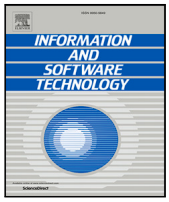
Information and Software Technology 160 (2023) 107225

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



An empirical comparison of combinatorial testing and search-based testing in the context of automated and autonomous driving systems

Florian Klück [a,1], Yihao Li [c,*], Jianbo Tao [b], Franz Wotawa [a,*]

[a] Christian Doppler Laboratory for Quality Assurance Methodologies for Autonomous Cyber–Physical Systems, Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/2, 8010, Graz, Austria
[b] AVL List GmbH, Hans-List-Platz 1, 8020, Graz, Austria
[c] School of Information and Electrical Engineering, Ludong University, Yantai, China

ARTICLE INFO

Keywords:
Test generation
Ontologies
Combinatorial testing
Search-based testing
Genetic algorithm

ABSTRACT

Context: More automated and autonomous systems are becoming daily use that implements safety–critical functions, e.g., autonomous driving or mobile robots. Testing such systems people depend on is challenging because some environmental interactions may not be expected during development but occur when those systems are in operation. Deciding when to stop testing or answering how to ensure sufficient testing is challenging and very expensive.

Objectives: For generating critical environmental interactions, i.e., critical scenarios, we present and compare two testing solutions focusing on generating critical scenarios utilizing combinatorial and search-based testing, respectively.

Methods: For combinatorial testing, we suggest using ontologies that describe the environment of an autonomous or highly automated system. For search-based testing, we rely on genetic algorithms. We experimentally compared the two testing approaches using two implementations of an industrial emergency braking function and random testing as the baseline. Furthermore, we compared the approaches qualitatively using several categories.

Results: From the experiments, we see that the combinatorial testing approach can find all different types of faults listed in Table 5 considering a combinatorial strength of 3. This is not the case for search-based and random testing in all experiments. Combinatorial testing comes with the highest combinatorial coverage. However, all approaches can reveal faulty behavior utilizing appropriate environmental models.

Conclusion: We present the results of an in-depth comparison of combinatorial and search-based testing. The be as fair as possible, the comparison relied on the same environmental model and other parameters like the number of generated test cases. The results show that combinatorial testing comes with the highest coverage and can find all different kinds of failures summarized in Table 5 providing a certain strength. Meanwhile, search-based testing is also capable of finding different failures depending on the coverage it can reach. Both approaches seem complementary and of use for the application domain of autonomous and automated driving functions.

1. Introduction

High-quality driving automation holds out the prospect of improved road safety and driving comfort but relies on complex software-based systems that are challenging to develop and test. Furthermore, Advanced Driver Assistant Systems (ADAS) and Automated Driving (AD) systems are considered safety–critical, meaning that failure or unintended system behavior might cause severe accidents. Ensuring quality and safety are highly important for such dependable systems, but the amount of software-based functionality built into modern vehicles is progressing, and automotive systems have become increasingly interconnected and complex [1]. While testing software and systems has always been challenging and resource-intensive, ensuring that a driving automation system shows correct behavior in any imaginable situation poses novel challenges. For instance, the role testing automobiles in the real world based on mileage accumulation is becoming less significant for these systems' quality and safety assurance. Accumulating hundreds

* Corresponding authors.
E-mail addresses: fklueck@ist.tugraz.at (F. Klück), yihao.li@ldu.edu.cn (Y. Li), Jianbo.Tao@avl.com (J. Tao), wotawa@ist.tugraz.at (F. Wotawa).
[1] Authors are listed in alphabetical order.

https://doi.org/10.1016/j.infsof.2023.107225
Received 1 November 2022; Received in revised form 31 March 2023; Accepted 3 April 2023
Available online 10 April 2023

of millions of accident-free test kilometers to statistically demonstrate that a self-driving vehicle is safer compared to a car steered by a human [2], is not only practically infeasible because of the huge number of potential interactions between a car and its environment but also dangerous and has led to fatal accidents [3], which should have been avoided. Hence, we see an urgent demand for verification and testing techniques for AD and ADAS.

Moreover, mileage-based testing may only provide limited quality assurance since driving on public roads mainly covers everyday traffic situations. In contrast, critical situations that challenge the system might only occur rarely. In any case, it is essential to reveal potential systematic failures already during development, long before vehicles are deployed on the public road, to avoid unnecessary risks and unscheduled repair efforts.

According to Koopman and Wagner [4], testing in simulation is a suitable approach, alongside other techniques such as error injection and formal proofs. In virtual testing, already at an early development stage, we can evaluate the complete driving system against diverse test data under realistic conditions. In addition, test results can be obtained much faster in simulation than in physical testing and at significantly lower costs. However, despite these advantages, there are also numerous challenges to performing virtual tests. First, the correct setup of an industry-scale simulation framework that provides realistic test conditions and reproducible simulation results is not trivial and highly error-prone [5]. Second, no general guideline prescribes how we shall perform virtual ADAS/AD testing. While we can generate virtual test scenarios automatically at relatively low costs, test execution and evaluation are much more expensive, demanding a certain quality of the generated test scenarios. Third, testing is an essential quality assurance task that aims to reveal the presence of errors in the ADAS/AD system instead of even attempting to prove the absence of failure, which is impossible in such complex systems, even in simulation.

In scenario-based testing, revealing the presence of an error corresponds to executing a test scenario that results in an unintended behavior of the ADAS/AD system under test or crash. However, since the number of relevant test parameters required for driving scenario definition is enormous, executing all possible parameter combinations (i.e., full factorial testing) is practically infeasible. Moreover, it may not be oriented towards a target, as most resulting test scenarios are not likely to reveal any unknown faults during execution. Especially with the increasing maturity of ADAS/AD functions, relevant areas in the input space are expected to become smaller and less frequent, thus, harder to identify. Therefore, more focused testing methods are required, such as combinatorial testing (CT) and search-based testing (SBT).

In previous research, e.g., [6–9], the authors already discussed the use of CT and SBT for ADAS/AD system testing based on experimental results obtained when using these techniques for testing an Automated Emergency Braking (AEB) function. While both methods aim to identify critical scenarios, their unique properties might be particularly suitable to address specific ADAS/AD testing challenges. However, an empirical method comparison is missing in the literature in the context of ADAS/AD testing. It is still being determined whether one method has a clear advantage over the other. Furthermore, it needs to be better understood for which exact purpose and at which ADAS/AD development stage each method's application would be most beneficial in the industrial context, considering limited time budgets for testing, development sprints, and potential simulator requirements. For example, since CT aims to identify all interaction failures and provides input space coverage, it might be most suitable for the final assessment and certification of high-quality driving features. In contrast, SBT might be ideal for identifying failures fast accompanying ADAS/AD feature development. Low system maturity at an early development stage may not require an elaborate testing strategy to reveal failure, and random testing may already be sufficient to reveal a failure at an early development stage.

The present work addresses these open questions by comparing the fault detection performance of CT and SBT in an industry-scale simulation framework, considering random testing (RT) as a comparison baseline. For the empirical study, we use an industry-scale simulation framework comprising a realistic 3D environment, vehicle, and physics simulation to test two prototypes of an Automatic Emergency Braking (AEB) system. Each AEB system includes a front object detection module and a brake control function. While it would be desirable to consider different ADAS/AD systems in this comparison, real industrial driving features for evaluation using such a framework are only available to a limited extent for research.

We are particularly interested in comparing each method's fault detection performance. As previously mentioned, in this work although the quality of generated tests is crucial as test generation is cheap, test execution is much more expensive. Therefore, even though a fair comparison is challenging because the working principles of CT, SBT, and RT are substantially different, we use the same sources for scenario generation (i.e., one shared combinatorial input model) and aim for similar test suite sizes to evaluate each method's performance as fair as possible.

We structure the paper as follows: First, we discuss related research in Section 2 and introduce the preliminaries behind the automatic test generation approaches in Section 3. Then, in Section 4, we discuss the AEB case study and the experimental setup for data collection. Next, in Section 5, we analyze the empirical result data in detail regarding the coverage of different crash events, failure detection probability, and input space coverage achieved by the respective CT, SBT, and RT test suites. Afterward, in Section 6, we discuss the obtained results and guidelines for selecting the appropriate testing method for a given task in the context of automated and autonomous driving validation. Finally, we conclude the paper and provide an outlook on future research in Section 7.

2. Related work

Of various advanced approaches used for software test generation, combinatorial testing and search-based testing would always be the first ones to pop into mind. On the one hand, CT has been successfully applied in different kinds of studies and could detect faults for different ordinary systems [10–17]. Wu et al. [18] report that CT has a better failure detection rate than random and adaptive random testing in 98 percent of 1,683 real test scenarios with available constraint information. Similarly, Arcuri and Briand [19] find that when constraints are present among features, random testing can fare arbitrarily worse than CT.

On the other hand, Zeller [20] argues that search-based techniques are best suited for testing at the system level. Bajaj and Sangwan [21] conclude that genetic algorithms have great potential in solving test case prioritization problems. Soltani et al. [22] observe that using a guided genetic algorithm can uncover failures that are undetected by classical coverage-based unit test generation tools.

