The SGSM Framework: Enabling the Specification and Monitor Synthesis of Safe Driving Properties through Scene Graphs

Trey Woodlief[a,*], Felipe Toledo[a], Sebastian Elbaum[a], Matthew B. Dwyer[a]

[a] *University of Virginia, 85 Engineer's Way P.O. Box 400740, Charlottesville, 22904, VA, USA*

Abstract

As autonomous vehicles (AVs) become mainstream, assuring that they operate in accordance with safe driving properties becomes paramount. The ability to specify and monitor driving properties is at the center of such assurance. Yet, the mismatch between the semantic space over which typical driving properties are asserted (e.g., vehicles, pedestrians) and the sensed inputs of AVs (e.g., images, point clouds) poses a significant assurance gap. Related efforts bypass this gap by either assuming that data at the right semantic level is available, or they develop bespoke methods for capturing such data. Our recent Scene Graph Safety Monitoring (SGSM) framework addresses this challenge by extracting scene graphs (SGs) from sensor inputs to capture the entities related to the AV, specifying driving properties using a domain-specific language that enables building propositions over those graphs and composing them through temporal logic, and synthesizing monitors to detect property violations. Through this paper we further explain, formalize, analyze, and extend the SGSM framework, producing SGSM++. This extension is significant in that it incorporates the ability for the framework to encode the semantics of *resetting* a property violation, enabling the framework to count the quantity and duration of violations.

*Corresponding Author

Email addresses: `adw8dm@virginia.edu` (Trey Woodlief), `ft8bn@virginia.edu` (Felipe Toledo), `selbaum@virginia.edu` (Sebastian Elbaum), `matthewbdwyer@virginia.edu` (Matthew B. Dwyer)

We implemented SGSM++ to monitor for violations of 9 properties of 3 AVs from the CARLA Autonomous Driving Leaderboard, confirming the viability of the framework, which found that the AVs violated 71% of properties during at least one test including almost 1400 unique violations over 30 total test executions, with violations lasting up to 9.25 minutes. Artifact available at `https://github.com/less-lab-uva/ExtendingSGSM`.

Keywords: runtime verification, autonomous vehicles, safe driving properties, scene graphs

1. Introduction

Autonomous vehicles (AVs) are quickly approaching wide-spread public-road deployment, with several companies already leveraging fleets of AV taxis in multiple US cities [1, 2]. However, deployments of full AV systems have led to multiple human and animal fatalities [3, 4, 5, 6, 7]. While some analysis from the companies deploying AVs suggests that they are involved in fewer collisions that pose risk of injury compared to human drivers [8, 9], we continue to see AVs violate required driving behavior with grave consequences.

Ideally, AVs would be deployed without latent faults due to extensive validation and verification [10, 11]. However, the inherent complexities of these systems and the long-tail of potential scenarios make it infeasible to provide complete and strong guarantees [12, 13]. These limitations have motivated the use of runtime monitors that can evaluate compliance of safety specifications during deployment [12, 13, 14, 15, 16]. However, current monitoring mechanisms are inadequate for checking driving behavior as they cannot account for the spatiotemporal distribution of entities (e.g., other vehicles, pedestrians, traffic signals) that may influence the AV driving behavior, and which can only be obtained from complex multi-dimensional sensors like camera and LiDAR. Alternatively, approaches that do account for driving behaviors do it through bespoke, handcrafted translation between the monitor's input, e.g. sensor input, or internal system state, and the semantics of the safety specifications, limiting generalizability (as per related work in Section 2).

A key challenge with developing monitors for the driving behavior of AVs is the mismatch between the semantic space over which typical road properties are asserted (e.g., cars, stop lights, intersections) and the input space of AVs which are typically in the form of sensed data (e.g., images,

LTL$_f$ Formula for ψ_9:
$\mathcal{G}((\neg\, hasStop \wedge \mathcal{X}\, hasStop) \rightarrow (\mathcal{X}(hasStop\, \mathcal{U}\, (isStopped \vee \mathcal{G}(hasStop)))))$

Atomic Propositions:
hasStop: $|\, relSet(Ego, \text{isIn}) \cap relSet(stopLine, \text{controlsTrafficOf})\, | > 0$
isStopped: $|\, filterByAttr(Ego, \text{speed}, \lambda\, \text{x: x} < \varepsilon)\, | = 1$

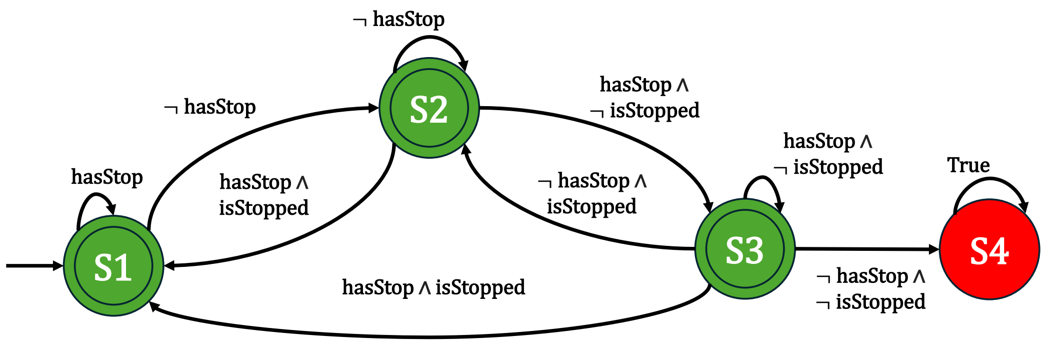


Figure 1: LTL$_f$ for safe driving property, Atomic Propositions over the image sensor data, and DFA for property ψ_9. Adapted from [18] for consistent notation.

