy certain signal shape patterns [Baresel et al., 2003] or temporal constraints [Wilmes and Windisch, 2010]. These testing strategies have mostly focused on unit/function-level testing, aiming to maximize coverage or diversity of test inputs. These strategies, however, are inadequate for testing complex physics-based dynamic models such as those used for self-driving systems. Our testing approach, proposed in Chapter 3 and Chapter 4, in contrast, is driven by system-level requirements as well as critical and stressful situations of the system and its environment.

7.2 Surrogate modeling

Surrogate modeling has been previously used to approximate expensive fitness computations and simulations in the context of evolutionary and meta-heuristic search algorithms. Surrogate modeling has been applied to scale up search-based solutions to optimization problems in avionics [Ong et al.,

2003], chemical systems [Caballero and Grossmann, 2008], and the medical domain [Douguet, 2010]. In particular, combination of surrogate modeling and multi-objective population-based search algorithms has been applied to optimization problems in mobile ad hoc networks [Efstathiou et al., 2014], manufacturing [Syberfeldt et al., 2008], and optimizing energy consumption in buildings [Magnier and Haghighat, 2010]. These techniques, however, solely rely on surrogate model predictions to select best candidate solutions without using the prediction errors and confidence levels. This may lead to a significant deviation between the best solutions selected based on surrogate model predictions and those solutions that would have been selected based on actual fitness computations. In contrast, in our work discussed in Chapter 3, we use the prediction errors to decide whether we should compute actual fitness values for candidate solutions or not. Further, we show that when actual fitness values are not better than their respective optimistic predictions, NSGAII and NSGAII-SM behave the same, but NSGAII-SM is likely to call less simulations per iteration than NSGAII. In [Matinnejad et al., 2014], surrogate modeling has been used in conjunction with single-objective local search such that prediction errors and actual fitness values are used to ensure the search algorithm accuracy. Our work described in Chapter 3 differs from the work of [Matinnejad et al., 2014] as we combine surrogate modeling with multi-objective population search algorithms.

7.3 Testing autonomous cars

Simulation, i.e., design time testing of system models, is arguably the most practical and effective way of testing autonomous cars [Belbachir et al., 2012]. This is because rich simulation environments are able to replicate various real world traffic situations. The main difficulty with simulation-based testing of ADAS is that the space of test input scenarios is complex and multidimensional. To address this limitation, many techniques rely on search-based system testing to automate test generation for ADAS [Bühler and Wegener, 2008]. For example, search-based system testing has been applied to a vehicle-to-vehicle braking assistance [Buehler and Wegener, 2005], and an autonomous parking [Bühler and Wegener, 2004]. Bühler and Wegener [Buehler and Wegener, 2005, Bühler and Wegener, 2004] base their testing on a single-objective search algorithm. Recently, Tian et. al. [Tian et al., 2018] and Zhang et. al. [Zhang et al., 2018] proposed a notion of neuron coverage and used it to guide the generation of tests for neural networks used in autonomous cars. In contrast to our proposed approach in Chapter 4, none of these approaches consider (static) environment variables in the test input space, and they vary only mobile objects' variables in test scenarios. Hence, these approaches are not able to automatically explore different environment conditions (e.g., different road types and weather conditions). Further, the above-cited work focuses on identifying individual critical simulation scenarios only. Our work in Chapter 4 deals with a considerably larger test input space that includes environment variables. Further, we provide in Chapter 4 a novel search-based testing algorithm that, in addition to identifying individual critical scenarios, characterizes critical regions of the ADAS test input space.

Further, none of the above-cited work study the feature interaction problem in autonomous cars. In Chapter 5, 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 [McMinn, 2004a, Tonella, 2004, Fraser and Arcuri, 2013a] and failure-based [Bühler and Wegener, 2008, Afzal et al., 2009, Briand et al., 2006a]) with our proposed unsafe overriding heuristic. Further, we tailor existing many-objective search algorithms [Panichella et al., 2015, Panichella et al., 2018] to detect feature interaction failures in our context.

7.4 Feature interactions

In this section, we compare our work with feature interactions approaches in software product lines, feature interaction detection techniques via model checking, and feature interaction resolution strategies.

7.4.1 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 [Oster et al., 2011, Patel et al., 2013, Lochau et al., 2012]. These approaches largely follow a model-based testing paradigm [Ammann and Offutt, 2008]. For example, they use combinatorial testing to drive test cases and oracles from feature models to verify individual products [Oster et al., 2011, Patel et al., 2013]. Our work in Chapter 5, in contrast, is model testing [Briand et al., 2016a]. 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 in Chapter 5 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 [Apel et al., 2013a, Kolesnikov et al., 2017, Ferreira et al., 2015]. For example, interactions between observable feature behaviors (i.e., external feature interactions [Apel et al., 2013a]) have been identified by static analysis of software code [Kolesnikov et al., 2017, Ferreira et al., 2015]. In contrast, our approach in Chapter 5 detects feature interactions prior to any software coding. It dynamically detects undesired feature interactions by testing function models capturing the SUT and its environment.

