STARS: A Tool for Measuring Scenario Coverage When Testing Autonomous Robotic Systems

Till Schallau[1 [0000−0002−1769−3486]], Dominik Mäckel[1 [0000−0002−7866−7927]], Stefan Naujokat[1 [0000−0002−6265−6641]], and Falk Howar[1,2 [0000−0002−9524−4459]]

[1] TU Dortmund University, Dortmund, Germany
till.schallau@tu-dortmund.de
[2] Fraunhofer ISST, Dortmund, Germany

Abstract. Extensive testing and simulation in different environments has been suggested as one piece of evidence for the safety of autonomous systems, e.g., in the automotive domain. To enable statements on the absolute number or fractions of tested scenarios, methods and tools for computing their coverage are needed. In this paper, we present STARS, a tool for specifying semantic environment features and measuring scenario coverage when testing autonomous systems.

1 Introduction

Autonomous systems are envisioned to operate in open and complex environments. Assuring the safety of systems in such environments is still an open challenge [14]. We rely on structured arguments (i.e., safety cases) about the safety of autonomous systems and test deployments for assessing the safety before rolling systems out at a large scale. One function of test deployments is to collect data for the estimation of reasonable risk by exposing the system to many different environments. The UL 4600 norm, e.g., lists the coverage of an operational design domain (ODD) as a mandatory aspect of testing automated systems [2]. There, an ODD specifies environment conditions (e.g., infrastructure, weather), under which an autonomous system is designed to operate [1]. Even before releasing prototypes into test deployments, we need to be reasonably sure of their safety.

To this end, scenario-based testing methods have been developed over the past couple of years [10, 20, 21]. While there is a strong focus on the automotive domain, case studies range from validating the safety of robotic systems [9], to generating accident scenarios [11], to testing of a nautic collision avoidance system [15], to testing the safe behavior of unmanned aerial vehicles in urban environments [19].

At both stages, it is important to decide whether the system was exposed to sufficiently many environments (or scenarios). There is, however, currently a lack of methods for measuring the degree of exposure to different environments or the coverage of scenarios. A prerequisite of measuring coverage is the definition of features that constitute different scenarios.

We have developed a conceptual framework, called STARS, for specifying scenario classifiers, determining the number of potentially observable scenarios,

and measuring their coverage by analyzing data recorded by the system under test [18] The STARS framework has so far been applied to different data sources in the field of automated driving systems. In previous works [18], we analyzed data generated with the CARLA [4] and other simulators. In ongoing projects, we analyze the KITTI dataset [7] containing real world driving data, and conduct a case study on a platooning controller for model-size vehicles.

In this paper, we present the STARS tool that we have developed to aid the analysis of environment coverage in different case studies. STARS automates the coverage analysis for user-defined application domains. It allows users to specify semantic environment features in a metric-time first order logic over observations and to construct scenario classifiers from these features. Moreover, the tool provides statistics over feature combinations, and can identify the absence of feature expressions in recorded data, enabling test coverage-driven test scenario selection. STARS is publicly available as open-source software.[3]

Related Work. Hildebrandt et al. present an approach for counting the number of unique environments experienced by an automated driving system [8]. In contrast to our work, they compute classifying signatures directly on sensor data and do not specify or compute semantic features. As a consequence, they analyze saturation but not coverage. Amersbach and Winner [3] present a method for defining the required test coverage of autonomous driving systems. They propose functional descriptions of scenarios that are similar to our predicate definitions. They do not provide an implementation. Li et al. [12] introduce the open-source research tool ComOpT which is capable of generating concrete scenarios based on higher-class descriptions of scenarios. While generating these scenarios, the framework maximizes the coverage of k-way combinatorial testing. Majumdar et al. [13] take a similar approach with their Paracosm tool and domain-specific language for defining and instantiating scenario components. Esterle et al. [6] formalize traffic rules for highway situations by using Linear Temporal Logic (LTL). They use Spot [5] to transform the defined formulas into deterministic finite automata. Rizaldi et al. [16] also use LTL formulas for monitoring traffic rules. They provide their formalizations as rules in Isabelle/HOL. All these works do not aim at coverage.

Outline. The remainder of the paper is structured as follows. Section 2 starts with an overview over the STARS tool. Section 2.1 describes how the it can be customized for particular domains, before showing the specification of features in Sect. 2.2 and the construction of classifiers in Sect. 2.3. Section 3 demonstrates the data analysis process and the computed metrics.