radar, point clouds). As a motivating example, consider the following rule (ψ_9 in Table 1) from the Virginia Driving Code § 46.2-821 *"The driver of a vehicle approaching an intersection on a highway controlled by a stop sign shall, immediately before entering such intersection, stop at a clearly marked stop line [...]"* [17]. Evaluating this property requires extracting information about road lanes, stopping signals, e.g. stop signs, painted markers, etc., which lanes the signals affect, and if the vehicle occupies those lanes.

To address these limitations we proposed utilizing scene graphs (SGs) to produce a framework for SG Safety Monitoring (SGSM) that enables the specification of driving properties for AVs and their automated synthesis as part of a system monitor. The approach builds on two key domain-specific components: 1) a spatial scene graph generator (SGG) that can extract rich scene representations from sensor inputs for the AV domain into SGs that abstract the entities related to the AV, and 2) a domain-specific language (DSL) that enables a developer to define programmable queries over the SG and compose the output of those queries as part of discrete metric temporal logic properties that can be monitored at runtime. Together, the SG and DSL offer a rich space to express common road properties relevant to AVs that can be automatically encoded as a runtime monitor.

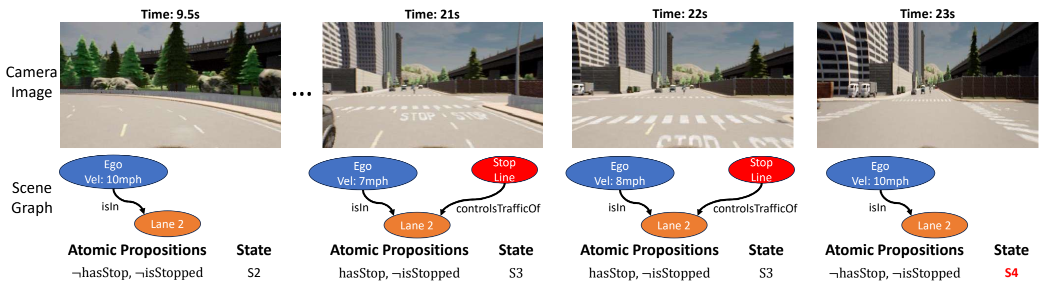


Figure 2: AV (TCP) running an intersection without stopping from [18]. Top: AV Camera Images. Middle: sub-SG for checking safety property. Bottom: Atomic Propositions evaluated from SG and updated state of the DFA shown in Fig. 1 leading to violation.

Our initial work in this area identified SGs as a useful abstraction for measuring test coverage for AVs at the semantic level [19]. That motivated our recent work introducing SGSM [18] to leverage the semantic abstraction of SGs for safety monitoring. In this work, we extend and further elaborate SGSM, producing SGSM++. In this paper, we 1) provide additional detail and discussion on related works (Section 2) and SGSM (Section 3), 2) extend the framework to SGSM++ to enable tracking multiple violations including the number and duration of individual violations (Section 4), 3) provide a formal analysis of the expressiveness of the framework (Section 5), and 4) expand the experimental study to analyze the new contributions (Section 6).

Returning to the motivating example of a stop sign, Fig. 1 shows the safety specification described in linear temporal logic over finite traces (LTL$_f$) [20] (further discussed in Section 2.5), the atomic propositions (APs) expressed in our DSL, and the deterministic finite automaton (DFA) automatically synthesized from the LTL$_f$ formula. Fig. 2 shows a snippet of a time sequence in which an AV passes an intersection controlled by a stop line without stopping. The top row shows the AV's camera input, while the second row shows the subgraph of the SG relevant to the property extracted from each input image. As the AV approaches the intersection, the stop line appears in the graph with the relationship that it "controlsTrafficOf" the lane that the ego vehicle (ego from now on) is in; yet, ego's velocity remains consistent. The APs and states shown at the bottom correspond to the transitions and state of the DFA at that time; note that as the input sequence progresses, and the stop line is included in the SG but ignored by the AV, the DFA moves toward and finally enters the failure state, S4, indicating a violation.

We introduce the first domain-cognizant, general, and extend-

able approach for the specification of AV safety driving properties that can be encoded for automatic monitoring during runtime. The approach is domain-cognizant in that it bridges the gap from raw sensor data to primitive propositions that capture domain concepts. It is general in that it is independent of the AV implementation, only requiring access to external inputs and outputs, e.g. sensor data and AV control commands, from which SGs can be derived. It is extendable in that the DSL building blocks can be combined to encode properties beyond the ones we study. We implemented the approach in CARLA [21] to explore its capabilities in simulation for 3 AVs from the CARLA leaderboard competition. We find that these AV systems, though highly performant under the existing competition metrics, consistently violate driving rules—in 50% of executions the AV crossed into opposing traffic (ψ_1) and in 73% of executions the AV ignored a stop sign (ψ_9). Further investigation using SGSM++ identified a combined 12.5s spent in the opposing lane and 31 ignored stop signs in total.

