Implementation of an ODD Definition Language within an ODD Engine for Runtime OD Checking in Automated Driving Systems

Joschua Schulte-Tigges[1], Moritz Rumpf[1], Michael Reke[1], Alexander Ferrein[1]

Abstract—Each automated driving system (ADS) is developed for an individual operational design domain (ODD). Therefore, a safe operation of the automation system is only possible, while the vehicle's operational domain (OD) remains in its ODD. In this paper, we present the implementation of an ODD Definition Language within an ODD-Engine, which parses a given ontology and ODD definition first and executes ongoing checks of the OD during runtime. For evaluation, the ODD-Engine was fully integrated into an automated driving system. For the formal definition of ODDs taxonomies, ontologies and domain specific languages are already known from literature. And it is also common sense, that the operational domain of vehicles has to be monitored and the vehicle has to react if it leaves its ODD. With our work, we will show that it is possible, with extensions to known ODD definition languages, to create not only a formal definition of the ODD but also to describe the reaction and degradation of the automation system by restrictions. This enables the implementation of a fully functional ODD-Engine. We will show that our publicly available implementation of the ODD-Engine[1] is able to manage a huge number of ODD and ontology attributes within a short execution time. Furthermore, we also present first experimental results of the integration to a real automated driving system.

I. Introduction

The development of automated vehicles is an ongoing field of research and as of the automated driving levels defined by SAE [1], first Level 3 automated vehicles are deployed on public roads. That means for the first time, the vehicle is partially responsible for all driving tasks while driving, in comparison to Level 2 where the driver is still in charge to observe the automated driving features. For the automotive industries, that means they have to ensure safety for every occurring event in traffic. Looking into the SAE definition of Level 3 and 4 it says: "These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met" [1]. These conditions can be described for an automated driving feature as Operational Design Domain (ODD) [2], formally the domain the feature is designed to operate in. An ODD can be described by Operation Conditions (OC) such as, weather, time of day, traffic conditions, regions, availability of components and connectivity etc. [3], [4]. First existing features that are allowed to drive on public roads, such as Mercedes's Drive Pilot[2] or Honda's Traffic Jam Pilot[3] are very restricted and can only be used under suitable circumstances and are primarily used as traffic jam assistance. To decide whether a vehicle is inside its defined ODD and is therefore able to operate autonomously, the ADS has to monitor its environment, in other words its Operational Domain (OD). OD-Monitoring therefore is a necessary part when checking if the OD, the ADS is currently in, is within its ODD at any given time during operation. The OD can always just be an abstraction of the real environment. OD-Monitoring therefore requires a definition of domain attributes and a definition of the ODD. An abstraction of an OD can be defined as taxonomy or ontology, that then can be used as attributes to define guardrails and rules for the ODD of the Automated Driving System (ADS). An ADS monitoring framework has to observe all hardware and software modules and has to verify, whether all constraints of the ODD are fulfilled within a given time horizon. If a constraint is about to fail to be fulfilled, the monitoring framework should e.g. trigger a Take-over-Request (ToR). Such a monitoring framework must be integrated within the ADS architecture itself to guarantee safety on a deeper level and to be able to observe hard- and software components.

In this paper, we propose a rule-based ODD definition language for OD-Monitoring and -Validation, which we have completely implemented. Our implementation, referred to as the so called ODD-Engine, uses the proposed language in a YAML format. Therefore, attributes have to be defined in an OWL-based ontology in RDF format. During operation of the ADS, the OD is abstracted by attributes defined in the ontology and the ODD-Engine validates the interference between OD and ODD. To also specify the reaction and the level of automation degradation of AD vehicles, we introduce the keyword RESTRICTION as an extension of known description languages, allowing for specified targets rather than only defining what is inside or outside the ODD. In chapter II we present related work to outline the current state of ODD definitions and OD monitoring. In chapter III we explain the concept behind our proposed language and the ODD-Engine. Chapter IV describes the implementation of the language and the ODD-Engine as well as details of its ODD validation process. In chapter V we provide insight into a performance analysis of the ODD-Engine and

[1]FH Aachen University of Applied Sciences, Aachen, Germany {schulte-tigges;reke;ferrein}@fh-aachen.de, moritz.rumpf@alumni.fh-aachen.de

[1]https://github.com/MASKOR/ODD_Engine

[2]https://www.mercedes-benz.de/passengercars/technology/drive-pilot.html

[3]https://global.honda/en/tech/Automated_drive_safety_and_driver-assistive_technologies_Honda_SENSING_Elite

first evaluation results. Finally, we conclude our findings in chapter VI.