7.4.2 Feature interaction detection via model checking

Several approaches are proposed to detect feature interactions by model checking requirements or design artifacts against formal specifications [Apel et al., 2013b, Arora et al., 2012, Juarez-Dominguez et al., 2008, Sobotka and Novak, 2013, Plath and Ryan, 2001]. For example, Apel et. al. [Apel et al., 2013b] verify features described in a formal feature-oriented language against temporal logic properties [Clarke et al., 1999]. Arora et. al. verify features defined as state machines against live sequence charts specifications. Dominguez et. al. [Juarez-Dominguez et al., 2008] verify features captured as StateFlows, and Sobotka and J. Novak [Sobotka and Novak, 2013] specify features in

timed automata [Alur, 1999]. Similar to our work in Chapter 5, 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 [Matinnejad et al., 2016, Abbas et al., 2013, Matinnejad et al., 2018]. Therefore, as suggested in the recent research on testing CPS models [Matinnejad et al., 2016, Abbas et al., 2013, Matinnejad et al., 2018, Zuliani et al., 2013], instead of model checking, we rely on simulation-based testing guided by meta-heuristics to analyze our function models.

7.4.3 Feature interaction resolution

Several approaches are proposed to devise resolution strategies to eliminate undesired feature interactions, for example, by proposing specific feature-oriented architectures [Jackson and Zave, 1998, van der Linden, 1994], by statically prioritizing features [Zimmer and Atlee, 2012, Hay and Atlee, 2000] or using runtime resolution mechanisms [Bocovich and Atlee, 2014, Zibaeenejad et al., 2017]. 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 proposed in Chapter 5 is still necessary to test the system behavior and to determine if the proposed resolution strategy can eliminate undesired behaviors under different environment conditions.

7.5 Program repair

Automated program repair has been the subject of considerable recent attention in the software engineering research community. Several techniques have been proposed to automatically repair faulty programs without human intervention [Monperrus, 2018]. These automated repair methods include search-based methodologies [Arcuri and Yao, 2008, Arcuri and Fraser, 2011b, Le Goues et al., 2012, Kim et al., 2013, Qi et al., 2014] and semantics-based methodologies [Nguyen et al., 2013]. Search-based methods explore the space of possible repairs and generate a repair candidate and validate this repair candidate against the provided test-suite. Arcuri et. al. [Arcuri and Yao, 2008, Arcuri and Fraser, 2011b] introduced the idea of using GP to repair software bugs automatically. They proposed a co-evolutionary model of bug fixing that relies on formal specifications for fitness evaluation. Their evaluation was limited to small programs such as bubble sorting and triangle classification. GenProg [Le Goues et al., 2012] and Par [Kim et al., 2013] also use GP to evolve patches to a buggy

program. GenProg generates a population of candidate patches by modifying source code using mutation and crossover. GenProg reuses existing code to synthesize the patches. To improve the repair quality obtained by GenProg, Par [Kim et al., 2013] generates candidate patches learned from human-written patches. Among the techniques that rely on GP [Le Goues et al., 2012, Kim et al., 2013, Arcuri and Yao, 2008, Weimer et al., 2009], GenProg and its extension [Le Goues et al., 2012] have shown the most promising results since it scales to large-scale real-world software. Other techniques using randomized search for patch generation have also been proposed. For example, RSRepair [Qi et al., 2014] replaces the GenProg genetic search algorithm with random search. Although this work indicates that random search performs better than GenProg in terms of the number of patch trials required to search a valid patch, a recent study [Kong et al., 2015] showed that GenProg performs better that RSRepair when applied to subjects different from those included in the original dataset of GenProg. Further, as discussed in [Arcuri and Fraser, 2011b], random search is unlikely to yield a correct implementation of any non-trivial software [Arcuri and Fraser, 2011b]. Hence, we do not use it as a baseline of comparison for our approach and instead we use a slightly adapted version of GenProg.

In contrast to search-based heuristic approaches, semantic approaches synthesize a repair using semantic information (via symbolic execution and constraint solving). For example, SemFix [Nguyen et al., 2013] uses semantic analysis to repair a program by relying on test cases as implicit program specification to guide the patch synthesis process. Although SemFix is shown to be more efficient than GenProg in terms of repairability and running time, SemFix can only be applied to small programs.

Our work in Chapter 6 differs from this line of research in the following ways: First, the above-mentioned approaches repair programs with a single fault and do not support multi-location bug fixing [Qi et al., 2015]. In contrast, our approach repairs a program with multiple faults. Second, existing techniques using GP assess the quality of the generated patches based on a single objective function that measures the number of passing test cases in a test suite (i.e., the patches with the most passing test cases are selected for continued evolution in the next generation). Instead, in our work, we define many objectives, where each objective is related to one safety requirement. We compute the objectives based on the distance from violating the safety requirements related to each test case, and therefore we provide more guidance to the search compared to existing techniques (e.g., GenProc). Third, existing approaches localize faults using statistical debugging techniques that rely on the number of statements executed by passing and failing test cases. This may lead to many statements having the same suspiciousness scores. In contrast, in our work, to determine the suspiciousness of each statement, we rely on the distance from violating safety requirements. Fourth, the existing approaches for repairing software code assume that a patch can often be reconstructed from fragments of code that already exist in the faulty program. However, in Chapter 6, we address a different repair problem since our goal is to repair decision rules of self-driving systems instead of software code. In our work, our mutation operators are designed in a domain specific way (i.e., to address the specificities of our repair problem).

Chapter 8

Conclusions and Future Work

In this chapter we summarize the contributions of this dissertation and discuss some perspectives on potential future work in this area.