2. Background and Related Work

We briefly survey related work in this area, including prior work on AV safety monitors and ontologies for the AV domain, and work that is foundational to our approach, including SGGs to extract scene semantics, formulations of propositions over graphs, and temporal logic to specify sequences of proposition values.

2.1. AV Safety Monitors

Prior work has examined monitoring end-to-end systems. Desai et al. propose using observable trajectories to monitor path following and safety buffers using signal temporal logic (STL) [22]. Similarly, Zapridou et al. also use STL to describe properties and check them using the CARLA simulator [23]. Stamenkovich et al. use system-independent runtime monitors that observe only the external inputs and outputs to check properties specified in LTL [13]. Castelino et al. propose using vehicle communication systems (V2X) to improve the robustness of runtime monitoring by the diversity of available data [24]. Morse et al. characterize spatial relationships between sensed objects and robot behaviors, by using graph representations and First Order Logic (FOL), that can be used for runtime monitoring [25]. Similarly, Matos Pedro et al. developed a runtime verification technique agnostic to the target system for checking spatio-temporal properties using LTL and Modal

Metric Spaces [26]. Yalcinkaya et al. propose a runtime assurance framework for programming AVs that emits a runtime monitor for the programmed behaviors [27]. Work in shielded reinforcement learning aims to learn [28] or enforce [29] safety properties for agents specified in temporal logic and has shown to increase robustness of learned behaviors.

However, each of the previous techniques assume there is a mapping from the sensed inputs to the semantics of the atomic propositions (APs) in their properties. Extracting the APs requires either limiting the propositions checked to those already consumed by the system or additional effort to extract the relevant semantics, both of which limit its generalizability. In our work we leverage Scene Graph Generators (SGGs), discussed in Section 2.3, to generate the sensor input abstraction which we then process using our domain-specific language to extract AP values, broadening the applicability of our techniques to any system. Further, SGG is a burgeoning field of study within machine learning and computer vision that is constantly advancing, with recent progress on SGGs showing improvements on research benchmarks [30, 31]. This work provides the framework to benefit from these improvements as we expect that the near future will bring faster, more accurate, more broadly applied methods for SG generation which will further broaden the applicability of our safety monitoring approach.

Prior work has also focused on building specialized monitors for AV software subcomponents rather than full end-to-end systems, such as trajectory prediction [32], collision avoidance [33, 34], lane changing and overtaking [34, 35], or interfaces between AV components such as the CAN bus [14, 36] or through ROS topics [37]. Additionally, most of these efforts include propositions over simple types, e.g., "disengaged cruise control", or "traveled for 2 seconds".

The introduction of machine-learned components to process multi dimensional sensors complicates the design of monitors due to the black-box nature of such components and their ability to perform what were previously separate subtasks as end-to-end computations. Kochanthara et al. provide a robust characterization of component-level safety properties for the Apollo AV, but do not explore methods for evaluating these properties at runtime [38]. Torfah et al. use counter-example guided learning to learn a runtime monitor that can predict from an observed state if the AV is going to leave its safe operation domain [39]. While this can learn to monitor for violations from examples, it cannot be used to encode a specific desired property a priori. Yang et al. use a reachability analysis tool for runtime safety verification of a neural network navigation control system using LiDAR to avoid collisions [40]; however, this is not generalizable to higher-order properties due to its narrow focus on control dynamics. Grieser et al. provide a mechanism for monitoring a limited set of safety properties based on LiDAR point clouds [41]. However, it is not generalizable to other sensors or properties because it is built around a DNN particularly tailored for this application that only takes in LiDAR points and outputs a torque value to control the motor and a steering angle command; different sensors or properties would require further bespoke configurations. Anderson et al. try to overcome this issue by introducing Spatial Regular Expressions for pattern matching over perception streams containing spatial and temporal data, leveraging object detection networks [42]; similarly, Balakrishnan et al. introduce PerceMon to monitor detection systems using specifications defined in TQTL [43]. Nonetheless, they can only reason about relationships given by bounding box overlap, and misse richer types of relationships like proximity between entities or traffic semantics. Similarly, Grundt et al. use STL to formally encode specifications, capturing physical attributes about the ego vehicle and the ego vehicle's relationships to other vehicles, e.g. the angle of the ego vehicle or its distance to another vehicle. However, the formalism cannot capture semantic relationships nor relationships between other entities, fundamentally limiting the specifications that can be encoded [44].

2.2. Ontologies for the AV Domain

Another line of related research has explored different ontologies in the AV domain [45] for scenario-based testing [46, 47] and for situation assessment and decision making [48, 49, 50]. The main limitation of these approaches, however, is that the ontologies are completely tied to the SUT, thus only encoding the information needed by the system and making them nongeneralizable. Our previous work on SGs for AV testing [19] demonstrated the utility of SGs as a basis for measuring coverage of nontemporal properties, but does not provide a mechanism to express and automatically check the rich properties studied here. Closer to our abstraction, Majzik et al. envisioned using a graph-based ontology of the environment with STL to monitor system performance [51], but defines no properties for self-driving cars. Our work extends and formalizes this notion with: a spatial-relation graph that can be computed from external, system-independent inputs, a graph-semantics logic DSL and LTL$_f$ that can specify safety-critical properties; and we demonstrate that this approach can automatically find property violations at runtime for AV driving systems.