II. Related Work

Related to our work are first known taxonomies and ontologies as well as ODD definition languages, which are the basis for formal definitions of the ODD. Once the domain itself is identified for a particular use case, the ODD is formally described using formal languages, which is the basis for online monitoring of the OD in comparison to the ODD. Therefore, we also present related work to the ODD identification and the current state of OD monitoring as well.

A. Taxonomies & Ontologies

Before the ODD of a vehicle can be defined, one has to specify the attributes that can be used within the ODD description and typically include relevant contextual elements such as road types, weather conditions or traffic scenarios. Those attributes are specified in taxonomies or ontologies like the ISO 34503 [5], PAS 1883 [6], SAE J3259 [2] or ASAMs OpenXOntology [7]. These existing taxonomies or ontologies are only first drafts and show only which attributes may be necessary until the ODD has to be defined on a larger scale. Mendiboure et al. [8] created an overview of existing taxonomies and by comparing them, they concluded elements that can be used by a more generic taxonomy. Erz et al. [9] proposed a hierarchical structured ontology with attributes. It is used not only for ODD definition, but also for AV architectures and scenario-based testing to bring consistency into the full development cycle of AVs.

B. ODD Definition Languages

Once an ODD for an ADS is identified, it has to be specified in one way or the other. For example, the AVSC presents two narrative examples in their Best Practices for describing an Operational Design Domain, each describing the attributes in simple tables [10]. Another way to define ODDs are ODD Definition Languages, which is a commonly used method and is also used in this paper. These Domain-Specific-Languages (DSL) are especially designed for the purpose of defining the Operational Constraints (OC) of an ADS to describe an ODD. Reflecting on the literature for ODD definition languages, the OpenODD project by ASAM e.V. stands out as a notable initiative [11], released in April 2025. OpenODD is considered as the most advanced ODD definition language by Charmet et al. in their comparison [12]. An impact on OpenODD was the work of Irvine and Schwalb et al. [13], [14] as well as Rohne et al. [15], who presented a comprehensive definition language for ODDs. Bruce et al. [16] argue that the proposed languages ([13], [14], [15]) lack a standardized format to provide a solid foundation for their application and propose an ODD definition language designed in accordance to the ISO 34503 standard. But all projects on definition languages remain works in progress, as they have not yet been integrated into a real ADS or tested within one. Aniculaesei et al. [17] created a methodology to translate the YAML based tool chain described in [15] into SMT-LIB and to use SMT (Satisfiability Modulo Theories) solvers, in their case CVC5, to verify an OD. But also their methodology is not supported by a tool to validate the OD during runtime.

C. ODD-Identification

In 2018, the National Highway Traffic Safety Administration (NHTSA) proposed five guiding principles based on the literature at the time, which have continued to shape research since then: *1) The need for an ODD taxonomy; 2) Accounting for variations in operational environments; 3) Defining what constitutes an "operational scenario"; 4) Identifying ODD boundaries; 5) Recognizing the current ODD state (self-awareness/OD monitoring)* [18]. As the validation of Automated Driving Systems (ADS) matured, formal safety frameworks became essential. ISO 26262 [19] provided a foundation for addressing functional safety by mitigating the risks associated with malfunctions based on "Hazard Analysis and Risk Assessment" (HARA) and development related to the identified safety goals. Later, ISO 21448 [20] emerged to ensure behavioral safety in the form of "Safety of the intended functionality" (SOTIF), by introducing scenario-based hazard mitigation. This is used e.g. to estimate the unforeseeability of subsystems based on machine learning-based detection approaches. So, the mentioned standards institutionalized the 'Safety by Design' philosophy [21] and are now central to the development and validation processes of ADS technologies, hence in the definition of ODDs. Furthermore, Sun et al. [3] argue that real-time ODD monitoring is the foundation for SOTIF and argue that the solution to ODD monitoring under the SOTIF standard could be to leverage other data validation sources, such as road maps and vehicle-to-infrastructure communication, as done, e.g., in [22]. In addition to the already existing standards, Lee et al. [21] suggest a method to define an ODD for an ADS based on statistical data and a quantifiable risk tolerance, ensuring that if the system fails, the expected loss remains below a set threshold. In the work of Gyllenhammar et al. [23] they present strategies based on use cases that provide a convenient strategy for a collection of operating conditions. In a related context, Zhang et al. [24] present a survey method to support those defining ODDs in selecting appropriate techniques to identify critical scenarios that match their specific project needs. It is noted in [23] that there are several differing visions regarding the purpose of ODDs and that their scope varies. In response, Shakeri proposes in [25] a mathematical formalization of the terms OD, ODD, and ODD specification, a contribution that is highly valuable for achieving clarity and consistency in the field.