8.1 Summary

In this dissertation, we focused on the problem of design time testing of ADAS in a simulated environment. The main challenges of simulation-based testing are that test execution is computationally expensive, the test input space is complex, large, and multidimensional, and the search space includes many local optima. In addition to testing individual ADAS, in this dissertation we address the testing of self-driving systems that include several ADAS. When self-driving systems include many features (i.e., individual ADAS), they may interact and impact one another's behavior in unknown ways, which is referred to as the feature interaction problem. To ensure the reliability of self-driving systems, feature interaction failures should be detected and resolution strategies should be deployed to resolve conflicts. In self-driving systems, feature interactions are numerous, complex and depend on several factors, and hence, are hard to detect. Developing resolution strategies is a complex task and despite the extensive domain expertise, such strategies can be erroneous and are too complex to be manually repaired.

In this dissertation, we proposed several approaches to alleviate the above challenges. We used a combination of multi-objective search and surrogate models to generate test scenarios indicating critical ADAS behaviors within an acceptable time budget. We relied on a combination of multi-objective search and decision tree classification models to generate test scenarios identifying critical scenarios within complex and multidimensional input spaces, and to characterize critical regions of the ADAS test input space. We presented an approach that generates test scenarios revealing feature interaction failures among ADAS systems. Further, we introduced an automated approach for repairing faults exposed by the identified feature interaction failures. The work presented in this dissertation has been done in collaboration with IEE [IEE, 2019], a leading supplier in automotive sensing systems enhancing safety and comfort in vehicles produced by major car manufacturers worldwide.

Chapter 3 presented our automated approach for testing ADAS that generates test cases indicating critical ADAS behaviors within an acceptable time budget. Our proposed technique relies on a combination of multi-objective search and neural networks. We developed meta-heuristics capturing critical aspects of the system and its environment to guide the search towards exercising behaviors that are likely to reveal faults. Our proposed search algorithm relies on neural network predictions to bypass costly simulations when predictions are sufficient to conclusively prune certain solutions from the search space. We evaluated our approach by applying it to an industrial automotive system. Our experiments showed that our search-based algorithm outperforms random test generation. Further, our approach is able to identify critical system and environment behaviors, and surrogate models help improve the quality of the generated test cases under a limited and realistic time budget.

Chapter 4 introduced our automated testing approach for ADAS that guides the search-based generation of tests faster towards critical test scenarios (i.e., test scenarios leading to failures) and accurately characterizes critical regions (i.e., the regions of a test input space that are likely to contain most critical test scenarios). Our algorithm builds on learnable evolution models and uses classification decision trees to guide the generation of new test scenarios within complex and multidimensional input spaces. We evaluated our approach on an industrial ADAS. The results indicate that our classification-guided search algorithm outperforms a baseline evolutionary search algorithm and generates 78% more distinct, critical test scenarios compared to the baseline algorithm. Our approach, further, characterizes critical regions of the ADAS input space. Based on our interviews with domain experts, such characterizations are accurate and help engineers debug their systems. They further help engineers identify environment conditions that are likely to lead to ADAS failures as well as hardware changes that can increase ADAS safety.

Chapter 5 described a technique for detecting feature interaction failures, among interacting ADAS, 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. We casted the problem of detecting undesired feature interactions into a search-based testing problem. We defined a test guidance that combines existing search-based test objectives with new heuristics specifically aimed at revealing feature interaction failures. We tailored existing many-objective search algorithms [35], [36] to automatically reveal feature interaction failures in a scalable way. We evaluated our approach using two versions of an industrial self-driving system. Our experiments demonstrated significant improvement in feature interaction failure identification compared to baseline search-based testing approaches. 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.

Chapter 6 presented an automated technique for identifying and repairing errors in the feature interaction resolution rules for self-driving systems. We developed a search-based repairing algorithm that localizes faults and mutates the faulty decision rules to generate patches. To localize faults, we defined a fault localization formula that aims to compute the suspiciousness of each statement by considering both the severity of failures and faulty paths. To mutate decision rules, we proposed mutation operators that are designed to address the specificities of the errors in the decision rules

for self-driving systems. Our approach is evaluated on an industrial ADAS including four features. The experiments showed that our approach efficiently and effectively repairs decision rules of self-driving systems. Our approach, further, can help engineers with extending their systems to include new features.

8.2 Future Work

In this dissertation, we focused on the problem of testing ADAS in a simulated environment. In the future, we would like to further assess the generalizability of our solutions (Chapter 3 to Chapter 6). In particular, we only have evaluated our approaches on four industrial ADAS, i.e., PP, AEB, TSR, and ACC. To better assess the applicability and effectiveness of our approaches, we should consider other well-known industrial cases such as: 1) Lane Keeping Assist (LKA) that keeps the vehicle in the center of the lane, and 2) Parking Aid (PA) that detects the presence of an object when the vehicle is moving in reverse, and alerts the driver in case he might hit objects. Furthermore, our solutions are potentially usable in other cyber-physical domains, e.g., aerospace, satellite.

In Chapter 6, we proposed an automated approach for repairing feature interaction failures in self-driving systems. Our repair algorithm produces various correct versions of the decision rules for self-driving systems. We would like to investigate whether the repaired decision rules produced by our approach are useful to practitioners and match their domain knowledge. To do so, we have planned to conduct semi-structured interview sessions with senior engineers at IEE. In the sessions, we plan to present the various versions of the correct decision rules and ask the practitioners whether the different versions match their domain knowledge or not. If there exist a correct decision rule that does not match such domain knowledge, this may point to an error in the system or the simulator.