2.3. Scene Graph Generation (SGG)

SGG is an emerging area of research focused on extracting relationships between objects from sensor data, e.g., from an image input inferring a pedestrian is on a crosswalk. SGs are directed graphs [52], with a vertex set V that represents the set of entities captured by a sensor, e.g., camera or LiDAR, and a set of directed edges $(u, v) \in E$ describing their relationships. More formally, an SG,

$$\begin{aligned} G = (&V, E : V \mapsto V, \mathit{Ego} \in V, \\ &\mathit{kind} : V \mapsto K, \mathit{rel} : E \mapsto R, \mathit{att} : V \cup E \mapsto M) \end{aligned}$$

has a distinguished *Ego* vertex and functions to access the entity *kind* of a vertex, the *rel*ation encoded by an edge, and *att*ribute values of vertices and edges. A map M is used to associate attribute values with each type of attribute.

SGGs are highly configurable, enabling the SGs to be tailored to different domains. SGGs can be configured to work with different parameterizations based on the kinds of entities (K); e.g., whether cars and trucks should be treated as separate classes, consolidated into a "vehicle" class, or ignored completely. The types of relationships (R) captured and their semantics can be configured based on the goals of the SGG; e.g., whether to include distance information such as entities being "near" and "far" from each other or higher-order information such as this lane "opposes" the other lane based on traffic rules. The SGG parameterization extends to the attribute information (M), e.g., whether to capture the color of the traffic light, the speed of the other vehicle, or the height of the pedestrian. In recent years, many SGG techniques have been developed [53] leveraging object detection systems (e.g. [54, 55]) to detect different entities, and then extract relationships between them. In the realm of AVs, more tailored SGGs have been proposed [56, 57, 58], that leverage domain-specific semantics like road types, vehicle types, and static or dynamic entities. Our framework uses an SGG to extract graph-based abstractions of sensor data from the world, and our study builds on an SGG that operates in the CARLA AV simulator [21].

2.4. Graph Properties

There is a rich literature on methods for specifying properties of graphs. Given the relational nature of graphs, properties could be specified as queries in relational algebra [59] or in more specialized graph query languages built on relational algebra primitives [60, 61]. Using such methods one can formulate a wide range of property specifications. For example, one can express that "a graph contains a stop signal that controls the lane ego is in" by combining primitives relational *join* and set *intersection* as follows:

$$join(Ego, \texttt{isIn}) \cap join(\texttt{stopsignal}, \texttt{controls}) \neq \emptyset$$

where $\texttt{stopsignal} = \{v : v \in V \wedge kind(v) = \text{stop signal}\}$.

In this work, we focus on core primitives that can be used to specify properties like the one described above. In addition to standard set operations, those primitives include join (relSet) and a primitive that allows selecting a subset of vertices based on properties of their attributes (filterByAttr), further discussed in Section 3.1.1. More complicated properties can be expressed over paths by composition and iteration using these primitives. Executable specifications built in this way are appropriate for runtime monitoring, in contrast to more declarative approaches [61].

2.5. Linear Temporal Logic

Linear Temporal Logic (LTL) is a formal language that has been widely used for modeling and analyzing systems with temporal aspects, including embedded and cyber-physical systems [62, 63, 64]. An LTL formula ψ, is satisfied by an infinite sequence of truth valuations of APs [65]. There are **logic operators**: *And* ($\wedge$), *Or* ($\vee$), *Not* ($\neg$), etc., and **temporal operators**: *Next* ($\mathcal{X}$), *Until* ($\mathcal{U}$), *Always* ($\mathcal{G}$), *Eventually* ($\mathcal{F}$). By leveraging these operators, LTL allows for the precise specification of the system's behavior over time. For runtime monitoring, we use LTL over discrete, finite traces (LTL$_f$) [20]. An LTL$_f$ formula can be automatically converted to a DFA that validates whether a finite trace satisfies the property [66, 67] as shown in Fig. 1. Each DFA has a defined *start state* based on the LTL$_f$ formula which will be used at the initial state when evaluating the formula. A given DFA may also have a *trap state*, a state whose only transitions are to itself such that regardless of the future AP values the DFA will remain in the trap state. LTL$_f$ does not provide a language for specifying the APs themselves; rather, the APs must be evaluated before being consumed by the LTL$_f$ formula.

3. SGSM Framework

Fig. 3 provides an overview of the monitoring framework which has two phases to enable the specification and runtime monitoring of driving proper-