D. OD-Monitoring

Charmet et al. [12] state the importance of OD-Monitoring, while coming to the conclusion that operational domain monitoring will be a needed piece to ensure safe navigation of intelligent vehicles in the future. Different approaches for OD monitoring in regard to its ODD exist: H.

Torfah et al. [26] propose ways to make machine learning-based components monitorable. M. Gyllenhammar et al. [23] show ideas to monitor conditions that are challenging to predict, like weather. Colwell et al. [27] introduced the concept of restricting the runtime ODD based on subsystem degradation modes, to allow the ADS to remain in a safe Domain when some aspects are restricted. In the thesis of Sun [28] geo-fenced (Map Based) as well as probabilistic ways of ODD monitoring are discussed and evaluated. Charmet et al. propose in [29] a method to monitor ODD attributes that experience measurement uncertainty, by estimating a degree of membership that can be computed from an observed or predicted OD.

E. Conclusion of the related work

In conclusion of the related work, we can say that appropriate taxonomies and ontologies, as well as domain specific languages are available to describe ODDs. But in the current state, these languages lack possibilities to describe the reaction of the vehicle when the ODD is left. In this paper, we will show that our proposed introduction of the new keyword RESTRICTION is one possibility to describe behavioral targets for the vehicle when a rule is violated. Additionally, to our best knowledge from literature, there is currently no public implementation or description of a runtime ODD validation with full integration into an ADS available like we present it in this paper. As similar projects are only unimplemented concepts. Furthermore, the proposed ODD-Engine facilitates a more focused approach to functional safety. Assuming a well-defined ODD (e.g., in accordance with SOTIF), verification can concentrate on the ODD-Engine and whether the behavioral targets, defined within the restrictions, are correctly executed.

III. Concept

The language used in this paper is derived from the YAML-based concept proposed in [13], [14] and [15]. The goal of our ODD definition language is to be able to define rules of an ODD and bind these rules to concrete restrictions of the system or its subsystem. Restrictions can be a speed reduction, being prohibited to overtake, or the triggering of maneuvers like a ToR or a Minimal-Risk-Maneuver (MRM). But others are also conceivable. In order to define ODDs for various use cases, the language must be domain-specific. Therefore, this concept uses a freely definable ontology that defines all usable attributes, which can then be used from OD and ODD. A software module, called ODD-Engine, parses both ontology and ODD during initialization and compares the OD to the ODD ongoing during runtime. Furthermore, it checks whether the ODD uses the correct attributes of the ontology. The data of the OD are obtained from the OD Monitoring and the data will be put into the ODD-Engine during runtime. Figure 1 shows how these components work together. The system uses the format OWL (Web Ontology Language)[4] for the ontology

[4]https://www.w3.org/OWL/

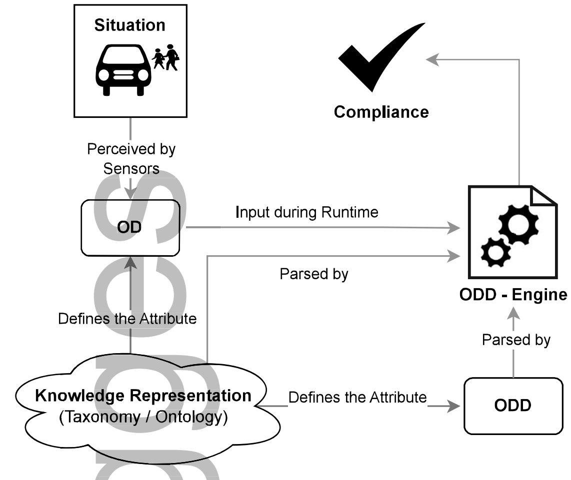

Fig. 1. Visualization of the ODD concept. Green arrows show the actions during the runtime while the yellow arrows show the actions in the initialization phase. The blue arrows show the dependency from the Knowledge Represantation.