IEE has shown interest in using our testing approaches, discussed in Chapter 3 and Chapter 4, to minimize the cost of their sensing technologies. In the future, we plan to use our approaches to choose optimal products (configurations) that satisfy safety requirements. We expect this to result in a trade-off between cost and safety of self-driving systems. For example, to select the best camera Field of View (FoV) that fulfills some specific requirements, we can decrease the FoV of the camera, and we use our solutions to check whether we identify test scenarios revealing critical failures.

We plan to develop a suite of tools to operationalize the proposed test generation approaches. These tools aim to help IEE in testing software systems in their development cycle. The suite of tools should provide informative and convenient user interfaces that can provide engineers with a better understanding about faults in their systems under test.

Cooperative vehicle systems and infrastructure, such as Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I), have shown to provide enormous benefits in terms of mobility and safety. Many self-driving systems rely on cooperative driving systems to further improve road safety [Kato and Tsugawa, 2001]. Potential interaction conflicts between automated vehicles in shared traffic spaces, such as highways, parking places or intersecting regions, need to be identified and solved

in a cooperative way. Testing cooperative systems of autonomous vehicles is more complex than testing ADAS because more vehicles are involved, and the number of possible interactions between these vehicles or between vehicles and infrastructures is huge. In the future, we plan to devise an automated strategy to identify possible interactions in cooperative driving of autonomous vehicles and to guide engineers to resolve potential conflicts.

List of Papers

Published papers included in this dissertation:

1. Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. **"Testing advanced driver assistance systems using multi-objective search and neural networks"**. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, Singapore, September 3-7, pp. 63-74, 2016.

2. Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. **"Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms"**. In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE'18)*, Gothenburg, Sweden, May 27 - June 3, pp. 1016-1026, 2018.

3. Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. **"Testing Autonomous Cars for Feature Interaction Failures Using Many-objective Search"**. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*, Montpellier, France, September 3-7, pp. 143-154, 2018.

Bibliography

[Abbas et al., 2013] Abbas, H., Fainekos, G., Sankaranarayanan, S., Ivančić, F., and Gupta, A. (2013). Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):95.

[Afzal et al., 2009] Afzal, W., Torkar, R., and Feldt, R. (2009). A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976.

[Alippi and Roveri, 2010] Alippi, C. and Roveri, M. (2010). Virtual k-fold cross validation: An effective method for accuracy assessment. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'10)*, pages 1–6.

[Alpaydin, 2010a] Alpaydin, E. (2010a). *Introduction to Machine Learning*. MIT Press, Cambridge, Massachusetts, USA, 2nd edition.

[Alpaydin, 2010b] Alpaydin, E. (2010b). *Introduction to Machine Learning*. The MIT Press, 2nd edition.

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

[Ammann and Offutt, 2008] Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition.

[Apel et al., 2013a] Apel, S., Kolesnikov, S., Siegmund, N., Kästner, C., and Garvin, B. (2013a). Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD'13)*, pages 1–8, Indianapolis, USA. ACM.

[Apel et al., 2013b] Apel, S., Von Rhein, A., ThüM, T., and KäStner, C. (2013b). Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399–2409.

[Arcuri, 2013] Arcuri, A. (2013). It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147.

[Arcuri and Briand, 2014] Arcuri, A. and Briand, L. (2014). A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250.

[Arcuri and Fraser, 2011a] Arcuri, A. and Fraser, G. (2011a). On parameter tuning in search based software engineering. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE'11)*, pages 33–47.

[Arcuri and Fraser, 2011b] Arcuri, A. and Fraser, G. (2011b). On parameter tuning in search based software engineering. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE'11)*, pages 33–47, Berlin, Heidelberg. Springer.

[Arcuri and Yao, 2008] Arcuri, A. and Yao, X. (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)*, pages 162–168. IEEE.

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

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

[Baresel et al., 2003] Baresel, A., Pohlheim, H., and Sadeghipour, S. (2003). Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO'03)*, pages 2428–2441.

[Barton, 1994] Barton, R. (1994). Metamodeling: a state of the art review. In *Proceedings of the conference on Winter simulation (WSC'94)*, pages 237–244.

[Behera, 2014] Behera, R. N. (2014). Artificial neural network: A soft computing application in biological sequence analysis. *International Journal of Computational Engineering Research*, 4(4):1–13.

[Belbachir et al., 2012] Belbachir, A., Smal, J.-C., Blosseville, J.-M., and Gruyer, D. (2012). Simulation-driven validation of advanced driving-assistance systems. *Procedia-Social and Behavioral Sciences*, 48:1205–1214.

[Ben Abdessalem, 2018a] Ben Abdessalem, R. (2018a). Supplementary materials. `https://figshare.com/s/50193ea5652147d2f036`.

[Ben Abdessalem, 2018b] Ben Abdessalem, R. (2018b). Test scenarios for Pedestrian Detection Vision based system (PeVi). `https://sites.google.com/site/testingpevi`.

[Ben Abdessalem et al., 2016] Ben Abdessalem, R., Nejati, S., Briand, L. C., and Stifter, T. (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)*, pages 63–74, Singapore. IEEE.

[Ben Abdessalem et al., 2018a] Ben Abdessalem, R., Nejati, S., Briand, L. C., and Stifter, T. (2018a). Testing vision-based control systems using learnable evolutionary algorithms. In *Proceedings of the International Conference on Software Engineering (ICSE'18)*, page to appear, Gothenburg, Sweden. ACM.

[Ben Abdessalem et al., 2018b] Ben Abdessalem, R., Panichella, A., Nejati, S., Briand, L. C., and Stifter, T. (2018b). Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the International Conference on Automated Software Engineering (ASE'18)*, pages 143–154, Montpellier, France. IEEE.