Almansour et al. [23] find that GA algorithms are more effective than standard random search and adaptive random testing techniques in data-flow testing. When comparing CT and SBT, Petke et al. [24] report that GA is competitive only for pairwise testing for subjects with a few constraints. However, Henard et al. [15] find that compared to search-based approaches, existing t-wise tools fail to handle large software product lines. It is not hard to see that there is no conclusive judgment regarding the superiority between CT and SBT. Moreover, with the fast development of driving automation systems, CT [6,8,25–30] and SBT [31–39] have been increasingly used to test automated and autonomous systems that are complex and intelligent. Therefore, we intend to contribute to this research direction and present an in-depth analysis regarding the testing performance of CT and SBT in the ADAS and AD domain, substantially extending previous similar comparisons.

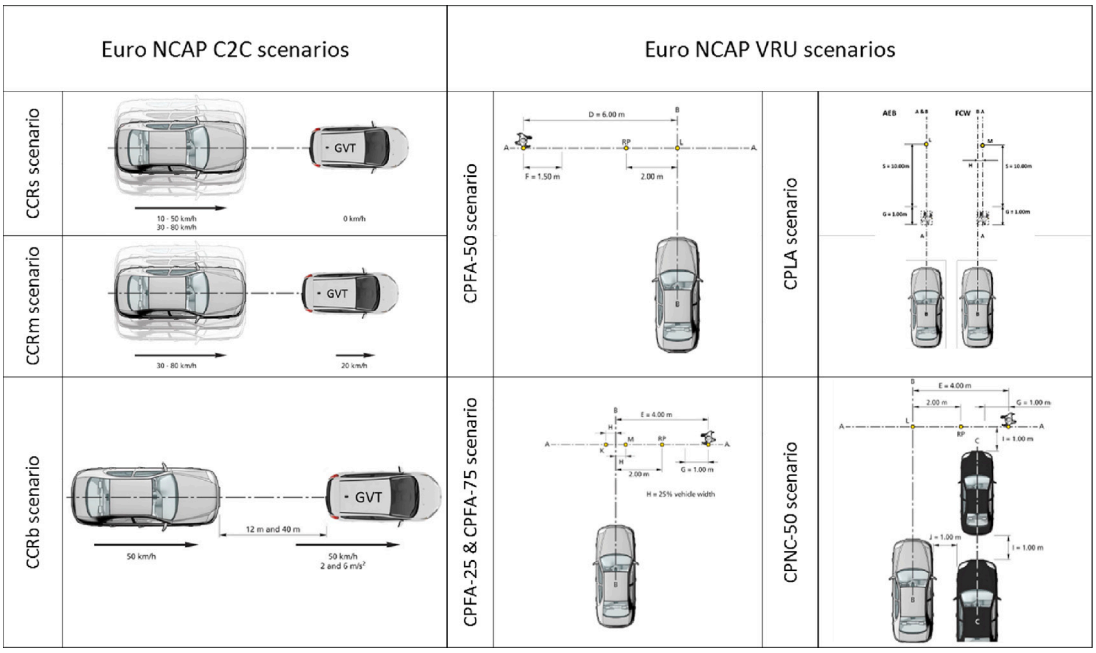


Fig. 1. AEB scenarios in EuroNcap protocol [40,41].

3. Preliminaries

In this section, we first discuss the desired properties of testing methods in the context of ADAS/AD and compare four state-of-the-art testing strategies. Next, we construct an ontology for AEB system testing considering relevant domain ranges according to European New Car Assessment Program (Euro NCAP) test protocols. Finally, we describe an automatic test generation process using the generated ontology as input for CT and SBT.

3.1. Desired properties of an ADAS/AD system testing method

An ideal testing method in the context of ADAS/AD system testing fulfills two basic requirements:

RQ1: The testing method can visit all relevant regions in the p-dimensional input space, where p refers to the number of test parameters. A relevant region describes an area in the input space where the driving function has the probability of failing, i.e., $prob(fail) > 0$. The parameter combinations that describe such relevant areas are unknown in advance and vary for different systems under test.

RQ2: The method can identify all relevant regions within a feasible time, allowing this method's practical application.

FFT: A conservative testing strategy such as full factorial testing (FFT) [42] guarantees to visit the whole input space and therefore fulfills RQ1. However, many parameters p need to be considered for testing in the context of automated driving. Depending on the magnitude of p, the combinatorial explosion leads to a test suite size that cannot be executed and evaluated within feasible time anymore, which violates RQ2.

RT: The same holds for random testing (RT) that would eventually visit all relevant regions, fulfilling RQ1. However, even for a moderate number of p, the time needed will be too large for practical application, which violates RQ2. Anyhow, visiting the whole input space is not target-oriented in general since many areas bear no risk for failure, i.e., $prob(fail) = 0$. Especially with the increasing maturity of the ADAS functions, relevant areas in the input space will become smaller and less frequent, thus harder to identify. Therefore, more focused testing methods are needed to meet the two requirements above. Hence approaches like CT and SBT are required.

CT: The CT (= partial factorial) approach overcomes the combinatorial explosion in the number of parameter combinations by considering only parameter combination subsets of strength t for testing, where t is sufficiently small to generate a test suite under given test resource. Therefore, CT performs better concerning RQ2 compared to RT and FFT. For other applications such as servers and databases, empirical studies showed that testing parameter combinations of strength $4 \leq t \leq 6$ is sufficient to reveal all faults (see [10]), suggesting that CT fulfills RQ1 if the combinatorial strength is selected high enough. However, the upper bound of t still needs to be identified in the context of ADAS/AD system testing.

SBT: The SBT approach uses a problem-specific cost function to guide the search towards areas in the input space with high failure probability. In the context of ADAS/AD, Key Performance Indicators (KPIs) are defined based on system requirements that specify desired properties of the driving function. An exemplary KPI for an Automatic Emergency Braking (AEB) system specifies the latest point in time where the system must initiate a braking maneuver to avoid a potential collision. This KPI is firmly based on an AEB internal measurement, the Time-to-Collision (TTC). Areas in the input space that result in low TTC can be seen as relevant, and a KPI violation or crash indicates failure. In favor of SBT and from a more general perspective, the various driving functions need to keep minimum distance towards other objects, show certain reaction times or fulfill other metric properties. These properties can easily be considered in a problem-specific cost function to guide the search towards failure. In addition, the SBT approach uses a random step to generate individuals in the initial population, ensuring that any area in the input space has an equal chance of getting visited. However, we cannot be sure that SBT covers all relevant areas; moreover, it is unknown how long it would take SBT to do so.

3.2. Sources for AEB ontology construction

Ontologies [43] are formal, explicit specifications of shared conceptualizations characterized by high semantic expressiveness required for increased complexity, which captures concepts and their relationships. An *ontology*[2] is a tuple $(C, A, D, \omega, R, \tau, \psi)$ where C is a finite set of

[2] Please refer to [6] for complete definitions of the ontology and the later introduced conversion algorithm.

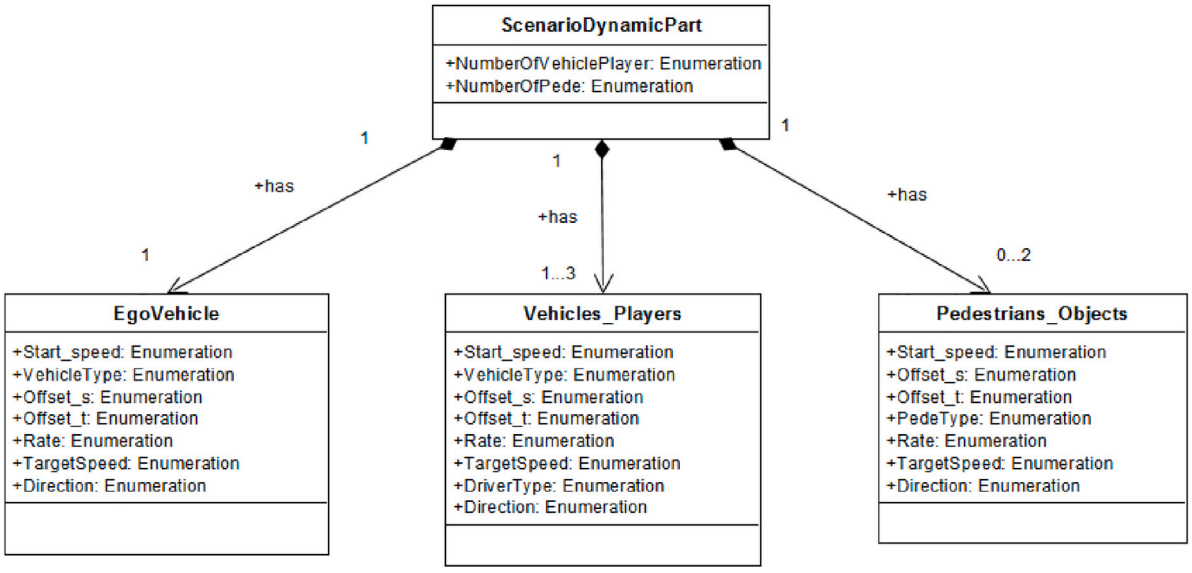


Fig. 2. Constructed AEB ontology based on EuroNCAP scenarios using UML.

Table 1
Euro NCAP Car-to-Car and Vulnerable Road Users Test Protocols.