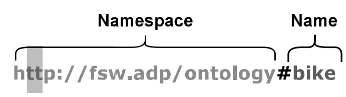

Fig. 2. Shows an IRI of a OWL resource

because it provides a machine-readable way to define complex relationships and concepts. It is built as an extension of the Resource Description Framework (RDF), which is a W3C standard for describing resources on the Internet. OWL can be written in different syntax versions such as turtle or OWL/RDF, but their main idea is to define knowledge as subject–predicate–object triples. For example, a triple can be a car that is of type class: *car(subject) type(predicate) class(object)*. This in turn can be a subclass of the class Vehicle: *car(subject) subclassOf(predicate) vehicle(object)*. Each resource of a triple is stored in OWL as an IRI, whereby the resource consists of a name and a namespace as shown in Figure 2. In addition to domain specificity, the language has additional requirements. It should be composable for better extendability and it should be easy to read to be usable by a wide community. Therefore, it uses the simple and common YAML-Format and extends it with various keywords. YAML is an extension of JSON and enables the representation of hierarchical data with a simple human- and machine-readable syntax. To use data from the OD, instances of an ontology class, called objects, have to be defined within the keyword OBJECT (See Listing 1). In this block, the objects are defined by a unique name and the ID of the class to which the object is referring. The object's values can then be filled, with either a subclass of the class hierarchy or a property of the class. The DNAMESPACE keyword defines a default namespace, eliminating the need to prefix it before every use of an ontology resource. Currently, only the data properties are supported, linking class attributes to specific data types (e.g., string, double, integer, etc.). To utilize data properties, a syntax similar to that of an object-oriented language is

used. First, the identifier of the object is specified and then, separated by a dot, the desired data property. If the results of an expression are used multiple times, it is helpful to define them not multiple times in the rules, but to store them in a variable, called ODD-variable, with the keyword VARIABLE. It allows declaring variables from different data types, which can be reused in further ODD-variables and rules. Listing 2 shows the syntax of the ODD-variable definition. In addition to the value, each ODD-variable has a unique name and a data type.

```
DNAMESPACE: http://fsw.adp/ontology#
OBJECT:
  egoADS: car
  path: path
  nextArea: area
```

Listing 1. Shows the definition of objects. Every object has an unique identifier a its class in the ontology.

```
VARIABLE:
  VelInMs:
    double: egoADS.velocity / 3.6
  nextAreaWithinToR:
    bool: path.distToNextArea / VelInMs < 30
  nextAreaWithinMRM:
    bool: path.distToNextArea / VelInMs < 20
```

Listing 2. Shows the definition of ODD-variables in the language. Every ODD-variable has an unique identifier, a datatype and a value.

```
RULES:
  approachingRestrictedArea:
    WHEN AND:
      nextAreaWithinToR: true
      nextArea: restrictedArea
```

Listing 3. Example of Keyword RULES. The Example shows a rule that is satisfied when ADS is on a highway and has a visibility greater than 50%.

The core of the ODD are the rules. Rules are defined within the keyword RULES, as shown in Listing 3. Every rule has a unique name, which can be used to reuse the rule in further rules, and a set of conditions. A condition consists of an identifier and a target value. In this version, a value of a condition can either be a Boolean, a sub-value of the underlying ontology or an expression whose result is evaluated as a Boolean. Expressions are introduced with Keyword EXP. The sub-values can also be a list of target values, instead of a single value, of which only one of the values must apply. One or more conditions are linked with one of these Keywords called logic blocks:

- WHEN OR: At least one of the following conditions must be satisfied.
- WHEN AND: Every following condition must be satisfied.
- EXCEPT WHEN EITHER: None of the following conditions must be satisfied.

Rules can consist of multiple of these Keywords. At this point, the rules just return true or false, but are not able to bind rules to concrete restrictions. This is the reason for our introduction of the new Keyword RESTRICTION, shown in Listing 4. Each restriction has a unique name that must not match the name of any rule. Unlike the rules, a restriction cannot be reused. It is intended as an end point for the rules. It uses the nested keyword TARGET which specifies a target key that will be returned if the rules are satisfied. The interpretation of the key lies with the system. Furthermore, there is the second nested Keyword RULE, that allows to bind rules to the restriction. The syntax is analogous to a rule created in RULES and can call the rules from that. Every rule, object, or ODD-variable to which a rule is referring, must be declared before its reference.

```
RESTRICTION:
  tor:
    TARGET: TOR
    RULE:
      WHEN OR:
        approachingRestrictedArea: true
        EXP: egoADS.visibility < 0.5
  mrm:
    TARGET: MRM
    RULE:
        WHEN AND:
            egoADS.active: true
            nextAreaWithinMRM: true
            nextArea: restrictedArea
```

Listing 4. Example of Keyword RESTRICTION.

IV. Implementation

The system has two central parts. First, the parsing of an ODD and Ontology in an efficient data structure, and secondly, the inference of the ODD with the ability to update the OD during the runtime. It was written in C++17 to achieve high performance and benefit from object-oriented features.

The first step is the parsing of the ontology, to check for the correctness of the ODD later on. The Redland libraries[5] are used to read the ontology file into a list of triplets, whereby the individual elements of the triplets are strings. The list is processed twice: first to create the ontology classes, and then to add the data properties. There is a simple separate data structure for this, called *Ontology*, which is shown in Figure 3. The *Ontology* essentially consists of two maps that store the classes and data properties. Each class always stores the keys of its top-classes and sub-classes and each data property stores the data type and the range, i.e. the classes that have the data property.