[Beyer and Deb, 2001] Beyer, H.-G. and Deb, K. (2001). On self-adaptive features in real-parameter evolutionary algorithms. *Transactions on Evolutionary Computation*, 5(3):250–270.

[Blom et al., 1994] Blom, J., Jonsson, B., and Kempe, L. (1994). Using temporal logic for modular specification of telephone services. In *Proceedings of the International Workshop on Feature Interactions in Telecommunications Systems (FIW'94)*, pages 197–216, Amsterdam, Netherlands. IOS Press.

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

[Booker et al., 1999] Booker, A. J., Dennis Jr, J., Frank, P. D., Serafini, D. B., Torczon, V., and Trosset, M. W. (1999). A rigorous framework for optimization of expensive functions by surrogates. *Structural Optimization*, 17(1):1–13.

[Bosch, 2017] Bosch (2017). Driving safety systems for passenger cars.

[Bowman et al., 2010] Bowman, M., Briand, L. C., and Labiche, Y. (2010). Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Transactions on Software Engineering*, 36(6):817–837.

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

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

[Breiman et al., 1984] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth, Belmont, CA, U.S.A.

[Briand et al., 2016a] Briand, L., Nejati, S., Sabetzadeh, M., and Bianculli, D. (2016a). Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the International Conference on Software Engineering Companion (ICSE'16)*, pages 789–792, Austin, TX, US. ACM.

[Briand et al., 2006a] Briand, L. C., Labiche, Y., and Shousha, M. (2006a). Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7(2):145–170.

[Briand et al., 2006b] Briand, L. C., Labiche, Y., and Shousha, M. (2006b). Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7(2):145–170.

[Briand et al., 2016b] Briand, L. C., Nejati, S., Sabetzadeh, M., and Bianculli, D. (2016b). Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the International Conference on Software Engineering, (ICSE'16) 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 789–792.

[Britton et al., 2019] Britton, T., Jeng, L., Carver, G., and Cheak, P. (2019). Reversible debugging software - quantify the time and cost saved using reversible debuggers.

[Buehler and Wegener, 2005] Buehler, O. and Wegener, J. (2005). Evolutionary functional testing of a vehicle brake assistant system. In *Proceedings of the Metaheuristics International Conference (MIC'05)*, pages 157–162, Vienna Austria. -.

[Bühler and Wegener, 2004] Bühler, O. and Wegener, J. (2004). Automatic testing of an autonomous parking system using evolutionary computation. Technical report, SAE Technical Paper.

[Bühler and Wegener, 2008] Bühler, O. and Wegener, J. (2008). Evolutionary functional testing. *Computers & Operations Research*, 35(10):3144–3160.

[Caballero and Grossmann, 2008] Caballero, J. A. and Grossmann, I. E. (2008). An algorithm for the use of surrogate models in modular flowsheet optimization. *AIChE journal*, 54(10):2633–2650.

[Calder et al., 2003] Calder, M., Kolberg, M., Magill, E. H., and Reiff-Marganiec, S. (2003). Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141.

[Campos et al., 2017] Campos, J., Ge, Y., Fraser, G., Eler, M., and Arcuri, A. (2017). An empirical evaluation of evolutionary algorithms for test suite generation. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE'17)*, pages 33–48, Paderborn, Germany.

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

[Clarke et al., 1999] Clarke, Jr., E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. MIT Press.

[Cobb and Grefenstette, 1993] Cobb, H. G. and Grefenstette, J. J. (1993). Genetic algorithms for tracking changing environments. In *Proceedings of the International Conference on Genetic Algorithms (ICGA'93)*, pages 523–530, San Francisco, CA, USA. Morgan Kaufmann Publishers.

[Coello et al., 2007] Coello, C. A. C., Lamont, G. B., and Veldhuizen, D. A. V. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems. Genetic and Evolutionary Computation*. Kluwer Academic.

[Cohen, 1977] Cohen, J. (1977). *Statistical power analysis for the behavioral sciences (rev)*. Lawrence Erlbaum Associates, Inc.

[Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

[data, 2017] data, E. (2017). Experiments data.

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

[Deb, 2001] Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, Chichester, New York.

[Deb, 2014] Deb, K. (2014). Multi-objective optimization. In *Search Methodologies*, pages 403–449. Springer US.

[Deb and Agrawal, 1995] Deb, K. and Agrawal, R. B. (1995). Simulated binary crossover for continuous search space. *Complex systems*, 9(2):115–148.

[Deb and Beyer, 2001] Deb, K. and Beyer, H.-g. (2001). Self-adaptive genetic algorithms with simulated binary crossover. *Evolutionary Computation*, 9(2):197–221.

[Deb and Deb, 2014] Deb, K. and Deb, D. (2014). Analysing mutation schemes for real-parameter genetic algorithms. *International Journal of Artificial Intelligence and Soft Computing*, 4(1):1–28.

[Deb and Jain, 2014] Deb, K. and Jain, H. (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):577–601.

[Deb et al., 2002] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

[Douguet, 2010] Douguet, D. (2010). e-LEA3D: a computational-aided drug design web server. *Nucleic Acids Research*, 38:615–621.

[Efron, 1983] Efron, B. (1983). Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331.

[Efstathiou et al., 2014] Efstathiou, D., McBurney, P., Zschaler, S., and Bourcier, J. (2014). Efficient multi-objective optimisation of service compositions in mobile ad hoc networks using lightweight surrogate models. *Journal of Universal Computer Science*, 20(8):1089–1108.