2 The STARS Tool

Figure 1 shows the data artifacts, specification formalisms, and processors involved when using the STARS tool. Only the two boxes depicted on the left need to

[3] `https://github.com/tudo-aqua/stars`

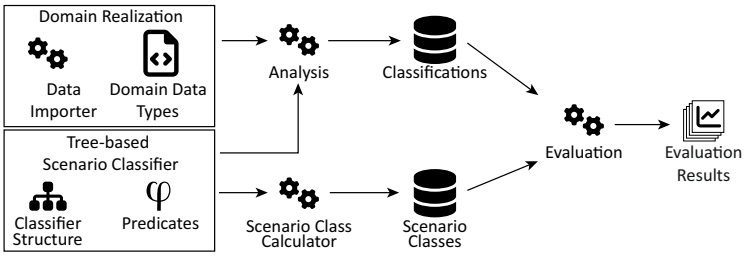

Fig. 1: Overview of the STARS coverage analysis tool[4]

be provided by users of the framework. While STARS already provides reusable realizations for some domains, the tool is generally designed to be domain-agnostic.

The first step when using STARS is to set it up for the targeted application domain (cf. Sect. 2.1), requiring two artifacts: the first artifact describes the *domain data types*, a data structure that abstractly captures properties of relevant entities (e.g., cars, robots, etc.) and the world they operate in (e.g., ground type, weather, etc.). Technically, this is done by implementing corresponding interfaces provided by the framework. STARS is implemented in Kotlin. Thus, any compatible JVM-based language (such as Java) can be used. The second artifact is a *data importer* that reads recorded data (e.g., from simulations or system deployments) and instantiates the domain data types. While the tool does not restrict the type of data that is imported this way, in our case-studies, we relied on the automatic binding of JSON format files to data classes.

The second step in the framework usage is modeling features of the operational design domain (ODD) of the analyzed autonomous system. Features can, e.g., be specified over data streams in formal logic (cf. Sect. 2.2). STARS provides a specification and decision implementation for the linear Counting Metric First-Order Temporal Binding Logic (CMFTBL) [18], but the tool allows for arbitrary computations of feature expression. Individual features can then be combined into tree-based classifiers that model combinatorial combinations of features as a basis for classifying perceived environments into scenario classes (cf. Sect. 2.3). Based on the tree-based classifiers, valid combinations of features can be calculated by STARS. These combinations form all possible scenario classes. Segments of recorded data can then be classified into one of these classes. During data analysis, the observed classifications and the set of possible scenario classes are computed. The tool provides implementations of various metrics and analyses, e.g., scenario class coverage, feature occurrence, scenario instance count, and missing features expressions (cf. Sect. 3), but also allows for the implementation of custom metrics for an application domain.

[4] Icons licensed under CC BY 4.0 `https://fontawesome.com/license/free`

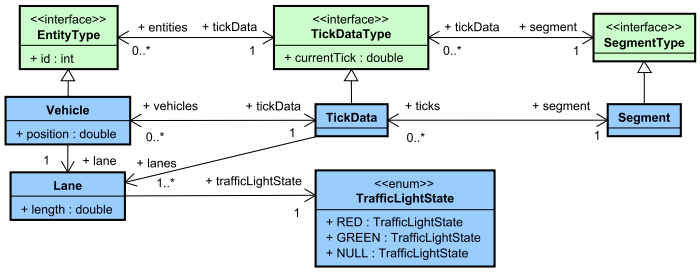

Fig. 2: Simple realization of the data types for an automated driving system

2.1 Instantiating STARS for an Operational Design Domain

To apply the framework to analyze recorded data of an autonomous system, it has to be instantiated first. In order to analyze data from various domains with STARS, the following parts have to be adapted.

Domain Data Types. The imported domain data has to implement the interfaces of the tool's generic data structure. On the top level, STARS requires a set of *segments*, which are the analysis units to be classified. The segments each hold a list of *ticks* in chronological order, where each tick holds a set of *entity* states for the tick's particular timestamp. Since the analysis is based on discrete events, the ticks' time distance (i.e., the sensor sampling rate) has to be selected such that no feature may occur in between. Additional domain information may also be added to the implementations of these basic data types. The domain data types are used to define the predicates used for classifying the data.