After parsing the ontology, the system reads the ODD. To do this, the YAML file is parsed with yaml-cpp[6] and can then be transferred block by block to the ODD-Engine's data structure, whereby each block is bound to a keyword. Keywords can be used multiple times in the ODD and can be defined in any order in the ODD. However, rules, ODD variables and objects must be defined before use. Otherwise, the system throws an exception if incorrect ODDs are used. All of these values are saved in the *Database*. This structure also stores the return values of rules and all related data types.

[5] https://librdf.org/
[6] https://github.com/jbeder/yaml-cpp

Fig. 3. UML-diagram of the structure *Ontology*

All of these values are saved in the *Database* data structure. This structure also stores the return values of rules and all related data types. Every ontology-related values such as objects or data types of data properties are compared with the ontology in the database to ensure that they are compliant.

In addition to the database, there is a structure called *ExpressionContainer*. It holds all the expressions and links their variables to data properties and ODD-variables from the database. The return value of an expression will be stored in the database. Before inference of the rules, the *ExpressionContainer* calculates all expressions and stores their result again in the *Database*.

All rules are stored in a separate structure shown in Figure 4. Each rule stores all its associated logic blocks, with every block containing the necessary conditions for evaluation. Conditions are separate structures which store their target values and a key to the *Database* entry, with the actual value. As Fig. 5 suggests, rules that reference each other can be viewed as a set of tree-like structures, where a single rule may depend on several others, and multiple rules may, in turn, depend on it. The so-called RuleMap stores the rules grouped according to the level of their dependencies. For inference, the RuleMap can be processed sequentially, from the lowest level with rules without dependencies to the highest level. The processing order in a particular level is irrelevant, as the rules in a level are not dependent on each other. The structure is optimized for fast inference during the runtime. Rules are treated as restrictions when their flag *isRestriction* is satisfied. If so, no other rule is allowed to depend on this restriction. In the event that an attempt is made to bind a rule to a restriction, the system throws an exception. The string target holds the restriction key, which will be returned if the restriction is true during the inference.

The database can now be updated with the current OD during runtime. There are two methods for this: set_sub_value(object, subclass) and set_data_property(dataproperty, value). The first checks the correctness of the ontology value when it is called, while the second checks the data type. If an error occurs, the program is stopped. Both methods are therefore only dynamically type-safe. When the entire OD has been updated, the inference of the entire ODD-Engine can be initiated with

Fig. 4. UML-Diagram of the rules. Every rule consists of one or more *LogicBlocks*, these in turn consist of one or more conditions. *Condition* is a abstract class and can either be a *BoolCondition* or a *ObjectCondition*.

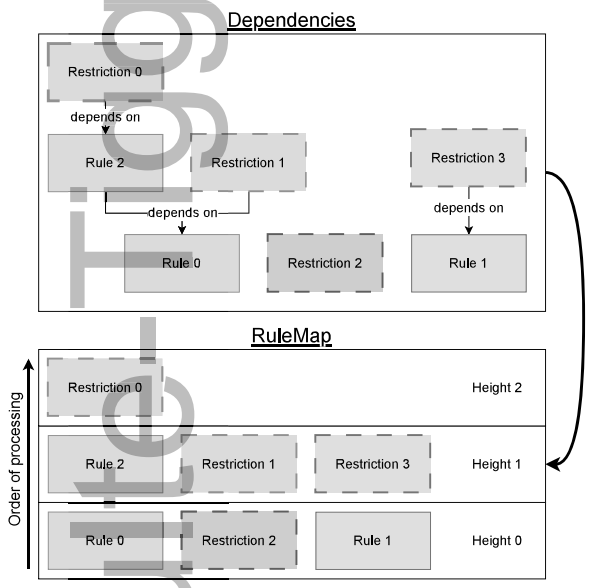

Fig. 5. The dependencies of the rules can be seen as a forrest of tree-like structure. For a efficient processing of the rules during the inference, the rules are stored in the *RuleMap*. The height stands for the height of dependency of the rules. The *RuleMap* will be processed from height 0 upwards.

the *inference* method. Fig. 6 shows the procedure for this. The *ExpressionContainer* is first updated, then calculates all expressions and writes the results to the *Database*. These values are then used in the *RuleMap* to verify the rules and their conditions. All restriction keys that apply are returned after processing.