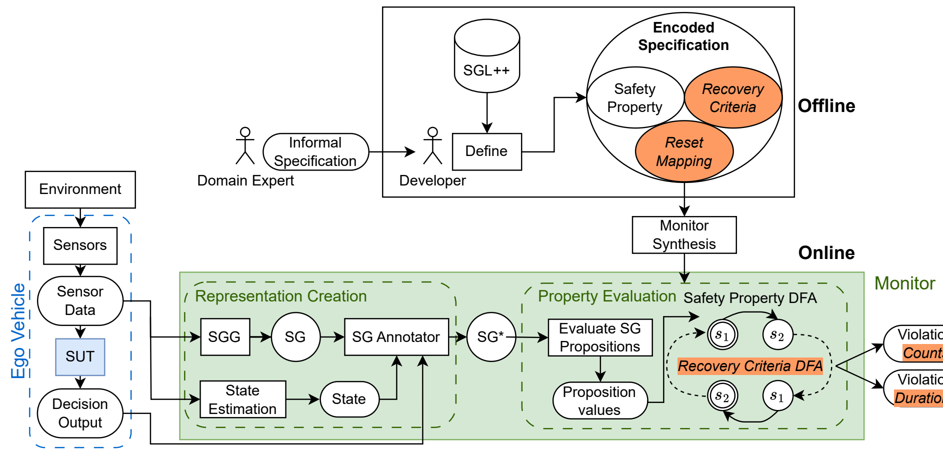


Figure 3: SGSM++ framework overview. SGSM++ offline phase at the top produces an encoded specification which is then synthesized into a monitor used in the online phase at the bottom to identify violations. Items in *orange italics* are part of the extension to SGSM++ (discussed in Section 4) from SGSM [18] (discussed in Section 3).

ties for AVs. First, in the offline phase shown at the top, a developer works to encode properties to be checked by the system. The encoded specification is then synthesized to produce a monitor which will evaluate each property during the online phase to monitor for property violations. In the online phase, the monitor leverages the sensor input and decision output of ego to create an enriched scene graph, SG*, which is then used to evaluate the APs over SG* to determine if a violation has occurred. Fig. 3 encompasses both SGSM, developed in our initial work [18], and further extensions explored in this work to create SGSM++. In Fig. 3, the novel components for SGSM++ are shown in orange italics for clarity. The following sections provide further discussion on the SGSM framework; Section 4 then builds from this as the basis for the SGSM++ framework.

3.1. Offline

SGSM assists developers in formally specifying driving properties prepared by domain experts, such as those found in a driving manual, in a computable format that is checkable at runtime. The assistance comes in the form of a DSL, called Scene Graph Language (SGL), that allows the developer to define LTL$_f$ formulas and evaluate propositions on the SG and ego's state. The developer begins by encoding the specification as a safety property in SGL, and defines the sampling rate at which the property needs to be checked. The safety property can then be synthesized into a system monitor that can be deployed during the online phase to check for property violations.

We now introduce SGL, which combines LTL$_f$ and a set of SG querying functions, to facilitate the specification of driving properties.

3.1.1. Specification Definition

SGL allows the developer to reason about ego's environment and state through propositions over sets of nodes in the SG. In addition to traditional set operations including union, intersection, and difference, SGL leverages two functions to perform graph queries: **relSet** and **filterByAttr**. Together, these functions will be used to derive sets of nodes with specific semantics, e.g., the set of red traffic lights. These sets will then allow for evaluating the atomic propositions by evaluating the elements of the set, e.g., whether or not there is a red traffic light is found by checking if the set of red traffic lights is non-empty.

The **relSet** function computes the join of a set of vertices and a relation.

$$\begin{aligned} \mathit{relSet} &: (V_1 \subseteq V, r \in R) \mapsto V_2 \subseteq V \\ V_2 &= \{v_2 : v_1 \in V_1 \wedge (v_1, v_2) \in E \wedge \mathit{rel}(v_1, v_2) = r\} \end{aligned}$$

For example, the set of lanes controlled by a stop sign is $\mathit{relSet}(\mathit{stopSigns}, \text{controlsTrafficOf})$. SGL also supports **relSetR** – the join of the transpose of the given relation. The transpose allows, e.g., finding the set of stop signs that control a lane.

The **filterByAttr** function selects a subset of vertices, V_1, whose attribute, m, satisfies a given predicate, f.

$$\begin{aligned} \mathit{filterByAttr} &: (V_1 \subseteq V, m \in M, f : T \mapsto \mathit{bool}) \mapsto V_2 \subseteq V \\ V_2 &= \{v : v \in V_1 \wedge \mathit{type}(\mathit{att}(v)[m]) = T \wedge f(\mathit{att}(v)[m])\} \end{aligned}$$

For example, $\mathit{filterByAttr}(\mathit{trafficLights}, \text{lightState}, \lambda x : x = \text{Red})$ yields the corresponding set of red traffic lights. See Table 2 in Section 6 for SGL encodings of the atomic propositions evaluated for the properties studied.

As defined here, SGL utilizes only a core set of primitive operators, *relSet* and *filterByAttr* as these, along with the set, logic, and temporal operators are sufficient for this application. This allows us to explore the benefit of the approach with minimal language engineering and future work could build higher-level languages that compile to these core primitives. The study examines in Section 6.2 the utility of this level of expression for its purpose as a runtime monitor.

SGL includes standard operators for numeric comparison, boolean logic, and set manipulation which are used to convert from vertex sets to APs. For example, whether ego has a throttle attribute below a given threshold, ϵ, is specified by $|\mathit{filterByAttr}(\mathit{Ego}, \text{throttle}, \lambda x : x < \epsilon)| = 1$. The APs are the building blocks for specifying different aspects of the AV's environment and behavior and can be combined with temporal operators through LTL$_f$ to express temporal relationships in the AV's behavior, enabling a precise characterization of its actions and responses in dynamic environments.