Scenario Class	Description
Car-to-Car Rear Stationary (CCRs)	Stationary vehicle in front
Car-to-Car Rear Moving (CCRm)	Leading vehicle traveling at constant speed
Car-to-Car Rear Braking (CCRb)	Leading vehicle traveling at constant speed then decelerates
Car-to-Pedestrian Farside Adult (CPFA)	Adult pedestrian crossing Ego path from the farside
Car-to-Pedestrian Nearside Adult (CPNA)	Adult pedestrian crossing Ego path from the nearside
Car-to-Pedestrian Nearside Child (CPNC)	Child pedestrian crossing Ego path from behind an obstruction from the nearside
Car-to-Pedestrian Longitudinal Adult (CPLA)	Adult pedestrian walking in the same direction in front of the vehicle

concepts, A is a finite set of attributes which characterize concepts, D is a finite set of domain elements. Function ω maps concepts to a set of tuples specifying the attribute and its domain elements. R is a finite set of tuples stating that two concepts are related. Function τ assigns a type (i.e., composition or inheritance) to each relation. Function ψ maps composition relations to its minimum and maximum arity, specifying how many composer concepts a composer concept may comprise, ranging from 0 to any natural number.

Two prototypes of an AEB system will be later used in our case study. AEB is a vehicle safety function that uses sensor information to detect hazardous situations that potentially result in collisions. In an emergency, the system will take action instead of the driver to mitigate the impact or prevent a collision. As a well-known vehicle safety assessment organization, the Euro NCAP provides a safety performance assessment through rating. Regarding the AEB system, Euro NCAP introduced a test protocol for AEB Car-to-Car (C2C) and AEB Vulnerable road users (VRU) scenarios that describe scenario classes depicting frequent accident situations. The different test scenario classes defined by the protocols are summarized in Table 1. As shown in Fig. 1, the set-up for these scenario classes is provided in detail, together with relevant parameter value ranges, to define concrete test scenarios that are executable on proving ground or, as in our case, in simulation.

Based on the gathered information, we construct an ontology for the AEB[3] to cover all prescribed C2C and VRU scenario classes, as shown in Fig. 2.

Now, the ontology model consists of four concepts and three compositional relations, describing all the scenario participants. In the top concept $NumberOfVehiclePlayer$ and $NumberOfPede$ are defined. These parameters define the number of participants in our generated test scenario, where $NumberOfVehiclePlayer$ can vary from 1 to 3 vehicles and $NumberofPede$ can vary from 0 to 2 pedestrians. Zero pedestrians are considered because C2C scenarios do not include pedestrians. The three composer concepts in our ontology define the objects: $EgoVehicle$, $Vehicle\_Players$, and the $Pedestrians$. Each concept contains all necessary parameters and factors defined by the test protocols to describe object types and their dynamic and positional properties.

3.3. Combining ontology and combinatorial testing

In this section, we describe the four steps to implement test generation that combines ontology with CT:

The first step is the **ontology conversion**. Specifically, given an ontology as input, the output will be the corresponding CT input model, which comprises a set of variables, their domains, and constraints restricting certain value combinations. For the conversion, we have to map the concepts and given attributes of ontology to variables. The domains of the CT input models come from the domains of the concepts' attributes. We assume that ontologies always have only one root concept, i.e., a unique concept from which we start conversion and that the arity of the relations is always fixed to a particular finite value. Next, we will briefly outline the basic ideas behind an ontology to CT input model algorithm (`onto2ctim`) [6]. Generally speaking, in the `onto2ctim` algorithm, the domains, variables, and constraints of a lower level (or composer) concept cumulatively form that of its direct related higher level (or composer) concept. We must extract variables from the given concepts to develop a CT input model. The idea is to map concepts' instances into variables considering their relationships. An instance of a concept has particular values for its attributes. Moreover, we have to consider the relationships between

[3] In the AEB ontology we convert all the inheritance relations to composition relations in order to reduce the input model size and complexity.

```
[System]
Name: DynamicPart_test
[Parameter]
NumberOfVehiclePlayer (enum) : 1, 2, 3
NumberOfPede (enum) : 1, 2, null
EgoVehicle1_Start_speed (enum) : 0,     1.388888889     ,2.777777778    ,4.166666667    ,5.555555556    ,6.944444444    ,8.333333333
,9.722222222    ,11.11111111    ,12.5   ,13.88888889    ,15.27777778    ,16.66666667    ,18.05555556    ,19.44444444    ,20.83333333
,22.22222222    ,23.61111111    ,25     ,26.38888889    ,27.77777778    ,29.16666667    ,30.55555556    ,31.94444444    ,33.33333333
,34.72222222    ,36.11111111    ,37.5   ,38.88888889    ,40.27777778    ,41.66666667
EgoVehicle1_VehicleType (enum) : Audi_A3_2009_white
EgoVehicle1_Offset_s (enum) : 0
EgoVehicle1_Offset_t (enum) : 0
EgoVehicle1_Rate (enum) : 5, 6, 7, 8, 9, 10
EgoVehicle1_Target_Speed (int) : 14, 28, 42, 55
Pedestrains_Objects1_Start_speed (enum) : 0, 0.28, 0.56, 0.83, 1.11, 1.39, 1.6, 1.94, 2.22, 2.5, 2.8, null
Pedestrains_Objects1_Offset_s (enum) : 3, 4, 5,6,7,8,9,10,11,12,13, null
Pedestrains_Objects1_Offset_t (enum) : 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, null
Pedestrains_Objects1_Rate (enum) : 0, 1, 2, 3, 4, 5, 6, 7, 8, null
Pedestrains_Objects1_Type (enum) : Adult, null
Pedestrains_Objects2_Start_speed (enum) : 0, 0.28, 0.56, 0.83, 1.11, 1.39, 1.6, 1.94, 2.22, 2.5, 2.8, null
Pedestrains_Objects2_Offset_s (enum) : -9,-10,-11,-12,-13,-14,-15,-16,-17,-18,-19,null
Pedestrains_Objects2_Offset_t (enum) : 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, null
Pedestrains_Objects2_Rate (enum) : 0, 1, 2, 3, 4, 5, 6, 7, 8,null
Pedestrains_Objects2_Type (enum) : Adult, null
Vehicles_Players1_Start_speed (int) : 0,        1       ,3,4,6,7,8,10,11,12,13,15,16,18,19,20,22,23,25
,26,27,29,30,32,33,34,36,37,38,40,41
Vehicles_Players1_VehicleType (enum) : Audi_A3_2009_white, null
Vehicles_Players1_Offset_s (enum) :  0,null
Vehicles_Players1_Offset_t (enum) : 100, 105, 120, 125, 130, 135, 140, 145, 150, 155, 160, 165, 170, 175, 180, 185, 190, 195, 200, null
Vehicles_Players1_Rate (enum) :  3, 4, 5, 6, 7, 8, 9, 10, null
Vehicles_Players1_Target_Speed (int) : 0, 1     ,2
Vehicles_Players1_DriverType (enum) : DefaultDriver, null
Vehicles_Players2_Start_speed (int) : 0
Vehicles_Players2_VehicleType (enum) : Audi_A3_2009_white, null
Vehicles_Players2_Offset_s (enum) : 4.5,null
Vehicles_Players2_Offset_t (enum) : 100, 105, 120, 125, 130, 135, 140, 145, 150, 155, 160, 165, 170, 175, 180, 185, 190, 195, 200, null
Vehicles_Players2_Rate (enum) : 0, null
Vehicles_Players2_Target_Speed (int) : 0
Vehicles_Players2_DriverType (enum) : No Driver, null
Vehicles_Players3_Start_speed (int) : 0
Vehicles_Players3_VehicleType (enum) : Audi_A3_2009_white, null
Vehicles_Players3_Offset_s (enum) : 4.5,null
Vehicles_Players3_Offset_t (enum) : 100, 105, 120, 125, 130, 135, 140, 145, 150, 155, 160, 165, 170, 175, 180, 185, 190, 195, 200, null
Vehicles_Players3_Rate (enum) : 0, null
Vehicles_Players3_Target_Speed (int) :  0
Vehicles_Players3_DriverType (enum) : No Driver, null
```

Fig. 3. Generated input model of AEB use case.

concepts. Together with their domains, these variables are the input model for CT.

In the second step, we take the generated input model and use a **CT algorithm** for computing a combinatorial test suite. In CT [10], we are interested in generating a test suite comprising values for the given input variables so that for any subset of input variables of size t, all value combinations are considered in the test suite. Therefore, pair-wise testing considers all combinations of values but only for two input variables.

In **concretization** as the third step, abstract test cases generated from CT will be covered into concrete test cases using scripts or program mapping for test execution.

In the last step, we use the concrete test cases for **executing** the SUT. We rely on 3D and physical simulation in application areas like autonomous driving. In addition, we have to define a *test oracle* that allows for characterizing a test case to be passing or failing. In the case of the AEB function, we are using time-to-collision (TTC), a safety indicator introduced in the following subsection, to check whether a crash occurred during execution or not.

3.4. Combining ontology and search-based testing

For the search-based test case generation, we rely on a genetic algorithm as described in our previous work [9,31] and used the DEAP Python library [44], which provides a good starting point for implementation of evolutionary algorithms. However, this study generates the seed population directly from the ontology input model. Therefore, the seed population shares the same parameter and value sequences used for combinatorial test case generation. The algorithm that guides the automatic seed population generation is designed to consider the same constraints as combinatorial testing, and a dedicated function checks if every parameter and parameter value in the input model was used at least once. If not, the seed population is discarded, and the generation process is repeated. Therefore, the algorithm ensures that only valid individuals are included in the seed and guarantees full one-way parameter coverage, required for later computation of the test suite's combinatorial coverage, as described in Section 5.