This system allows building the entire data structure in the initialization phase and inferring it only during the execution phase. As the structure is no longer growing, every verification of a condition is constant in its time complexity. It leads to a time-efficient execution of each inference step, which can be called numerous times per second.

V. Validation results

For validation, first the performance of the ODD-Engine was rated in terms of execution time and second, an exemplary integration into an automated driving system was realized.

Figure 7 shows the results of the performance measurements of the ODD-Engine. An example ODD with an example ontology was created, the example ODD and ontology

Fig. 6. Shows the sequence of an inference iteration of the ODD-Engine.

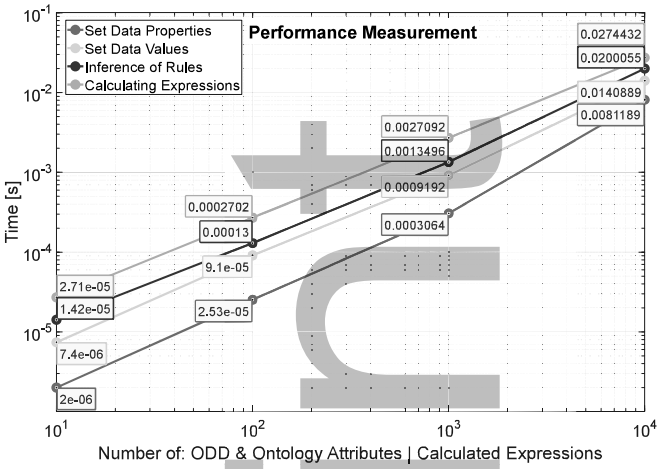

Fig. 7. Performance measurements in logarithmic scale the X axis indicates the scaling of attributes in ODD and ontology or the number of expressions. Yellow time of calculating x expressions; Blue time of inference of x rules; Green time to set x data values; Red time to set x data properties

was scaled up to have an amount of $x = [10; 10^2, 10^3; 10^4]$ data properties, values, rules and expressions. The observation was made that all parts of the ODD-Engine scale linearly. Even though an ODD scaled dramatically to 10^4 elements, an iteration adds up to about only 0.07 seconds, and therefore still holding up to a demanded 10Hz calling rate. The execution time for parsing the ontology and the ODD definition can be neglected because it is only performed once during initialization.

For further evaluation, the ODD-Engine was integrated into the ADS architecture proposed in [30] for real-world applications in different projects. With the integrated version, we conducted validation tests to show the ODD-Engine in action. A simulation of a test track, that we use frequently with our real automated vehicle was used, and a "Restricted Area" was defined as to see in figure 8. The vehicle drives towards the restricted area at 50km/h, meanwhile all hard- and software modules in the ADS populate the ontology attributes for an abstracted representation of the OD. During operation, the ODD-Engine's inference described in figure 6

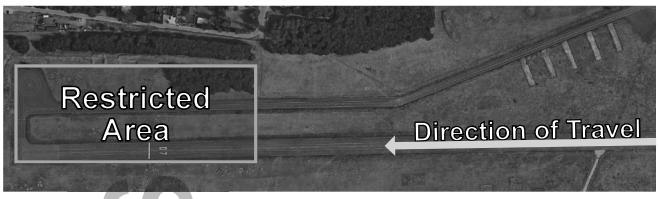

Fig. 8. Satellite picture of the test track used in simulation, with a marking for a "Restricted Area".

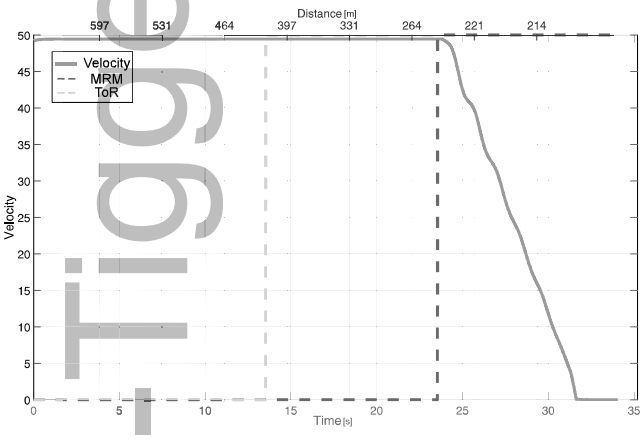

Fig. 9. Velocity profile of the automated vehicle against time as well as distance to the restricted area. The green line indicates the trigger of a ToR by the ODD-Engine while the red line indicates the start of a MRM.