[Emadi and Mahfoud, 2011] Emadi, D. and Mahfoud, M. (2011). Comparison of artificial neural network and multiple regression analysis techniques in predicting the mechanical properties of {A3} 56 alloy. *Procedia Engineering*, 10:589–594.

[Ferber et al., 2002] Ferber, S., Haag, J., and Savolainen, J. (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)*, pages 235–256, San Diego, CA, USA. Springer.

[Ferreira et al., 2015] Ferreira, G., Kästner, C., Pfeffer, J., and Apel, S. (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)*, page 17, Urbana, IL, USA. ACM.

[Ferrucci et al., 2013] Ferrucci, F., Harman, M., Ren, J., and Sarro, F. (2013). Not going to take this anymore: multi-objective overtime planning for software engineering projects. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*, pages 462–471.

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

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

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

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

[Gazzola et al., 2017] Gazzola, L., Micucci, D., and Mariani, L. (2017). Automatic software repair: A survey. *IEEE Transactions on Software Engineering*.

[Goldberg, 1989] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

[Golias et al., 2002] Golias, J., Yannis, G., and Antoniou, C. (2002). Classification of driver-assistance systems according to their impact on road safety and traffic efficiency. *Transport reviews*, 22(2):179–196.

[Goyal and Goyal, 2012] Goyal, S. and Goyal, G. K. (2012). Article: Study on single and double hidden layers of cascade artificial neural intelligence neurocomputing models for predicting sensory quality of roasted coffee flavoured sterilized drink. *International Journal of Applied Information Systems*, 1(3):1–4.

[Gross et al., 2012] Gross, F., Fraser, G., and Zeller, A. (2012). Search-based system testing: High coverage, no false alarms. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'12)*, pages 67–77.

[Group, 2017] Group, O. M. (2017). Object constraint language (ocl).

[Hagan and Menhaj, 1994] Hagan, M. T. and Menhaj, M. B. (1994). Training feedforward networks with the marquardt algorithm. *Neural Networks, IEEE Transactions on*, 5(6):989–993.

[Hall et al., 2011] Hall, M., Witten, I., and Frank, E. (2011). *Data mining: Practical machine learning tools and techniques (3rd edition)*. Morgan Kaufmann.

[Hamlet, 2002] Hamlet, R. (2002). Random testing. *Encyclopedia of software Engineering*.

[Harman et al., 2012a] Harman, M., Mansouri, S. A., and Zhang, Y. (2012a). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11.

[Harman et al., 2012b] Harman, M., Mansouri, S. A., and Zhang, Y. (2012b). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61.

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

[Haykin, 1998] Haykin, S. (1998). *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 2nd edition.

[Henard et al., 2013] Henard, C., Papadakis, M., Perrouin, G., Klein, J., and Traon, Y. L. (2013). Pledge: a product line editor and test generation tool. In *Proceedings of the International Software Product Line Conference co-located workshops (SPLC'13)*, pages 126–129, New York, NY, USA. ACM.

[Herrera et al., 2003] Herrera, F., Lozano, M., and S·nchez, A. M. (2003). A taxonomy for the crossover operator for real-coded genetic algorithms: An experimental study. *International Journal of Intelligent Systems*, 18(3):309–338.

[Holland, 1992] Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

[Honkela, 2001] Honkela, A. (2001). Nonlinear switching state-space models. *Master's thesis, Helsinki University of Technology, Espoo, Finland*.

[Hornik, 1991] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257.

[IEE, 2019] IEE (2019). International electronics & engineering. `https://www.iee.lu/`.

[Jackson and Zave, 1998] Jackson, M. and Zave, P. (1998). Distributed feature composition: a virtual architecture for telecommunications services. *IEEE TSE*, 24(10):831–847.

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

[Jin, 2011] Jin, Y. (2011). Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70.

[Jones et al., 2002] Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE '02)*, pages 467–477, New York, NY, USA. ACM.

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

[Karsoliya, 2012] Karsoliya, S. (2012). Approximating number of hidden layer neurons in multiple hidden layer BPNN architecture. *International Journal of Engineering Trends and Technology*, 3(6):713–717.

[Kato and Tsugawa, 2001] Kato, S. and Tsugawa, S. (2001). Cooperative driving of autonomous vehicles based on localization, inter-vehicle communications and vision systems. *Jsae Review*, 22(4):503–509.

[Kim et al., 2013] Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*, pages 802–811. IEEE.

[Knowles et al., 2006a] Knowles, J., Thiele, L., and Zitzler, E. (2006a). A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. Technical report, Computer Engineering and Networks Laboratory of Zurich.

[Knowles et al., 2006b] Knowles, J. D., Thiele, L., and Zitzler, E. (2006b). A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. Technical report, Computer Engineering and Networks Laboratory of Zurich.

[Kolesnikov et al., 2017] Kolesnikov, S., Siegmund, N., Kästner, C., and Apel, S. (2017). On the relation of external and internal feature interactions: A case study. *arXiv preprint arXiv:1712.07440*.

[Kong et al., 2015] Kong, X., Zhang, L., Wong, W. E., and Li, B. (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)*, pages 194–204.

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

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

[Koza, 1992] Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press.

[Kuhn et al., 2004] Kuhn, D. R., Wallace, D. R., and Gallo, A. M. (2004). Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421.