In general, the seed population is a set of individuals with specific properties (genes) that comprise concrete scenario parameter values or, in other words, a set of candidate solutions to an optimization problem. The seed population forms the initial starting point for the genetic algorithm to optimize the underlying parameters, considering a predefined evaluation criterion. Each individual in the seed population represents a separate test case and comprises six genes, one for the Ego vehicle, three for the additional vehicle players, and two for the pedestrian players. The possible scenario types an individual can take have been described in Section 3 in Table 1 and are selected at random. The scenario type determines if a gene describes a concrete player instantiation or contains a *NULL*. Table 2 shows how the scenario players are represented as genes in the seed population concerning the selected scenario type.

Each gene furthermore comprises one concrete parameter value, specifying the positional or dynamical properties of the player. Table 3 provides an exemplary overview on how the chromosomes of individual 1 from Table 2 could be filled, considering parameter values from the input model.

In generation zero, individuals in the starting population are executed in simulation to evaluate their fitness score using predefined evaluation criteria. To assess an individual's fitness score, we consider the TTC value, a well-known and often used safety indicator in the ADAS domain. The TTC value at an instance x is defined as the remaining time for two objects to collide, given that they proceed on their trajectories and maintain their current velocities. Individuals with a higher fitness score (i.e., those which result in a low TTC

Table 2
Representation of scenario players as genes in the seed population.

Individual	Gene 0	Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Scenario Type
Player	Ego	Vehicle I	Vehicle II	Vehicle III	Pedestrian I	Pedestrian II	
Indv 0	E 0	VI 0	VII 0	VIII 0	PI 0	PII 0	E-3V-2P
Indv 1	E 1	VI 1	VII 1	*NULL*	PI 1	*NULL*	E-2V-1P
Indv 2	E 2	VI 2	VII 2	VIII 2	*NULL*	*NULL*	E-3V-0P
Indv n	E n	VI n	*NULL*	*NULL*	*NULL*	*NULL*	E-1V-0P

Table 3
Genes represent concrete player instantiations and comprise scenario parameters with values from the input model.

Gene	Scenario P0	Scenario P1	Scenario P2	Scenario P3	Scenario P4	Scenario P5	Scenario P6
Indv 1	StartSpeed	TargetSpeed	OffsetS	OffsetT	Rate	PlayerType	DriverType
E 1	11.11	28	0	0	5	v-model 1	n/a
VI 1	10	2	0	120	6	v-model 2	default driver
VII 1	0	0	4.5	175	0	v-model 1	no driver
VIII 1	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
PI 1	0.56	n/a	13	200	3	p-model	n/a
PII 1	*NULL*	n/a	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*

Table 4
Exemplary gene crossing between individual 1 and individual 2.

Individual Player	Gene 0 Ego	Gene 1 Vehicle I	Gene 2 Vehicle II	Gene 3 Vehicle III	Gene 4 Pedestrian I	Gene 5 Pedestrian II	Scenario Type
Indv 1	E 1	VI 1	VII 1	*NULL*	PI 1	*NULL*	E-2V-1P
Indv 2	E 2	VI 2	VII 2	VIII 2	*NULL*	*NULL*	E-3V-0P
Indv 1 crossed	*E 2*	VI 1	VII 1	*VIII 2*	*NULL*	*NULL*	E-3V-0P
Indv 2 crossed	*E 1*	VI 2	VII 2	*NULL*	*PI 1*	*NULL*	E-2V-1P

value) have a better chance of being part of the new population. For selection, we follow a best-of-three strategy, where we randomly pick three individuals and choose the one with the best score to be part of the next generation. With a certain probability, some individuals are selected for crossing and mutation to produce new offspring for the next generation. Since there is no general rule for defining the crossing and mutation probabilities, we set $p_{\text{crossing}} = 0.9$ and $p_{\text{mutation}} = 0.2$ based on our experience from initial experimental test runs. When two individuals are selected for crossing, they exchange genes to form two new individuals for the next generation.

For every gene in the individual, it is decided based on an independent crossing probability $ind\,p_{\text{crossing}} = 0.4$ if the gene is swapped or not. In Table 4, an example is given for crossing individual 1, and individual 2, where the genes zero, three, and four are exchanged. When an individual is selected for mutation, it is decided for each gene, based on an independent mutation probability $ind\,p_{\text{mutation}} = 0.2$ if the old parameter value in the gene is replaced with a randomly selected new parameter value. Offspring resulting from crossing or mutation is again checked against the constraints before it is included in the new population.

4. The AEB case study

This section discusses the experiment design, set-up, and execution of our AEB case study. In the case study, we apply two focused test generation methods, CT and SBT, to two different AEB implementations, considering RT as a reference method for comparison.

4.1. AEB ontology to input model conversion

As an initial preparation step for our case study, the AEB ontology (domain model) we described in Section 3, is converted into a test input model. The input model is shared by all three test scenario generation methods (CT, SBT, RT). For conversion, we apply the presented *onto2ctim* [6] algorithm to convert the AEB ontology into a corresponding combinatorial input model. The resulting test input model contains all test relevant parameters and value ranges as shown in Fig. 3.

In addition, our algorithm automatically generates constraints to exclude invalid parameter combinations in the subsequent scenario generation process, as described in [25]. Invalid parameter combinations are, for instance, overlapping pedestrian or vehicle positions, or if the number of pedestrians in a scenario is 1, the other pedestrian's parameters must be *"NULL"*. For the latter example, the generated constraint may look like this:

$$(NumberOfPedestrians = \text{“1”}) => \left(\begin{array}{l} Pedestrians\_2\_Start\_speed = \text{“}NULL\text{”} \wedge \\ Pedestrians\_2\_Offset\_s = \text{“}NULL\text{”} \wedge \\ Pedestrians\_2\_Offset\_t = \text{“}NULL\text{”} \wedge \\ Pedestrians\_2\_PedeType = \text{“}NULL\text{”} \wedge \\ Pedestrians\_2\_Rate = \text{“}NULL\text{”} \wedge \\ Pedestrians\_2\_TargetSpeed\_speed = \text{“}NULL\text{”} \wedge \\ Pedestrians\_2\_Direction = \text{“}NULL\text{”} \end{array} \right)$$

4.2. AEB systems under test and experimental setup

Both AEB implementations (i.e., AEB1 and AEB2) used in this study were compiled into the FMU (functional mockup unit) for the co-simulation, aiming to automatically detect an imminent forward collision and activate braking to avoid or mitigate a collision. However, the calibration parameters of these two systems, as well as their setup, are different, making their responses to the same driving scenario vary from one another. As explained in the following, for CT, SBT, and RT, we aim to generate a comparable set of test scenarios executed against the two AEB implementations.

CT-based Scenario Generation: We created the test suite using the CT tool from AVL, "Load Matrix for Software", which is based on the IPOG algorithm. The working environment we used to perform all the experiments was a Dell Precision Laptop with 2.8 GHz, Intel Core i7, and 32 GB memory running under Windows 7. Using the CT input model, the tool generated 978 test cases total for a combinatorial strength of $t = 2$ (CT2) and 21,418 for a combinatorial strength of $t = 3$ (CT3).

SBT-based Scenario Generation: For search-based test case generation, we aim for a test suite of similar size compared to combinatorial

testing with strength two (CT2). SBT utilizes the same input model and set of constraints to be comparable to CT. For strength two, the combinatorial test generation method computes a test suite that contains 970 unique test cases. For one search-based test run, we create a seed population with 40 individuals from the input model and terminate the genetic algorithm after five generations. Finally, we combine six independent test runs to obtain one test run set of similar size compared to combinatorial testing. We repeated this procedure to obtain ten test run sets ranging in size from 963 to 1,024 for AEB1 and ten test run sets ranging from 837 to 1,006 test cases for AEB2. Regarding the odd numbers between test run sets, we only count test cases executed in the simulation. The genetic algorithm does not re-simulate test cases selected for the next generation, which were not affected by crossing or mutation (i.e., stayed unchanged and already have a fitness score assigned). Only in the case when an identical test case results from crossing and mutation it needs to be re-evaluated since it has no fitness score assigned.

RT-based Scenario Generation: We conduct ten additional experiments using a random test generation approach for the same AEB function. The same simulation setup and configurations are used. For a clear comparison, each random generation experiment consists of the same number of test cases as the CT approach, i.e., 978 test cases. The test cases are generated randomly for random test generation, where we often rely on a uniform distribution. We conduct two steps in generating the random test cases to have a comparable test case set. Firstly, we take the parameters from the same input model from the CT approach. The value for each parameter is selected randomly from its domain defined in the CT input model. Initial random test cases without any combination constraints were then generated. Secondly, we applied constraints of the CT generation to remove invalid combinations. These constraints are also applied to the generated initial test cases, and the invalid ones, including the constrained parameter combinations, are removed. Finally, the random test cases with the same number of test cases and constraints are fed to the simulation execution.

4.3. Framework for automated test case generation, execution and evaluation

Our case study follows a comprehensible path for testing, from parameter selection to test scenario generation, execution, and final assessment. The ontology as a formal representation of scenario sources is converted into a CT input model that specifies parameters and value ranges for each test generation method. The resulting cases are converted into an XML format according to the OpenScenario specification,[4] to form executable test scenarios. Those scenarios are finally fed to the automated test execution and evaluation framework. An overview of the method for empirical data collection is shown in Fig. 4.