Predicates. To automatically analyze the recorded data, environment features need to be modeled and evaluated in a programmable manner. Therefore, existing temporal logics (cf. Sect. 2.2) or other specification and evaluation methods can be used to phrase and evaluate predicates for different scenario features. They are used to express domain properties that can be evaluated based on the given domain data.

TSC. In order to define the ODD of the autonomous system, the defined predicates are structured hierarchically using Tree-based Scenario Classifiers (TSC) (cf. Sect. 2.3). As every analyzed autonomous system has different intended functionalities and differing environments in which they have to operate, a dedicated TSC has to be defined. Additionally, the underlying logic evaluation system (cf. Sect. 2.2) and the evaluation metrics (cf. Sect. 3) can be replaced, making the STARS framework highly customizable for the applied domain.

To demonstrate the adaptability of STARS to different domains, we give a simple example of an implementation for the domain data types in Fig. 2. The three interfaces printed in green (i.e., *EntityType, TickDataType*, and *SegmentType*) show the generic data structure of the tool, while the domain-specific classes are printed in blue. For simplicity reasons, the implementing classes *Vehicle*, *TickData*, and *Segment* only contain minimal information about a vehicle's state and its relation to classes modeling the environment (i.e., *Lane* and

```
1  val hasRelevantRedLight = predicate(Vehicle::class){ ctx, v ->
2   eventually(v) { v -> hasRedLight.holds(ctx, v) && isAtEndOfRoad.holds(ctx, v) }
3  }
4  val hasRedLight = predicate(Vehicle::class){ ctx, v ->
5   v.lane.trafficLightState == TrafficLightState.RED
6  }
```

Listing 1: Kotlin implementation of predicates (1) and (2)

TrafficLightState). For every tick, we store the position of the vehicle on the lane it is driving on relative to the lane's length as well as the traffic light state for each lane. The available states are defined by the set {*RED, GREEN, NULL*}. State *NULL* indicates that there is no traffic light for the lane. They are later used to demonstrate the definition of predicates (cf. Sect. 2.2). A simple scenario classifier based on this domain data structure is given in Section 2.3.

2.2 Specifying Environment Features

For feature classification, STARS supports implementations of various logics, or evaluations, as long as they are executable on or callable from the JVM and utilize the implemented domain data types (cf. Sect. 2.1). The tool already implements the CMFTBL logic, which we introduced in a previous work [18]. An example property expressed in CMFTBL over our minimal domain data model (cf. Fig. 2) is to check for a *relevant red light*. A relevant red light is observed if at any point in time the traffic light belonging to the lane the vehicle v is driving on shows red. We also require v to be at the end of the road, which we define to hold if v has a distance of $3m$ or less to the end of its current lane. If *hasRelevantRedLight* evaluates to *true*, the feature is present in the analyzed segment. The predicate is formally defined in (1), hierarchically using predicates (2) and (3).

$$hasRelevantRedLight(v) := \Diamond(hasRedLight(v) \wedge isAtEndOfRoad(v)) \tag{1}$$

$$hasRedLight(v) := v.lane.trafficLightState = RED \tag{2}$$

$$isAtEndOfRoad(v) := v.pos \geq v.lane.length - 3 \tag{3}$$

These predicates can be directly implemented using the CMFTBL implementation provided by STARS, which supplies all required logical operators (i.e., *eventually, globally,* etc.) as Kotlin extension functions. Listing 1 illustrates the Kotlin implementation of predicates (1) and (2) to demonstrate how to express hierarchic predicate definitions. Predicate (3) follows analogously.

2.3 Constructing Scenario Classifiers from Features

We briefly introduce Tree-based Scenario Classifiers (TSCs) and how the tool supports their definition. A TSC describes a tree structure in which each node represents one feature that should be classified. An edge that is connected to a node requires a predicate which should evaluate to the boolean value *true* iff the feature represented by the node is present. Evaluating all predicates of a TSC for a given data segment results in a concrete TSC instance classifying the scenario that was observed in the analyzed segment. Consistency checking of the TSC is ongoing research and has to be ensured by the user for now.

```
TSC(root<Vehicle, TickData, Segment>{
 all("Root"){
  exclusive("Traffic Density"){
   leaf("High Traffic"){condition = {ctx->hasHighTrafficDensity.holds(ctx)}}
   leaf("Middle Traffic"){condition = {ctx->hasMidTrafficDensity.holds(ctx)}}
   leaf("Low Traffic"){condition = {ctx->hasLowTrafficDensity.holds(ctx)}}
  }
  bounded("Traffic Light", (0 to 1)){
   leaf("Has Green Light"){condition = {ctx->hasRelevantGreenLight.holds(ctx)}}
   leaf("Has Red Light"){condition = {ctx->hasRelevantRedLight.holds(ctx)}}
  }
  exclusive("Maneuver"){
   leaf("Lane Change"){condition = {ctx->changedLane.holds(ctx)}}
   leaf("Lane Follow"){condition = {ctx->followsLane.holds(ctx)}}
  }
 }
})
```