is called by the ADS with a frequency of 10Hz. In the example ODD, a restriction was defined which targets a ToR (Take-over-Request to the driver), if the vehicle is about to drive into a restricted area. If the driver does not overtake the control and the ADS remains active, a MRM (Minimal-Risk-Maneuver of the vehicle) is triggered, by which the vehicle automatically changes to the emergency lane and reduces the speed to standstill. Figure 9 shows the velocity of the vehicle and the green line shows when the ODD-Engine triggers a ToR, while the red line indicates the point in time when the target changes to a MRM. When the ODD-Engine gives back a MRM as target, the ADS performs the MRM and the velocity slows down to full stop. Figure 10 shows the vehicle driving on the virtual test track (white line). When the target changes to a ToR the line is colored green and changes to red when the MRM is performed. A frequency of 10Hz for the ODD-Engine iterations is considered to be safe. The upper presented performance analysis has shown that the ODD-Engine is capable to stay within the given time limit of 100ms, the reciprocal value of 10Hz.

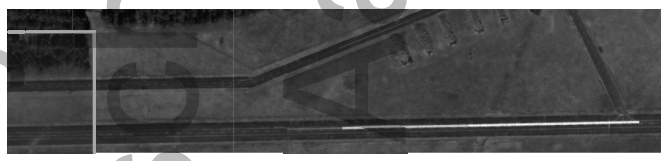

Fig. 10. Satellite picture of the automated vehicle's position, indicating in green at which position a ToR was triggered by the ODD-Engine and in red when a MRM is executed.

VI. Conclusion and Outlook

In this paper, we presented our successful implementation of an ODD Definition Language within our ODD-Engine for an ongoing checking of the OD during runtime. This includes also the behavioral reaction and the degradation of the automation system, which could be specified via our newly introduced RESTRICTIONS. Therefore greatly improving the expressivenessby having the option to choose a target rather than just to determine to be inside or outside its ODD. We explained our implementation in detail and the complete code is publicly available as OpenSource on GitHub[7]. In the evaluation part, we could show that our implementation is able to manage a huge number of ODD and ontology attributes with the corresponding expressions within a short inference time. Furthermore, we presentedfirst experimental results of the integration into a real automated driving system of a passengercar.

The benefit of our implementation is that it can be applied to any kind of vehicle and any kind of ODD just by creating own ontologies and ODD definitions. In our future work, we will do further validation and we will apply the ODD-Engine to automated vehicles in the different domains road traffic and mining. But due to the open-sourcerepository, we also hope to motivate other research teams to use and extend our ODD-Engine.

References

[1] SAE, *Taxonomyand Definitions for Terms Related to Driving Automation Systemsfor On-Road Motor Vehicles J3016_202104*, Std.

[2] SAE, *Taxonomy& Definitions for Operational Design Domain (ODD) for Driving Automation SystemsJ3259*, Std.

[3] C. Sun, Z. Deng, W. Chu, S. Li, and D. Cao, "Acclimatizing the operational design domain for autonomous driving systems," *IEEE Intelligent Transportation SystemsMagazine*, vol. 14, no. 2, pp. 10–24, 2022.

[4] P. Koopman and F. Fratrik, "How many operational design domains, objects, and events?" in *SafeAI@AAAI*, ser. CEUR Workshop Proceedings, vol. 2301. CEUR-WS.org, 2019. [Online]. Available: http://dblp.uni-trier.de/db/conf/aaai/safeai2019.html#KoopmanF19

[5] *Road Vehicles - Test Scenarios for Automated Driving Systems. Specification for Operational Design Domains*, ISO Standard ISO 34 503:2023.

[6] BSI, *Operational Design Domain (ODD) Taxonomy for an Automated Driving System(ADS). Specification*, ISO Std. PAS 1883:2020.

[7] ASAM. ASAM OpenXOntology. Abgerufen am 23.07.2024.[Online]. Available: https://www.asam.net/standards/asam-openxontology/

[8] L. Mendiboure, M. L. Benzagouta, D. Gruyer, T. Sylla, M. Adedjouma, and A. Hedhli, "Operational Design Domain for Automated Driving Systems: Taxonomy Definition and Application," in *2023 IEEE Intelligent Vehicles Symposium(IV)*. IEEE, pp. 1–6.

[9] J. Erz, B. Schutt, T. Braun, H. Guissouma, and E. Sax, "Towards an Ontology That Reconciles the Operational Design Domain, Scenario-based Testing, and Automated Vehicle Architectures," in *2022 IEEE International SystemsConference (SysCon)*. IEEE, pp. 1–8.

[10] AVSC, "AVSC best practice for describing an operational design domain: Conceptual framework and lexicon," *SAE Industry Technologies Consortia*, 2020.