[Le Goues et al., 2012] Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (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)*, pages 3–13, Piscataway, NJ, USA. IEEE Press.

[Le Goues et al., 2012] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012). Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)*, 38(1):54–72.

[Li et al., 2015] Li, B., Li, J., Tang, K., and Yao, X. (2015). Many-objective evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)*, 48(1):13.

[Li et al., 2007] Li, Z., Harman, M., and Hierons, R. M. (2007). Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4).

[Lochau et al., 2012] Lochau, M., Oster, S., Goltz, U., and Schürr, A. (2012). Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Journal*, 20(3-4):567–604.

[Luke, 2013] Luke, S. (2013). *Essentials of Metaheuristics*. Lulu, Fairfax, Virginie, USA, second edition. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[MacKay, 1992] MacKay, D. J. C. (1992). Bayesian interpolation. *Neural Computation*, 4(3):415–447.

[Magnier and Haghighat, 2010] Magnier, L. and Haghighat, F. (2010). Multiobjective optimization of building design using TRNSYS simulations, genetic algorithm, and artificial neural network. *Building and Environment*, 45(3):739–746.

[Mariani et al., 2014] Mariani, L., Pezzè, M., Riganelli, O., and Santoro, M. (2014). Automatic testing of gui-based applications. *Software Testing Verification and Reliability*, 24(5):341–366.

[Matinnejad et al., 2014] Matinnejad, R., Nejati, S., Briand, L., and Brcukmann, T. (2014). MiL testing of highly configurable continuous controllers: scalable search using surrogate models. In *Proceedings of the International Conference on Automated Software Engineering (ASE'14)*, pages 163–174.

[Matinnejad et al., 2018] Matinnejad, R., Nejati, S., Briand, L., and Bruckmann, T. (2018). Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Transactions on Software Engineering*, page to appear.

[Matinnejad et al., 2015] Matinnejad, R., Nejati, S., Briand, L., Bruckmann, T., and Poull, C. (2015). Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology*, 57:705–722.

[Matinnejad et al., 2016] Matinnejad, R., Nejati, S., Briand, L. C., and Bruckmann, T. (2016). Automated test suite generation for time-continuous simulink models. In *Proceedings of the International Conference on Software Engineering (ICSE'16)*, pages 595–606, Austin, TX, US. ACM.

[Matlab, 2019] Matlab (2019). Matlab/simulink. `https://nl.mathworks.com/products/simulink.html`.

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

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

[Michalski, 2000] Michalski, R. S. (2000). Learnable evolution model: Evolutionary processes guided by machine learning. *Machine learning*, 38(1):9–40.

[Møller, 1993] Møller, M. F. (1993). A scaled conjugate gradient algorithm for fast supervised learning. *Neural networks*, 6(4):525–533.

[Monperrus, 2018] Monperrus, M. (2018). Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24.

[Nguyen et al., 2013] Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. (2013). Semfix: Program repair via semantic analysis. In *Proceedings of the International Conference on Software Engineering (ICSE '13)*, pages 772–781, Piscataway, NJ, USA. IEEE Press.

[Nguyen and Cripps, 2001] Nguyen, N. and Cripps, A. (2001). Predicting housing value: A comparison of multiple regression analysis and artificial neural networks. *Journal of Real Estate Research*, 22(3):313–336.

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

[Ong et al., 2003] Ong, Y. S., Nair, P. B., and Keane, A. J. (2003). Evolutionary optimization of computationally expensive problems via surrogate modeling. *AIAA journal*, 41(4):687–696.

[Oster et al., 2011] Oster, S., Zink, M., Lochau, M., and Grechanik, M. (2011). Pairwise feature-interaction testing for spls: potentials and limitations. In *Proceedings of the International Software Product Line Conference, Volume 2 (SPLC'11)*, page 6, Munich, Germany. ACM.

[Panichella et al., 2015] Panichella, A., Kifetew, F. M., and Tonella, P. (2015). Reformulating branch coverage as a many-objective optimization problem. In *Proceedings of the International Conference on Software Testing, Verification and Validation, (ICST'15)*, pages 1–10, Graz, Austria.

[Panichella et al., 2018] Panichella, A., Kifetew, F. M., and Tonella, P. (2018). Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158.

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

[Peng and Tang, 2011] Peng, F. and Tang, K. (2011). Alleviate the hypervolume degeneration problem of NSGA-II. In *Proceedings of the International Conference on Neural Information Processing (ICONIP'11)*, pages 425–434.

[Philomin et al., 2000] Philomin, V., Duraiswami, R., and Davis, L. (2000). Pedestrian tracking from a moving vehicle. In *Proceedings of the IEEE Intelligent Vehicles Symposium (IV'2000)*, pages 350–355, Dearborn, MI, USA. IEEE.

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

[Prehofer, 1997] Prehofer, C. (1997). Feature-oriented programming: A fresh look at objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, pages 419–443, Jyväskylä, Finland.

[Qi et al., 2014] Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. (2014). The strength of random search on automated program repair. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*, pages 254–265, New York, USA. ACM.

[Qi et al., 2015] Qi, Z., Long, F., Achour, S., and Rinard, M. (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)*, pages 24–36. ACM.

[Renieres and Reiss, 2003] Renieres, M. and Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering (ASE'03)*, pages 30–39.

[Rojas et al., 2017] Rojas, J. M., Vivanti, M., Arcuri, A., and Fraser, G. (2017). A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2):852–893.