Listing 2: Kotlin implementation of the TSC from Figure 3

The nodes of the TSC are furthermore annotated with multiplicity constraints, which provide an upper and lower bound for the number of child nodes in a TSC instance, to ensure the validity of the data and correctness of the predicates. An example TSC is given in Figure 3. Different multiplicity constraints (A: All, X: Exclusive) and ranges (0..1) are used to ensure data integrity and the correct combination of predicates.These bounds are used to calculate all valid predicate combinations (i.e., scenario classes), and therefore TSC instances, that are observable. For example, an X constraint only allows exactly one related predicate to be valid. Therefore, predicates at edges originating from a node marked with X may never occur simultaneously. This means that the total amount of possible instances only increases by the number of predicates under the node, in contrast to an optional node (marked with O; not in this example) that allows for all combinations of those predicates. For the mathematical calculations of the combination count, we refer the interested reader to [18]. Listing 2 shows the implementation of the TSC.

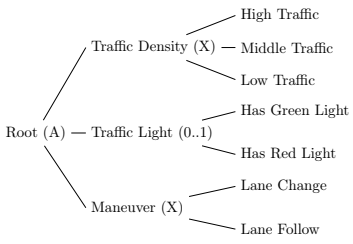

Fig. 3: Example Tree-Based Classifier

3 Measuring Scenario Coverage

STARS uses evaluation components for computing scenario coverage and other metrics (cf. Fig. 1) on classified segments of recorded data and supports the export of computed results to structured text files and plots. For some framework metrics, such as the saturation over time of the *scenario class coverage* for TSCs (cf. Fig. 4)[5], pre-defined plots are provided.

[5] Note: The data model used for the evaluation of the TSCs is described in [17].

Coverage and other Metrics. The unique contribution of STARS is the capability to compute *scenario class coverage*, the percentage of scenario classes that is present in a data set. The metric is computed by calculating the number of possible scenario classes and by counting observed ones. Other evaluation components count feature expressions, and compute the distribution of features and feature combinations. The *scenario class count* and the related *scenario class distribution* are useful for analyzing saturation over time and long-tail distribution of scenarios. STARS identifies scenario classes and feature combinations that are not observed, which can help to identify missing test cases. These metrics are independent of the domain data model and can be used in every analysis with STARS.

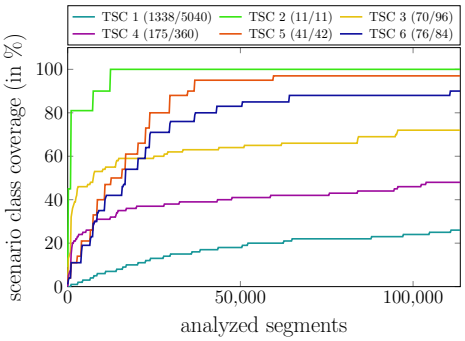

Fig. 4: The *scenario class coverage* metric plot, showing the coverage progress for six TSCs. The legend also details the amount of occurred unique scenario classes and possible scenario classes.

Multi-TSC Analysis. It is possible to define multiple TSCs and analyze them in parallel in order to decompose the combinatorial explosion of feature combinations where appropriate, e.g., to analyze coverage of weather-related and infrastructure-related features independently. Each TSC then specifies the analysis scope and defines which features are relevant for the current analysis (cf. Fig. 4).

Customization. Finally, users can define new plots, as STARS uses the lets-plot library that supports a variety of diagrams and plots. With custom evaluation components, domain-specific metrics over the domains' specific data models (e.g., average flight altitude in aeronautics) can equally be analyzed.