SGL also builds on Linear Temporal Logic on Finite Traces (LTL$_f$) [20] which provides a set of logical and temporal operators (described in Section 2.5) for describing whether a finite trace of atomic proposition values satisfies the given LTL$_f$ formula. For example, Fig. 1 shows the LTL$_f$ formula for ψ_9 which encodes that the AV must stop at stop signs. Informally, this formula states that always, once the AV detects a stop sign, it must detect the stop sign until it has stopped; i.e., if it stops detecting the stop sign without having stopped, then it has run the stop sign. Section 4.1.2 provides additional, precise discussion of ψ_9's LTL$_f$ encoding.

In addition to the standard temporal operators, SGL also defines a discrete metric operator, \$[N][AP], to ease the LTL$_f$ specification over repeated APs by unrolling the AP N times using the $\mathcal{X}$ operator. Under a set sampling rate, this can be used to specify, e.g., that an AP has a certain value for a certain duration. This is studied in Section 6 for properties that check that the ego vehicle completes an action within a certain time window. We also adopt the use of the `last` keyword from prior work as a shorthand for $\neg\mathcal{X}$ *True* which encodes intuitively that the current input must be the last input—all future inputs lead to the formula not accepting [20].

3.1.2. Monitor Synthesis

The foundation of all SGL specifications is the safety property encoding to enable the synthesis of the monitor. The property must be encoded as a safety constraint, i.e. a property that must be continuously satisfied during execution. The formula will be evaluated repeatedly at runtime by the monitor and thus it must be specified in such a manner so that all satisfactory sub-traces are also accepting. In the corresponding DFA for the LTL$_f$ formula this means there must be a unique non-accepting state and that state must also be a trap state. This is exemplified by the property ψ_9 shown in Fig. 1; examining the DFA, we see that there are three accepting states that correspond to the progression of the LTL$_f$ formula, and a unique failure state that is also the trap state.

The correctness of the synthesized monitor is predicated on maintaining a consistent state between the monitor and the system. While this warrants careful consideration in the construction of the whole encoding, particular care is required setting the start state during initialization to reflect the state of the system. This could be enforced through outside guarantees of the system being in a known state at the start of monitoring, or by the encoding assuming no particular initial state and using the APs during the first several time steps to identify a known state being reached.

3.2. Online

As ego senses the environment, it provides data to the system under test (SUT) to produce a decision output for the vehicle. SGSM provides a runtime monitor (green box) to check for property violations by processing sensor data and appending the ego's decision output to create SG*. It then evaluates the SGL function over SG* to assign the values of the APs. These AP values are then used to update the LTL$_f$ DFA state machine. Finally, the monitor outputs if the property holds or is violated based on the DFA state. The monitor has two main modules described next.

3.2.1. Representation creation

This module consumes sensor data to estimate the state of ego and to produce an SG through the SGG component. The resulting SG is enriched by the SG annotator component with information about the SUT's output, and the state of ego to produce SG*. For example, Fig. 2 shows the relevant subgraphs generated for evaluating ψ_9 to monitor for stop sign violations. As shown in the middle two time steps, the SG contains ego in lane 2 as well as a stop line that controls lane 2. This information can be readily computed from ego's external sensor input to perceive which lane it is in and the presence of a stop signal, perhaps in combination with available high-definition maps. Additionally, the ego node contains an attribute for its velocity which could be measured from its speedometer, etc. to check if ego is stopped. Though not used in this example, SG is also annotated with ego's decision output to produce SG*. We can imagine a related property to ψ_9 that instead stated that immediately after detecting a stop signal, ego must begin to decelerate—this could be monitored by checking that ego immediately output a sufficient brake command.

In this presentation, SGSM aims to monitor for property violations that occur in the world. As such, the creation of SG* is designed to be black-box with respect to the SUT to maximize the generality of the approach, only using externally observable sensor data, i.e. SUT inputs, and decisions, i.e. SUT outputs. However, SGSM could also be employed in a white-box fashion to monitor the internal components of the SUT. For example, the SG annotator could additionally enrich SG* to include information from the motion planning or control components of the SUT to allow for specifying safety properties over, e.g., the planned future trajectory of the AV. We leave the exploration of such properties for future work.

3.2.2. Property evaluation

This module takes in SG* and an SGL function containing the LTL$_f$ property. It first evaluates each AP by querying SG*, and then uses the AP values to update the DFA state. Depending on the DFA state, the monitor returns whether the property holds or is violated.

Table 3 gives a complete list of APs computed to evaluate the properties studied. These APs yield an understanding of the spatial and temporal distribution of entities related to ego. For example, ψ_9 checks if ego responds to stop signs by evaluating the *hasStop* and *isStopped* APs. *hasStop* is true iff the set of lanes controlled by stop signs and lines (*stopSignLanes*) intersect with the set of lanes ego is in (*egoLanes*) is non-empty, which would indicate that ego is being directed to stop. *isStopped* is true iff the set of ego with speed $< \epsilon$ (*egoStopped*) is non-empty, indicating that ego is stopped.

4. Extending SGSM

We will first introduce the challenges to SGSM and then introduce two extensions to address them.

4.1. Motivation

We now explore the limitations of SGSM as presented across two dimensions. First, the properties as encoded in SGL differ from those encountered in, e.g., the driving code in one crucial dimension: the ability to handle multiple violations, i.e. to count the number and duration of different violations.