The framework comprises both AEB systems and co-simulation (i.e., 3D environment, traffic simulation, vehicle dynamic, sensor model, and so on). The software we use in our test environment is AVL VSM for vehicle dynamic simulation, VTD from VIRES[5] as a virtual driving environment platform and AVL Model.CONNECT is the co-simulation tool to connect all the software with the AEB function. The AEB systems are tested and assessed in the virtual test environment against the various generated test scenarios. For evaluation, we monitor the TTC, where the minimum TTC reveals how close a scenario came to a crash situation during execution. A TTC close to zero seconds represents a collision with a vehicle or pedestrian (i.e., AEB failure). In case of collision, we distinguish five different crash events, as shown in Table 5. Here, FCV refers to a crash where the Ego vehicle strikes the rear part of the leading vehicle. FCP1 and SCP1 refer to a front, respectively, side collision with the pedestrian one, depending on which side of the pedestrian's surrounding bounding box the Ego vehicle strikes. The same applies to FCP1 and SCP2 for pedestrian two. In Practice, side collisions are possible but may only occur rarely. Nevertheless, for completeness, we included this crash event in the study.

Table 5
Defined crash flags.

Crash flags	Description
FCV	Collision with front vehicle
FCP1	Front Collision with pedestrian 1
FCP2	Front Collision with pedestrian 2
SCP1	Side Collision with pedestrian 1
SCP2	Side Collision with pedestrian 2

Table 6
SBT, RT and CT test generation and crash summary for AEB1.

SBT	AEB1	Total	Pass	Fail	FCV	FCP1	SCP1	FCP2	SCP2
1	SBT01	1024	806	218	0	161	56	0	1
2	SBT02	983	857	126	0	31	94	1	0
3	SBT03	1002	802	200	0	120	79	0	1
4	SBT04	1010	743	267	0	216	50	0	1
5	SBT05	1005	807	198	3	149	41	4	1
6	SBT06	988	884	104	0	42	55	4	3
7	SBT07	971	743	228	0	199	24	0	5
8	SBT08	990	771	219	0	156	61	0	2
9	SBT09	963	764	199	0	165	34	0	0
10	SBT10	1021	801	220	2	134	82	0	2
RT	AEB1	Total	Pass	Fail	FCV	FCP1	SCP1	FCP2	SCP2
1	RT01	978	949	29	0	19	9	1	0
2	RT02	978	936	42	0	24	17	1	0
3	RT03	978	943	35	1	16	17	1	0
4	RT04	978	938	40	0	24	16	0	0
5	RT05	978	944	34	0	17	16	0	1
6	RT06	978	926	52	0	34	18	0	0
7	RT07	978	949	29	0	12	17	0	0
8	RT08	978	939	39	0	19	19	0	1
9	RT09	978	932	46	0	24	20	2	0
10	RT10	978	937	41	0	21	17	3	0
CT2	AEB1	Total	Pass	Fail	FCV	FCP1	SCP1	FCP2	SCP2
1	CT2_01	978	942	36	0	12	18	4	2
CT3	AEB1	Total	Pass	Fail	FCV	FCP1	SCP1	FCP2	SCP2
1	CT3_01	21418	19567	1851	2	604	999	159	87

5. Results

As part of the presented case study, we want to compare the properties and capabilities of each method in detecting the relevant areas in the input space within a feasible time. We considered relevant areas as a set of concrete scenario parameters resulting in crashes with the front vehicle, pedestrian 1, or pedestrian 2. We first summarize the distribution of AEB1 and AEB2 crash events detected by each method. Considering test coverage, we first discuss each method's t-way combinatorial test coverage.

5.1. AEB1 and AEB 2 crash event distribution

In Table 6, we summarize the evaluation result for each test generation technique applied on AEB1. If a crash event is observed during scenario execution, we consider the scenario as failed, otherwise passed. In the table presented, $Total$ refers to the number of test cases generated, $Pass$ summarizes the number of successful test cases, and $Fail$ refers to the number of failed test cases. For failing test scenarios, we distinguish between five different crash events, where for instance, FCV refers to the number of scenarios where a collision with the front vehicle was observed. The same applies for $FCP1$, $SCP1$, $FCP2$, and $SCP2$. Due to their probabilistic elements, search-based test generation (SBT) and random testing (RT) have each been executed ten times against

[4] See http://www.openscenario.org/.
[5] See https://vires.com/vtd-vires-virtual-test-drive/.

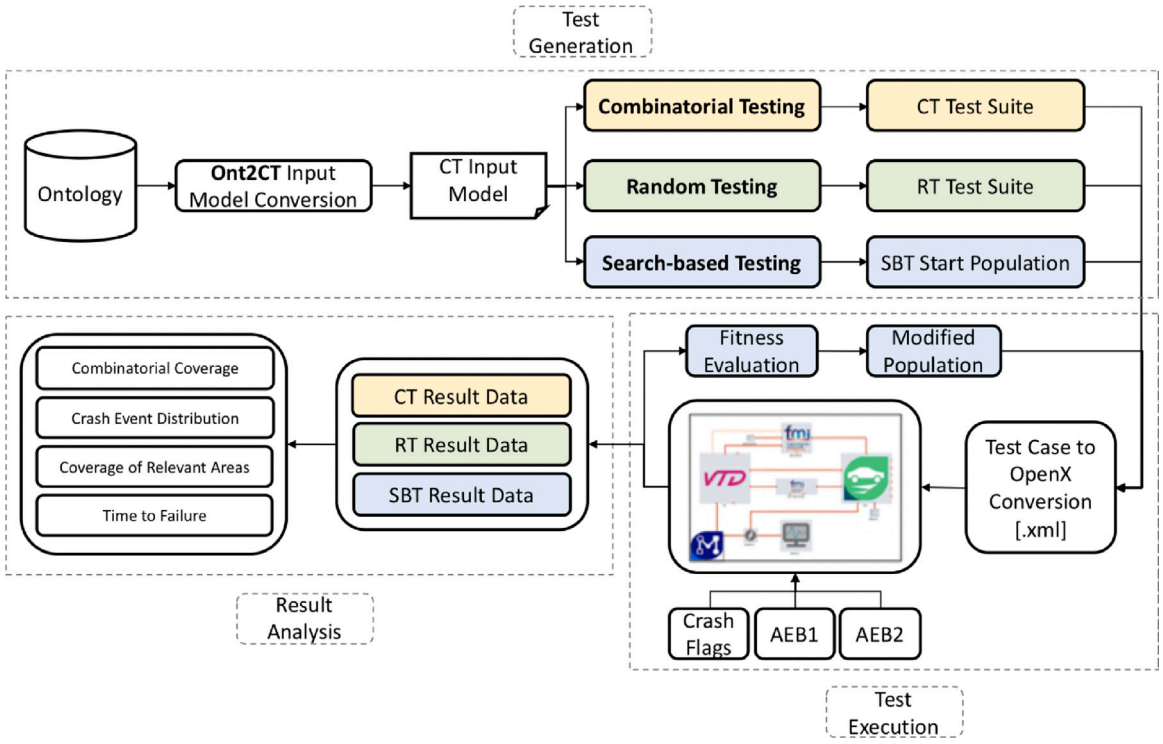


Fig. 4. One CT input model derived from an AEB ontology was shared for search-based, combinatorial, and random test generation. The empirical results from executing the corresponding test scenarios against two AEB systems were used for test coverage and efficiency analysis.

Table 7
SBT, RT and CT test generation and crash Summary AEB 2.

SBT	AEB2	Total	Pass	Fail	FCV	FCP1	SCP1	FCP2	SCP2
1	SBT01	976	867	109	95	3	11	0	0
2	SBT02	916	287	629	547	37	45	0	0
3	SBT03	897	309	588	455	51	65	0	17
4	SBT04	990	746	244	173	20	51	0	0
5	SBT05	923	285	638	514	68	55	0	1
6	SBT06	931	282	649	533	84	32	0	0
7	SBT07	925	264	661	548	69	28	0	16
8	SBT08	907	677	230	186	11	33	0	0
9	SBT09	1006	896	110	65	2	35	0	8
10	SBT10	946	837	109	97	2	10	0	0
RT	AEB2	Total	Pass	Fail	FCV	FCP1	SCP1	FCP2	SCP2
1	RT01	978	960	18	10	6	2	0	0
2	RT02	978	953	25	17	5	3	0	0
3	RT03	978	952	26	18	6	2	0	0
4	RT04	978	937	41	27	9	5	0	0
5	RT05	978	936	42	16	5	20	0	1
6	RT06	978	947	31	14	13	4	0	0
7	RT07	978	945	33	20	3	10	0	0
8	RT08	978	929	49	16	5	28	0	0
9	RT09	978	942	36	17	2	17	0	0
10	RT10	978	937	41	23	1	17	0	0
CT2	AEB2	Total	Pass	Fail	FCV	FCP1	SCP1	FCP2	SCP2
1	CT2_01	978	901	77	57	3	12	1	4
CT3	AEB2	Total	Pass	Fail	FCV	FCP1	SCP1	FCP2	SCP2
1	CT3_01	21418	19520	1898	1356	122	96	286	38

AEB1 and AEB2, marked from SBT01 to SBT10 and RT01 to RT10, respectively. The combinatorial test generation results with strengths two (i.e., CT2_01) and three (i.e., CT3_01) are also presented in the table.