[Sayyad and Ammar, 2013] Sayyad, A. S. and Ammar, H. (2013). Pareto-optimal search-based software engineering (POSBSE): A literature survey. In *Proceedings of the International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE'13)*, pages 21–27.

[Shalev-Shwartz and Ben-David, 2014] Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.

[Shams et al., 2006] Shams, M., Krishnamurthy, D., and Far, B. (2006). A model-based approach for testing the performance of web applications. In *Proceedings of the International Workshop on Software Quality Assurance (SOQUA'06)*, pages 54–61.

[Sheela and Deepa, 2013] Sheela, K. G. and Deepa, S. N. (2013). Review on methods to fix number of hidden neurons in neural networks. *Mathematical Problems in Engineering*, 2013:1–11.

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

[Software, 2014] Software, U. (2014). Increasing software development productivity with reversible debugging. Technical report, Undo Software.

[Suttorp and Igel, 2006] Suttorp, T. and Igel, C. (2006). Multi-objective optimization of support vector machines. In *Multi-objective machine learning*, pages 199–220. Springer, -.

[Syberfeldt et al., 2008] Syberfeldt, A., Grimm, H., Ng, A., and John, R. I. (2008). A parallel surrogate-assisted multi-objective evolutionary algorithm for computationally expensive optimization problems. In *Proceedings of the Congress on Evolutionary Computation (CEC'08)*, pages 3177–3184.

[TASS-International, 2019] TASS-International (2019). Prescan. `https://www.tassinternational.com/prescan`.

[Tian et al., 2018] Tian, Y., Pei, K., Jana, S., and Ray, B. (2018). Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the International Conference on Software Engineering (ICSE'18)*, page to appear, Gothenburg, Sweden. ACM.

[Tom et al., 2014] Tom, B., Lisa, J., Graham, C., Paul, C., and Tomer, K. (2014). Reversible debugging software. Technical report, University of Cambridge, UK.

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

[van der Horst and Hogema, 1993] van der Horst, R. and Hogema, J. (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)*, pages 109–121.

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

[van Lamsweerde, 2009] van Lamsweerde, A. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley Publishing, 1st edition.

[Van Veldhuizen and Lamont, 1998a] Van Veldhuizen, D. A. and Lamont, G. B. (1998a). Multiobjective evolutionary algorithm research: A history and analysis. Technical report, Air Force Institute of Technology.

[Van Veldhuizen and Lamont, 1998b] Van Veldhuizen, D. A. and Lamont, G. B. (1998b). Multiobjective evolutionary algorithm research: A history and analysis. Technical report, Air Force Institute of Technology.

[Vargha and Delaney, 2000] Vargha, A. and Delaney, H. D. (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):101–132.

[Wainer, 2009] Wainer, G. A. (2009). *Discrete-event modeling and simulation: a practitioner's approach*. CRC press.

[Wang et al., 2016] Wang, S., Ali, S., Yue, T., Li, Y., and Liaaen, M. (2016). A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE'16)*, pages 631–642, New York, NY, USA. ACM.

[Weimer et al., 2009] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering (ICSE'09)*, pages 364–374. IEEE.

[Wilmes and Windisch, 2010] Wilmes, B. and Windisch, A. (2010). Considering signal constraints in search-based testing of continuous systems. In *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops (ICSTW'10)*, pages 202–211.

[Windisch, 2010] Windisch, A. (2010). Search-based test data generation from stateflow statecharts. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO'10)*, pages 1349–1356.

[Witten et al., 2011] Witten, I. H., Frank, E., and Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition.

[Wohlin et al., 2012] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer-Verlag, Berlin Heidelberg.

[Wojtusiak and Michalski, 2004] Wojtusiak, J. and Michalski, R. S. (2004). The lem3 implementation of learnable evolution model: user's guide. In *Proceedings of the Machine Learning and Inference Laboratory, George Mason University, (MLI'04)*, pages 04–05, Fairfax, Virginie, USA. Citeseer.

[Xie et al., 2013] Xie, X., Chen, T. Y., Kuo, F.-C., and Xu, B. (2013). A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31.

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

[Zander et al., 2017] Zander, J., Schieferdecker, I., and Mosterman, P. J. (2017). *Model-based testing for embedded systems*. CRC press.

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

[Zeller, 2017] Zeller, A. (2017). Search-based testing and system testing: A marriage in heaven. In *Proceedings of the International Workshop on Search-Based Software Testing (SBST'17)*, pages 49–50, Piscataway, NJ, USA. IEEE.

[Zhan and Clark, 2006] Zhan, Y. and Clark, J. A. (2006). The state problem for test generation in simulink. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO'06)*, pages 1941–1948.

[Zhang et al., 2018] Zhang, M., Zhang, Y., Zhang, L., Liu, C., and Khurshid, S. (2018). Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the International Conference on Automated Software Engineering (ICSE'18)*, ASE 2018, pages 132–142, New York, NY, USA. ACM.

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

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

[Zitzler et al., 2000] Zitzler, E., Deb, K., and Thiele, L. (2000). Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8(2):173–195.

[Zitzler and Thiele, 1999a] Zitzler, E. and Thiele, L. (1999a). Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *Transactions on Evolutionary Computation*, 3(4):257–271.

[Zitzler and Thiele, 1999b] Zitzler, E. and Thiele, L. (1999b). Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271.

[Zuliani et al., 2013] Zuliani, P., Platzer, A., and Clarke, E. M. (2013). Bayesian statistical model checking with application to stateflow/simulink verification. *Formal Methods in System Design*, 43(2):338–367.