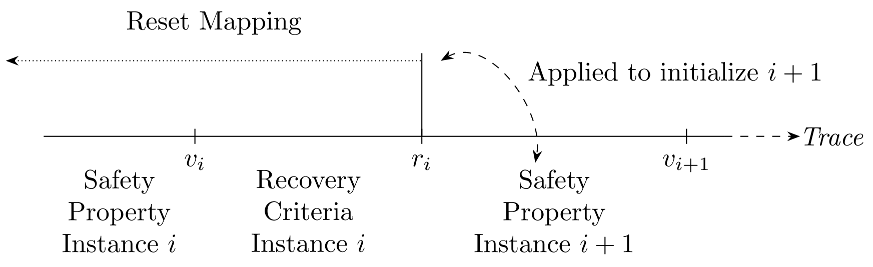

Reset Mapping
Applied to initialize $i+1$
Trace
v_i
r_i
v_{i+1}
Safety Property Instance i
Recovery Criteria Instance i
Safety Property Instance $i+1$

Figure 4: Timeline diagram, time advancing left to right, illustrating the use of SGSM++ to identify the start of a violation at time v_i, its recovery at time r_i, and the initialization of a the safety property to monitor future violation v_{i+1}.

Second, we consider the impact of extensions in this direction on how the properties' initializations are encoded.

At a high level, Fig. 4 illustrates the timeline of a trace of an AV system being monitored for violations of a safety property specified in LTL$_f$. At time v_i, the i^{th} property violation is identified. In order to count future violations, a recovery criteria specified in LTL$_f$ begins monitoring to determine when the violation has concluded, which eventually occurs at time r_i. Once the recovery formula has been satisfied, the approach must reset the safety property to enable monitoring. However, the conditions that are present at the time of reset may be different from the initial conditions that were assumed by the system and property when monitoring began. To address this, a reset mapping specified in LTL$_f$ is provided that captures the relevant subset of the trace history and uses it to initialize the safety property monitor. This paradigm allows for identifying multiple violations of a property over a single trace and by examining the timing of the violations and subsequent recoveries, the number of total violations and the duration of each violation can be calculated. The following sections briefly motivate and describe this process in more depth.

4.1.1. Counting and Duration

Consider ψ_9 shown in Fig. 1, the original specification from the driving code, § 46.2-821 says "[a] vehicle approaching an intersection on a highway controlled by a stop sign, shall, immediately before entering such intersection, stop [...] before entering" [17]. Though not specified, this specification implicitly applies over *each* intersection that the vehicle approaches. However, it is encoded in SGL as $\mathcal{G}((\neg \mathit{hasStop} \wedge \mathcal{X}(\mathit{hasStop})) \rightarrow (\mathcal{X}(\mathit{hasStop}\,\mathcal{U}\,(\mathit{isStopped} \vee$

$\mathcal{G}(\mathit{hasStop})))))$, which says that *globally* ($\mathcal{G}$), the vehicle must never run a stop sign. This difference between "for each intersection" and "globally for all intersections" leads to different behaviors after the first violation. Under the encoding, once any stop sign has been passed without stopping, the vehicle is in violation, and it cannot recover from this violation, in effect resulting in no additional monitoring for future stop signs. We can imagine a meta-property stating that the vehicle can run no more than N stop signs; for fixed N, this can be encoded by chaining the inner terms of ψ_9, but this cannot monitor for an unbounded number of violations.

Further, some violations lend themselves to a notion of "duration of violation". Consider § 46.2-804 (ψ_1) which states "Wherever a highway is marked with double traffic lines consisting of two immediately adjacent solid yellow lines, no vehicle shall be driven to the left of such lines" [17]. This is encoded as $\mathcal{G}(\neg \mathit{isOppLane})$ to denote that globally the vehicle should not be in the opposing lane. Even with the ability to count the number of times that the vehicle is in the opposing lane, we may additionally want to track a qualitatively different metric capturing the *duration* that the vehicle was in violation, e.g. the amount of time spent in the opposing lane. Under SGL, for specific durations of interest, different parameterizations of the property could be encoded to track, e.g., spending less than 1 second or 10 seconds in the opposing lane, but with finitely many properties there is a strict bound on the number of distinct durations that can be tracked.

This work extends the expressiveness of SGL to handle both counting and duration tracking in Section 4.2.

4.1.2. Handling Re-initialization

As noted in Section 3.1.2, the correctness of the monitor is predicated on maintaining a consistent state between the monitor and the state of the system. This is particularly critical when the monitor is initialized; either external guarantees must be in place to ensure that the system and monitor are in a consistent initial state, or the safety property encoding must be specified so as to not assume a particular prior state, i.e. it must initialize into a "warm up" period where it observes the trace until it can reach a known state. For example, if the AV always begins in a set location such as a parking garage with known characteristics, the safety property could use this assumption in its initialization; by contrast, if there exists a possibility that the monitor is not enabled until the AV is mid-deployment, no such assumptions can be made. Controlling for this initialization can impact the semantics of the encoded safety property. Consider ψ_9, the original specification from the driving code, §46.2-821 says "[a] vehicle approaching an intersection on a highway controlled by a stop sign, shall, immediately before entering such intersection, stop [...] before entering" [17]. However, the natural language description shown in Table 1 says "once the ego vehicle detects a *new* stop signal controlling its lane, it must stop before passing the stop signal". The word "*new*" is added because the specification cannot make any assumptions at initialization. It is possible that at the time the safety monitor is enabled there is a stop sign already present. In such a case, the AV may have stopped prior to the safety monitor being enabled; given this uncertainty, the developers must decide which way to err in implementation: either require the AV to stop, potentially in excess of what is required, or do not require the AV to stop at the first stop sign, potentially missing a violation. The implementation of ψ_9 chooses the latter, only requiring the AV to stop at *new* stop signs. As such, the start state is incorrect to serve as the initial state for reinitialization to monitor for future violations; once a violation has occurred, we know from the history of the scenario that ego should stop for the next stop sign as it definitely has not stopped for it in the past. This highlights how the initial state at monitor startup and the initial state of the monitor when being reset to observe for additional violations may need to be different. We refer to the initial state for future monitors as the *reset state*.