References

1. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. Standard J3016, SAE International (2021), https://www.sae.org/standards/content/j3016_202104/
2. Standard for safety for the evaluation of autonomous products. Standard ANSI/UL 4600-2023, UL Standards & Engagement (2023), https://ul.org/UL4600
3. Amersbach, C., Winner, H.: Defining required and feasible test coverage for scenario-based validation of highly automated vehicles. In: ITSC 2019. IEEE (2019). https://doi.org/10.1109/itsc.2019.8917534
4. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: An open urban driving simulator. In: PMLR 2017. vol. 78. PMLR (2017), https://proceedings.mlr.press/v78/dosovitskiy17a.html
5. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: ATVA 2016. Springer (2016). https://doi.org/10.1007/978-3-319-46520-3_8

6. Esterle, K., Gressenbuch, L., Knoll, A.C.: Formalizing traffic rules for machine interpretability. In: CAVS 2020. IEEE (2020). https://doi.org/10.1109/CAVS51000.2020.9334599
7. Geiger, A., Lenz, P., Stiller, C., Urtasun, R.: Vision meets robotics: The kitti dataset. International Journal of Robotics Research (IJRR) (2013)
8. Hildebrandt, C., von Stein, M., Elbaum, S.: PhysCov: Physical test coverage for autonomous vehicles. In: ISSTA 2023. ACM (2023). https://doi.org/10.1145/3597926.3598069
9. Huck, T.P., Ledermann, C., Kröger, T.: Simulation-based testing for early safety-validation of robot systems. In: SPCE 2020. IEEE (2020). https://doi.org/10.1109/SPCE50045.2020.9296157
10. ISO Central Secretary: Road vehicles - safety of the intended functionality. Standard ISO 21448:2022, International Organization for Standardization (2022), https://www.iso.org/standard/77490.html
11. Jenkins, I.R., Gee, L.O., Knauss, A., Yin, H., Schroeder, J.: Accident scenario generation with recurrent neural networks. In: ITSC 2018. IEEE (2018). https://doi.org/10.1109/itsc.2018.8569661
12. Li, C., Cheng, C.H., Sun, T., Chen, Y., Yan, R.: ComOpT: Combination and optimization for testing autonomous driving systems. In: ICRA 2022. IEEE (2022). https://doi.org/10.1109/icra46639.2022.9811794
13. Majumdar, R., Mathur, A., Pirron, M., Stegner, L., Zufferey, D.: Paracosm: A language and tool for testing autonomous driving systems (2021). https://doi.org/10.48550/arXiv.1902.01084
14. Mariani, R.: An overview of autonomous vehicles safety. In: IRPS 2018. IEEE (2018). https://doi.org/10.1109/irps.2018.8353618
15. Porres, I., Azimi, S., Lilius, J.: Scenario-based testing of a ship collision avoidance system. In: SEAA 2020. IEEE (2020). https://doi.org/10.1109/SEAA51224.2020.00090
16. Rizaldi, A., Keinholz, J., Huber, M., Feldle, J., Immler, F., Althoff, M., Hilgendorf, E., Nipkow, T.: Formalising and monitoring traffic rules for autonomous vehicles in Isabelle/HOL. In: iFM 2017. Springer (2017). https://doi.org/10.1007/978-3-319-66845-1_4
17. Schallau, T., Naujokat, S.: Validating behavioral requirements, conditions, and rules of autonomous systems with scenario-based testing. Electronic Communications of the EASST **82** (2023). https://doi.org/10.14279/tuj.eceasst.82.1222
18. Schallau, T., Naujokat, S., Kullmann, F., Howar, F.: Tree-based scenario classification: A formal framework for coverage analysis on test drives of autonomous vehicles. https://doi.org/10.48550/arXiv.2307.05106, submitted to NFM 2024
19. Schmidt, T., Hauer, F., Pretschner, A.: Understanding safety for unmanned aerial vehicles in urban environments. In: IV 2021. IEEE (2021). https://doi.org/10.1109/IV48863.2021.9575755
20. Weber, H., Bock, J., Klimke, J., Roesener, C., Hiller, J., Krajewski, R., Zlocki, A., Eckstein, L.: A framework for definition of logical scenarios for safety assurance of automated driving. Traffic Injury Prevention **20**(sup1) (2019). https://doi.org/10.1080/15389588.2019.1630827
21. Weng, B., Capito, L., Ozguner, U., Redmill, K.: Towards guaranteed safety assurance of automated driving systems with scenario sampling: An invariant set perspective. IEEE Trans. on Intelligent Vehicles **7**(3) (2022). https://doi.org/10.1109/tiv.2021.3117049