As we observe from the summary table, FCV is the most challenging crash type to detect for all three methods. SBT generated five FCV scenarios in two test runs (SBT05 and SBT10) and RT in just one test run (RT03) out of ten. CT2 test scenarios did not cover an FCV crash event; however, CT3 detected two FCV crash events. The remaining four crash events are detected by each test generation method. However, some of them are less frequent as FCP2, which SBT only identifies in three out of ten test runs (SBT02, SBT05, and SBT06), or SCP2, which RT only covers in two out of ten test runs (RT05 and RT08).

Furthermore, we observe that each technique can generate a substantial number of scenarios that result in FCP1 or SCP1 events, indicating that AEB1 has noticeable defects in avoiding collision with pedestrian 1. Moreover, all methods detect collisions with Pedestrian 2 (P2) less frequently than Pedestrian 1 (P1). P1 crosses the road from the right side from the ego point of view. In some cases, P1 might be "occluded" by the parked vehicles or entering the road outside the radar's cone-shaped field of view. In contrast, P2 crossing from the left is never occluded by parked vehicles and travels a longer distance (i.e., across the oncoming lane), which is easier to detect and leaves more reaction time for the system to initiate breaking. Since P1 crashes are apparently "easier" to detect, SBT focuses on these regions in the input space in the proceeding search, reducing the overall probability of generating a scenario resulting in a crash with P2. As a result, SBT generates approximately five times the number of failed test cases compared to RT or CT2, where the average number of failed test cases generated by RT is similar to CT2.

In contrast, CT3 seems the most reliable technique for crash type coverage as all crash types categorized in Table 5 are guaranteed to be covered by its test suite. However, the CT3 test suite is twice as large as all executed SBT or RT test runs combined. Further, SBT and RT have also demonstrated to detect all crash types in the table eventually, even if not in every test run.

Similar to AEB1, Table 7 shows the evaluation results for each test generation technique concerning AEB2 crash event distribution. As we observe in Table 7, FCV is the most frequent crash type detected by all techniques, which is quite different from the results observed for AEB1. For SBT and RT, FCP2 becomes the most challenging crash type, which both methods fail to detect, followed by SCP2. Still, CT2 and CT3 can detect both crash types and all techniques have no difficulties detecting

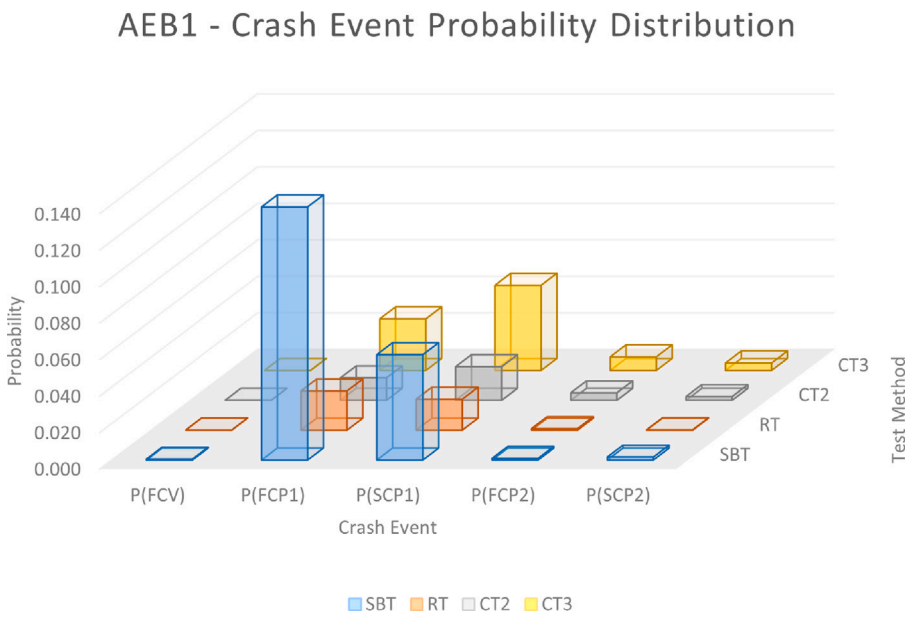


Fig. 5. AEB1 crash event probability distribution.

FCP1 and SCP1. Compared to AEB1, AEB2 seems more vulnerable, with noticeable defects that lead to crash scenarios with the front vehicle and pedestrian 1. Concerning the number of detected crash types, both CT2 and CT3 reliably detect all crash types in the table, followed by SBT, which has slight advantages over RT in detecting SCP2.

5.2. AEB1 and AEB 2 failure probability distribution

Fig. 5 shows each method's corresponding probabilities per crash event. Here, $P(FCV)$ refers to the probability that a generated test case can detect FCV during execution. We calculate:

$$P(FCV) = \frac{FCV}{Total} * 100[\%]$$

The same applies to $P(FCP1)$, $P(SCP1)$, $P(FCP2)$, and $P(SCP2)$. On average, test cases generated by SBT have a higher possibility of detecting a failure compared to RT, CT2, and CT3, especially for FCV (0.05%), FCP1 (13.78%), and SCP1 (5.77%). However, test cases generated by CT2 and CT3 show a higher probability of detecting FCP2 with (0.41%) and SCP2 (0.20%), respectively, compared to SBT and RT. Considering the probabilities shown in Fig. 6, except for FCP2, SBT always shows a higher crash detection possibility with FCV1 (34.71%), FCP1 (3.75%), SCP1 (3.90%), and SCP2 (0.45%) than RT, CT2, and CT3. CT3 and CT2 have the highest and second-highest possibility of detecting FCP2 with (1.34%) and (0.10%), respectively. Notably, for both AEB1 and AEB2, the probability distribution of crash events is similar between SBT and RT, which we can explain, considering SBT's random element used for generating the initial seed population. In contrast, the probability distribution of CT2 and CT3 is different, which can be explained by the structural coverage of the entire input space achieved by combinatorial testing.

5.3. T-way combinatorial coverage analysis

According to [11] most failures for ordinary systems are triggered by the combined combinatorial effect or interaction of input parameters (up to 6). In other words, testing all t-way combinations can provide strong assurance for fault detection. Therefore, from this point of view, we would like to see the performance of each test method by analyzing the t-way test case coverage.

In Table 8, we present the CT coverage information from 2-way to 5-way for SBT, RT, and CT2 for both AEB1 and AEB2. The given coverage values are averages over all runs for SBT and RT where it is worth noting that the value deviation over the runs is minor. SBT has a slightly lower coverage than RT concerning the 2-way, 3-way, 4-way, and 5-way coverage. A possible explanation may be that the search space was guided and thus focused more on specific regions. CT2 has undoubtedly reached 100% of 2-way coverage, 42% for 3-way, 13% for 4-way, and 4% for 5-way coverage, respectively. For CT2, each coverage is higher than the corresponding n-way (n=2,3,4,5) coverage for SBT and RT. Again, CT is more effective for testing rare events as it ensures coverage of all t-way interactions thus increasing the confidence in detecting those extreme faults.

Table 8
SBT, RT, and CT coverage results summary (2-way to 5-way). All values are average values over the runs.

SUT	Method	2-way coverage	3-way coverage	4-way coverage	5-way coverage
AEB1	$\overline{SBT}$	75,7%	30,2%	8,9%	2%
AEB2	$\overline{SBT}$	74,5%	29,2%	8,8%	2%
AEB1\AEB2	$\overline{RT}$	79,9%	34,1%	10,8%	3%
AEB1\AEB2	CT2_01	100%	42%	13%	4%

5.4. Threats to validity

A significant concern is the completeness of the input model (i.e., the ontology). In the current study, the ontology model only considers the dynamic part (i.e., all moving objects in the scenario), while the static parts, which represent road and traffic infrastructures, are not included in our model. As a result, our experiments' failures and critical scenarios detected and described may only work for the ontology with dynamic parts. Possibly they will not manifest when static parts are taken into consideration. In addition, the values used for fault detection need to increase as well. To alleviate this, we will extend our current ontology by integrating both static and dynamic parts and conduct further investigations on the testing performance between CT, SBT, and RT in the future. We believe our findings will be more comprehensive and general.

Another threat is the simulator fidelity for results interpretation. We did not validate the relationship between the simulated world and the real world in the paper. Therefore, critical scenarios and

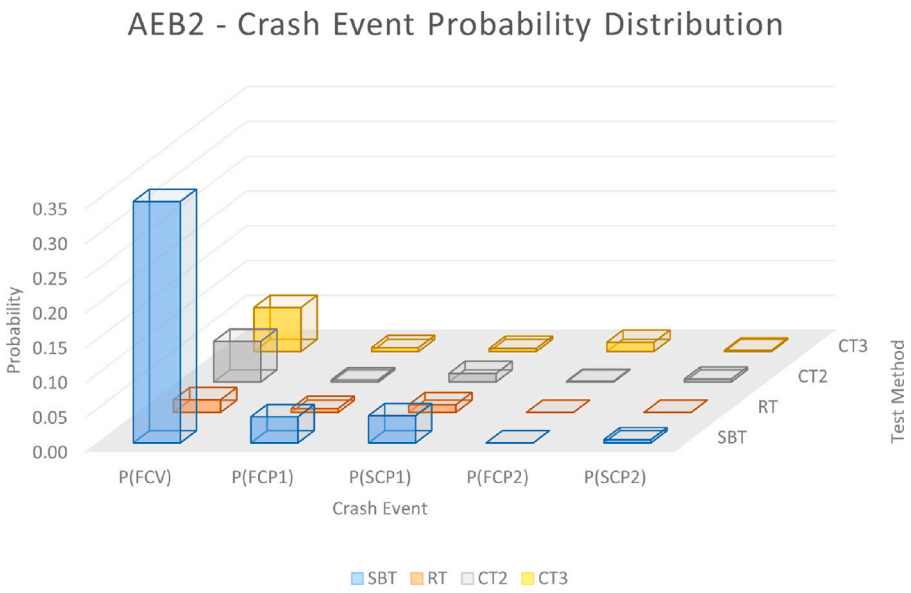


Fig. 6. AEB2 crash event probability distribution.

behaviors identified/detected in the simulator may deviate from the real world. However, this challenge applies to peer research as well. Meanwhile, the simulation software used in our study is the-state-of-art in the market today with the best accuracy, which can narrow the gap between simulation and reality to the maximum extent.