[11] ASAM, "OpenODD." [Online]. Available: https://www.asam.net/standards/detail/openodd/

[12] T. Charmet, V. Cherfaoui, J. Ibanez-Guzman, and A. Armand, "Overview of the operational design domain monitoring for safe intelligent vehicle navigation," in *2023IEEE 26th International Conference on Intelligent Transportation Systems(ITSC)*, pp. 5363–5370.

[13] P. Irvine, X. Zhang, S. Khastgir, E. Schwalb, and P. Jennings, "A two-level abstraction ODD definition language: Part I," in *2021 IEEE International Conference on Systems,Man, and Cybernetics (SMC)*, pp. 2614–2621.

[14] E. Schwalb, P. Irvine, X. Zhang, S. Khastgir, and P. Jennings, "A two-level abstraction odd definition language: Part ii," in *2021 IEEE International Conference on Systems,Man, and Cybernetics (SMC)*. IEEE, pp. 1669–1676.

[15] D. Rohne, A. Richter, and E. Schwalb, "Implementing ODD as single point of knowledge to support the development of automated driving," in *2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, pp. 1364–1370.

[16] G. Bruce, A. B. Da Costa, S. Khastgir, and P. Jennings, "Towards Robust ISO 34503 ODD Language Syntax and Semantics," in *2024 IEEE 27th International Conference on Intelligent Transportation Systems(ITSC)*. IEEE, pp. 4124–4129.

[17] A. Aniculaesei, C. Schindler, C. Knieke, A. Rausch, D. Rohne, and A. Richter, "A Method for ODD Specification and Verification with Application for Industrial Automated Driving Systems," in *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, pp. 1519–1526.

[18] E. Thorn, S. C. Kimmel, M. Chaka *et al.*, "A framework for automated driving system testable cases and scenarios." [Online]. Available: https://rosap.ntl.bts.gov/view/dot/38824/dot_38824_DS1.pdf

[19] ISO, *Road Vehicles – Functional Safety*, Standard ISO 26 262-1:2018.

[20] ISO, *Road Vehicles — Safety of the Intended Functionality*, Std. ISO 21 448:2022.

[21] C. W. Lee, N. Nayeer, D. E. Garcia, A. Agrawal, and B. Liu, "Identifying the Operational Design Domain for an Automated Driving System through Assessed Risk," in *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, pp. 1317–1322.

[22] J. Schulte-Tigges, M. Rondinone, M. Reke, J. Wachenfeld, and D. Kaszner, "Using v2x communications for smart odd management of highly automated vehicles," in *2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC)*, 2023, pp. 3317–3322.

[23] M. Gyllenhammar, R. Johansson, F. Warg, D. Chen, H.-M. Heyn, M. Sanfridson, J. Söderberg, A. Thorsén, and S. Ursing, "Towards an operational design domain that supports the safety argumentation of an automated driving system," in *10th European Congress on EmbeddedReal Time Systems(ERTS 2020)*. [Online]. Available: https://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Akth%3Adiva-267132

[24] X. Zhang, J. Tao, K. Tan, M. Törngren, J. M. G. Sánchez, M. R. Ramli, X. Tao, M. Gyllenhammar, F. Wotawa, N. Mohan, M. Nica, and H. Felbinger, "Finding Critical Scenarios for Automated Driving Systems: A Systematic Mapping Study," vol. 49, no. 3, pp. 991–1026.

[25] A. Shakeri, "Formalization of Operational Domain and Operational Design Domain for Automated Vehicles," in *2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. IEEE, pp. 990–997.

[26] H. Torfah, C. Xie, S. Junges, M. Vazquez-Chanlatte, and S. A. Seshia, "Learning monitorable operational design domains for assured autonomy," in *International Symposiumon Automated Technology for Verification and Analysis*. Springer, pp. 3–22.

[27] I. Colwell, B. Phan, S. Saleem, R. Salay, and K. Czarnecki, "An Automated Vehicle Safety Concept Based on Runtime Restriction of the Operational Design Domain," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, pp. 1910–1917.

[28] Sun, Chen, "Operational design domain monitoring and augmentation for autonomous driving." [Online]. Available: http://hdl.handle.net/10012/18964

[29] T. Charmet, V. Cherfaoui, J. Ibanez-Guzman, and A. Armand, "Operational design domain monitoring with uncertain measurements,"in *2024 IEEE 27th International Conference on Intelligent Transportation Systems(ITSC)*, 2024, pp. 1023–1030.

[30] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Matheis, "A Self-Driving Car Architecture in ROS2," in *2020 International SAUPEC/RobMech/PRASAConference*, 2020, pp. 1–6.

[7] https://github.com/MASKOR/ODD_Engine