Last but not least, our experiments were done only based on the two AEB systems. To increase the functionality and system diversity, we used two implementations of the AEB system to conduct experiments, and these two systems also responded to different results and characteristics. Hence, we used this to compensate for the lack of diversity.

6. Discussion

As observed from the case studies of AEB1 and AEB2, all three methods (i.e., RT, CT, and SBT) can find critical scenarios, i.e., situations where the two SUTs fail to work as expected, resulting in a crash between the ego vehicle another car or a pedestrian. In general, CT and SBT tend to be complementary to each other. On the one hand, CT is more effective than SBT regarding the detected crash types. On the other hand, SBT is likely to be more efficient than CT having a higher crash detection probability (especially in case of specific crash types). RT is mediocre and does not appear to have a clear advantage over CT or SBT.[6] Therefore, in the following discussion, we will focus on the difference between SBT and CT to provide some guidance for selecting the appropriate testing method for a given task in the context of automated and autonomous driving validation. For viewing convenience, the following discussion details are further summarized in Table 9.

Prerequisites & requirements

Combinatorial testing: The CT approach aims to find critical interactions between parameters that lead to crashes or other fault scenarios. To carry out this approach, first, we need an ontology to capture the environmental information of the SUT. Second, we need a conversion algorithm to convert the ontology into an input model to use a combinatorial test case generation algorithm. Furthermore, we need constraints, which are very important for avoiding impossible combinations of parameters to create more reasonable scenarios and, in the end, an oracle to check the executions. For automated or autonomous function testing, we constructed the ontology using the environment of an ego-vehicle, like road construction, weather condition, traffic participants, and other driving and environmental features. Regarding combinatorial testing generation, algorithms are available for use. An oracle in this application could use time-to-collision and other relevant KPIs for the dedicated function.

Search-based testing: The objective of SBT is to find at least one set of concrete parameter values that results in an optimal solution, in our case, a critical driving situation. As a prerequisite, SBT requires formulating test cases as candidate solutions to an optimization problem and defining a cost function that guides the search for an optimal solution. In the case of testing automated and autonomous driving functions on the system level, the candidate solutions are virtual test scenarios for the system under test, where the search algorithm optimizes the underlying set of scenario parameters to result in faulty behavior of the system under test, often as a consequence of critical driving situations. Using a genetic algorithm to implement search-based testing requires selecting suitable genetic operators (i.e., crossing, mutation, and selection) and carefully fine-tuning the underlying parameters to guide the search towards optimal solutions efficiently. The scenario representation and configuration of genetic operators are described in detail in Section 3. The input model defines the permissible parameter and parameter value ranges, and the criticality of each scenario is determined based on a cost function that assigns a high fitness score to scenarios that result in a low time-to-collision value.

Additional effort:

Combinatorial testing: There is no additional effort when applying CT besides choosing and deciding the right combinatorial strength[7] for the function faulty behavior detection. This paper uses a combinatorial strength of 2 and later extends it to 3 for discussion purposes. Although for an ordinary system, a strength of 6 is sufficient for detecting all possible faults [10], for automated or autonomous driving functions, the required strength remains an open question.

Search-based testing: Besides selecting and fine-tuning an appropriate search algorithm, SBT requires defining an evaluation function to optimize the overall search procedure. Since there is no general procedure available on how to optimize search, additional effort is

[6] Nevertheless, RT may provide minimum guarantees on the probability of fault detection when test budgets and constraint information are limited [18, 19].

[7] The size of the set of arbitrary parameters where all combinations are considered.

Table 9
Qualitative comparison of CT and SBT.

	CT	GA	RT
Prerequisites	Construction of ontology Conversion of input model CT algorithm	Formulation of test cases to an optimization problem Definition of cost function and search algorithm	RT generation algorithm
Efficiency	Focus on finding parameter interactions for critical scenario More efficient in terms of detecting crash types	Focus on finding set of concrete parameter values Faster search and detection of critical scenario	A low-cost and quick solution compared to CT and GA in terms of quick system testing focusing on finding critical scenario according to oracle
Additional effort	Choosing and deciding the right combinatorial strength	Selection and fine tuning of search algorithm Definition an evaluation function	No further additional efforts
Guarantee	Guarantee of correct results based on the generated input model and combinatorial strength T-way test case coverage	No specific quarantees can be given from search-based testing Lower t-way test case coverage compared to CT	No specific quarantees can be given from random test generation

required for trial and error. In the case of using a genetic algorithm, this also includes selecting and carefully parametrizing appropriate genetic operators for crossing, mutation, and candidate selection. **Guarantees:**

Combinatorial testing: The approach's results are correct based on the underlying input model and CT strength. Therefore, the outcome correctness can also be interpreted as correct concerning the ontology based on the knowledge. Hence, if the quality of the ontology in terms of engineering knowledge know-how and completeness can be ensured, we can state the correctness of testing results based on the applied combinatorial strength.

Search-based testing: No specific guarantees can be given on the test results obtained from SBT. Once a failing test case is observed, we know that the system under test does not show the intended functionality, hence is faulty. If no failing test case is observed, the result is unspecified.

7. Conclusion and future work

In this paper, we conducted an in-depth comparison of the fault detection performance of two test generation methods that combine ontologies with CT, and SBT, respectively. An utterly fair comparison of the CT, SBT, and RT fault detection capability is impossible due to each testing strategy's substantially different working principle. In order to compare as fairly as possible, a unified comparison criterion had to be set, which in our case, was the number of generated test cases per method. The test suite size set by CT2, predefined by the combinatorial strength $t = 2$, has been used as a starting point for all three techniques. RT generated the exact and SBT approximately the same number of test cases after specific iterations. Originally it was only intended to consider a combinatorial strength $t = 2$ for the comparison. However, since CT2 did not trigger all failures, a combinatorial test suite of strength $t = 3$ was also considered to check if 3-way would eventually detect all five crash events.

The results from an AEB case study comprising two different AEB implementations indicate that CT and SBT are mutually complementary, where CT is more effective in detecting different crash types and the degree of combinatorial coverage. SBT is more efficient, showing a higher detection likelihood for specific crash scenarios. In addition, CT and SBT are generally superior to random testing concerning crash type detection and crash detection probability. Furthermore, we added a comprehensive discussion regarding the prerequisites, requirements, additional effort, and interpretation of outcome, as well as obtained guarantees when applying CT and SBT to automated and autonomous driving validation, which can be the basis for selecting the most appropriate method for testing different ADAS functions or AD systems.

Future work includes applying the CT, and SBT approaches to more complex autonomous driving functions using more sophisticated ontologies with more input parameters and testing values. In addition, we plan to perform a more detailed analysis of all the simulated results in terms of triggering parameter combinations and interactions. With this further analysis, we intend to summarize and compare the testing results of different automated driving functions to answer the question about a suitable combinatorial strength for application in the AD and ADAS domains. It is also worth investigating how to repeatedly use lower combinational strength to achieve similar fault detection performance compared to using a (much) higher strength directly.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, Austria and the Christian Doppler Research Association is gratefully acknowledged.

References

[1] J. Ibañez-Guzmán, C. Laugier, J.-D. Yoder, S. Thrun, Autonomous driving: Context and state-of-the-art, in: A. Eskandarian (Ed.), Handbook of Intelligent Vehicles, Springer, pp. 1271–1310, http://dx.doi.org/10.1007/978-0-85729-085-4_50.

[2] N. Kalra, S.M. Paddock, Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? Transp. Res. Part A Policy Pract. 94 (2016) 182–193, http://dx.doi.org/10.1016/j.tra.2016.09.010, URL http://www.sciencedirect.com/science/article/pii/S0965856416302129.

[3] F. Siddiqui, M. Laris, Self-driving Uber vehicle strikes and kills pedestrian, URL https://www.washingtonpost.com/news/dr-gridlock/wp/2018/03/19/uber-halts-autonomous-vehicle-testing-after-a-pedestrian-is-struck/.

[4] P. Koopman, M. Wagner, Challenges in autonomous vehicle testing and validation, 4, 2016, pp. 15–24, http://dx.doi.org/10.4271/2016-01-0128.

[5] M.E. Mostadi, H. Waeselynck, J.-M. Gabriel, Seven technical issues that may ruin your virtual tests for ADAS, in: 2021 IEEE Intelligent Vehicles Symposium (IV), pp. 16–21, http://dx.doi.org/10.1109/IV48863.2021.9575953.

[6] Y. Li, J. Tao, F. Wotawa, Ontology-based test generation for automated and autonomous driving functions, Inf. Softw. Technol. 117 (2020).

[7] F. Klück, F. Wotawa, G. Neubauer, J. Tao, M. Nica, Analysing experimental results obtained when applying search-based testing to verify automated driving functions, in: 2021 8th International Conference on Dependable Systems and their Applications, DSA, 2021, pp. 213–219, http://dx.doi.org/10.1109/DSA52907.2021.00033.

[8] H. Felbinger, F. Klück, Y. Li, M. Nica, J. Tao, F. Wotawa, M. Zimmermann, Comparing two systematic approaches for testing automated driving functions, in: 2019 IEEE International Conference on Connected Vehicles and Expo, ICCVE, 2019, pp. 1–6, http://dx.doi.org/10.1109/ICCVE45908.2019.8965209.

[9] F. Klück, M. Zimmermann, F. Wotawa, M. Nica, Performance comparison of two search-based testing strategies for ADAS system validation, in: C. Gaston, N. Kosmatov, P. Le Gall (Eds.), Testing Software and Systems, in: Lecture Notes in Computer Science, Springer International Publishing, Cham, 2019, pp. 140–156, http://dx.doi.org/10.1007/978-3-030-31280-0_9.

[10] D. Kuhn, R. Kacker, Y. Lei, J. Hunter, Combinatorial software testing, Computer (2009) 94–96.

[11] D. Kuhn, R. Kacker, Y. Lei, Introduction to Combinatorial Testing, in: Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, Taylor & Francis, 2013.

[12] L. Sh. Ghandehari, Y. Lei, R. Kacker, R. Kuhn, T. Xie, D. Kung, A combinatorial testing-based approach to fault localization, IEEE Trans. Softw. Eng. 46 (6) (2020) 616–645, http://dx.doi.org/10.1109/TSE.2018.2865935.

[13] X. Yuan, M.B. Cohen, A.M. Memon, GUI interaction testing: Incorporating event context, IEEE Trans. Softw. Eng. 37 (4) (2011) 559–574, http://dx.doi.org/10.1109/TSE.2010.50.

[14] X. Niu, C. Nie, H. Leung, Y. Lei, X. Wang, J. Xu, Y. Wang, An interleaving approach to combinatorial testing and failure-inducing interaction identification, IEEE Trans. Softw. Eng. 46 (6) (2020) 584–615, http://dx.doi.org/10.1109/TSE.2018.2865772.

[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, Y. Le Traon, Bypassing the combinatorial explosion: Using similarity to generate and prioritize T-wise test configurations for software product lines, IEEE Trans. Softw. Eng. 40 (7) (2014) 650–670, http://dx.doi.org/10.1109/TSE.2014.2327020.

[16] L. Hu, W.E. Wong, D. Kuhn, R. Kacker, How does combinatorial testing perform in the real world: an empirical study, Empir. Softw. Eng. 25 (4) (2020) 2661–2693.

[17] L. Hu, W.E. Wong, D. Kuhn, R. Kacker, S. Li, CT-IoT: a combinatorial testing-based path selection framework for effective IoT testing, Empir. Softw. Eng. 27 (2) (2022).

[18] H. Wu, C. Nie, J. Petke, Y. Jia, M. Harman, An empirical comparison of combinatorial testing, random testing and adaptive random testing, IEEE Trans. Softw. Eng. 46 (3) (2020) 302–320, http://dx.doi.org/10.1109/TSE.2018.2852744.

[19] A. Arcuri, L. Briand, Formal analysis of the probability of interaction fault detection using random testing, IEEE Trans. Softw. Eng. 38 (5) (2012) 1088–1099, http://dx.doi.org/10.1109/TSE.2011.85.

[20] A. Zeller, Search-based testing and system testing: A marriage in heaven, in: 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing, SBST, 2017, pp. 49–50, http://dx.doi.org/10.1109/SBST.2017.3.

[21] A. Bajaj, O.P. Sangwan, A systematic literature review of test case prioritization using genetic algorithms, IEEE Access 7 (2019) 126355–126375, http://dx.doi.org/10.1109/ACCESS.2019.2938260.

[22] M. Soltani, A. Panichella, A. van Deursen, Search-based crash reproduction and its impact on debugging, IEEE Trans. Softw. Eng. 46 (12) (2020) 1294–1317, http://dx.doi.org/10.1109/TSE.2018.2877664.

[23] F.M. Almansour, R. Alroobaea, A.S. Ghiduk, An empirical comparison of the efficiency and effectiveness of genetic algorithms and adaptive random techniques in data-flow testing, IEEE Access 8 (2020) 12884–12896, http://dx.doi.org/10.1109/ACCESS.2020.2966433.

[24] J. Petke, M.B. Cohen, M. Harman, S. Yoo, Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection, IEEE Trans. Softw. Eng. 41 (9) (2015) 901–924, http://dx.doi.org/10.1109/TSE.2015.2421279.

[25] F. Wotawa, Y. Li, From ontologies to input models for combinatorial testing, in: M.G. Medina-Bulo, R. Hierons (Eds.), Testing Software and Systems, Springer International Publishing, 2018, pp. 155–170.

[26] J. Tao, Y. Li, F. Wotawa, H. Felbinger, M. Nica, On the industrial application of combinatorial testing for autonomous driving functions, in: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW, 2019, pp. 234–240, http://dx.doi.org/10.1109/ICSTW.2019.00058.

[27] J. Chandrasekaran, Y. Lei, R. Kacker, D. Richard Kuhn, A combinatorial approach to testing deep neural network-based autonomous driving systems, in: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW, 2021, pp. 57–66, http://dx.doi.org/10.1109/ICSTW52544.2021.00022.

[28] H. Shu, H. Lv, K. Liu, K. Yuan, X. Tang, Test scenarios construction based on combinatorial testing strategy for automated vehicles, IEEE Access (2021) 1, http://dx.doi.org/10.1109/ACCESS.2021.3103912.

[29] G. Dhadyalla, N. Kumari, T. Snell, Combinatorial testing for an automotive hybrid electric vehicle control system: A case study, in: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, 2014, pp. 51–57, http://dx.doi.org/10.1109/ICSTW.2014.6.

[30] J. Duan, F. Gao, Y. He, Test scenario generation and optimization technology for intelligent driving systems, IEEE Intell. Transp. Syst. Mag. (2020) 1, http://dx.doi.org/10.1109/MITS.2019.2926269.

[31] F. Klück, M. Zimmermann, F. Wotawa, M. Nica, Genetic algorithm-based test parameter optimization for adas system testing, in: 19th IEEE International Conference on Software Quality, Reliability and Security, QRS, 2019, pp. 418–425.

[32] R. Ben Abdessalem, S. Nejati, L.C. Briand, T. 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, in: ASE 2016, Association for Computing Machinery, New York, NY, USA, 2016, pp. 63–74, http://dx.doi.org/10.1145/2970276.2970311.

[33] R. Ben Abdessalem, S. Nejati, L. C. Briand, T. Stifter, Testing vision-based control systems using learnable evolutionary algorithms, in: 2018 IEEE/ACM 40th International Conference on Software Engineering, ICSE, 2018, pp. 1016–1026, http://dx.doi.org/10.1145/3180155.3180160.

[34] O. Bühler, J. Wegener, Evolutionary functional testing, Comput. Oper. Res. 35 (2008) 3144–3160, http://dx.doi.org/10.1016/j.cor.2007.01.015.

[35] O. Bühler, J. Wegener, Automatic Testing of an Autonomous Parking System Using Evolutionary Computation, SAE Technical Papers, 2004, http://dx.doi.org/10.4271/2004-01-0459.

[36] R. Ben Abdessalem, A. Panichella, S. Nejati, L.C. Briand, T. Stifter, Testing autonomous cars for feature interaction failures using many-objective search, in: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering, ASE, 2018, pp. 143–154, http://dx.doi.org/10.1145/3238147.3238192.

[37] M.-C. Chiu, K.M. Betts, M.D. Petty, Automated search-based robustness testing for autonomous vehicle software, Model. Simul. Eng. 2016 (2016) 5309348, http://dx.doi.org/10.1155/2016/5309348.

[38] F. Klück, M. Zimmermann, F. Wotawa, M. Nica, Performance comparison of two search-based testing strategies for adas system validation, in: IFIP International Conference on Testing Software and Systems, ICTSS, 2019, pp. 140–156.

[39] F. Klück, Y. Li, M. Nica, J. Tao, F. Wotawa, Using ontologies for test suites generation for automated and autonomous driving functions, in: 29th IEEE International Symposium on Software Reliability Engineering (ISSRE2018), 2018.

[40] NCAP Euro, TEST PROTOCOL - AEB systems, 2017, Brussels, Belgium: Eur. New Car Assess. Programme (Euro NCAP).

[41] NCAP Euro, AEB VRU Test Protocol, TEST PROTOCOL - AEB VRU test, 2017.

[42] S. Oimoen, Classical Designs: Full Factorial Designs, STAT COE-Report-35-2018, 2019.

[43] C. Feilmayr, W. Wöß, An analysis of ontologies and their success factors for application to business, Data Knowl. Eng. (2016) 1–23, http://dx.doi.org/10.1016/j.datak.2015.11.003.

[44] F.M.D. Rainville, F.A. Fortin, M.A. Gardner, M. Parizeau, C. Gagné, Deap: A Python framework for evolutionary algorithms, in: Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, GECCO '12, ACM, New York, NY, USA, 2012, pp. 85–92, http://dx.doi.org/10.1145/2330784.2330799, http://doi.acm.org/10.1145/2330784.2330799.