From this understanding of ψ_9, we can then ascribe semantics to each of the four DFA states shown in Fig. 1. We will use this running example to better understand the solution described in Section 4.3.

S1: *(accepting)* The AV is controlled by a stop sign but either has already fulfilled its obligation to stop or does not need to stop because the stop sign was already present at initialization.

S2: *(accepting)* The AV is not controlled by a stop sign.

S3: *(accepting)* The AV is controlled by a stop sign and has not yet fulfilled its obligation to stop.

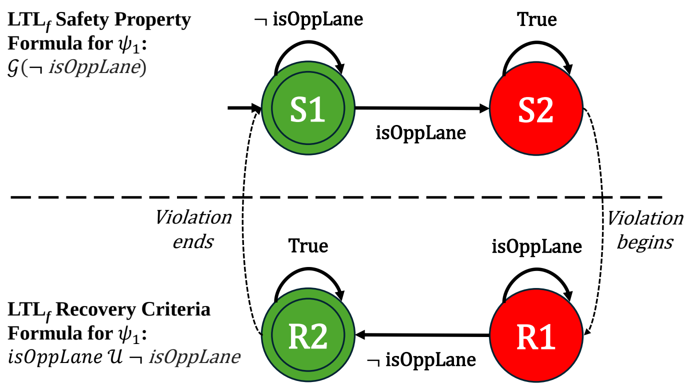
S4: *(failure; trap)* The AV was controlled by a stop sign and did not stop before it stopped being controlled by a stop sign.

4.2. Recovery Criteria Encoding

As discussed, the safety property encoding of a SGL specification must have a unique failure/trap state. In order to count the duration and number

Algorithm 1 SGSM++ violation and duration counting

```
1: procedure VIOLATIONCOUNTDURATION($propDFA, recoveryDFA$)
2:     $currTime \leftarrow 0$
3:     $violationStarts \leftarrow \{\}$                         ▷ Track violation start times
4:     $violationEnds \leftarrow \{\}$                           ▷ Track violation end times
5:                       ▷ Violation start/end time yields count, duration, and timing info
6:     $inViolation \leftarrow False$
7:     while $isRunning$ do
8:         $currentState \leftarrow evalAPs(getCurrSG())$               ▷ Update APs from SG
9:         $propDFA.step(currentState)$                          ▷ Step the property DFA
10:                          ▷ Violation means trap state, is safe to keep stepping
11:        if $\neg propDFA.isAccepting()$ then                            ▷ In violation
12:            if $\neg inViolation$ then
13:                $violationStarts.push(currTime)$                ▷ Mark violation start
14:                $inViolation \leftarrow True$
15:            $recoveryDFA.step(currentState)$                     ▷ Step the reset DFA
16:            if $recoveryDFA.isAccepting()$ then                      ▷ Violation ended
17:                $inViolation \leftarrow False$
18:                $violationEnds.push(currTime)$                    ▷ Mark violation end
19:                $propDFA.reset()$                                   ▷ See Section 4.3
20:                $recoveryDFA.reset()$
21:        $currTime \leftarrow currTime + 1$
22:                      ▷ If ending in violation, there will be an unmatched start time
23:    return $violationStarts, violationEnds$
```



LTL$_f$ Safety Property Formula for ψ_1: $\mathcal{G}(\neg\ isOppLane)$
¬ isOppLane
True
S1
S2
isOppLane
Violation ends
Violation begins
True
isOppLane
LTL$_f$ Recovery Criteria Formula for ψ_1: $isOppLane\ \mathcal{U}\ \neg\ isOppLane$
R2
R1
¬ isOppLane

Figure 5: LTL$_f$ for ψ_1 and its recovery criteria

of violations, there must be a way to exit the trap state. For this, SGSM++ uses a *recovery criteria* that specifies when the safety property DFA should exit the trap state.

Algorithm 1 provides a pseudocode outline of this procedure; we refer to the line numbers in this algorithm in the following explanation. The recovery criteria expresses, through LTL$_f$, that acceptance only occurs when the AV has stopped violating the safety property. Once the safety property DFA has entered the non-accepting state which is also the trap state (line 11), SGSM++ marks the start of a new violation (line 13) and begins evaluating the recovery criteria LTL$_f$ formula (line 15). Once the recovery criteria DFA accepts (line 16), the end of the violation is marked (line 18) and the safety property DFA is reset (line 19); more information about how that reset is performed is explained in the next section. In this paradigm, converse to the safety property encoding, the corresponding DFA for the recovery criteria must have a unique *accepting* state that is also the trap state. The reset DFA is also reset once it accepts (line 20). As shown in Algorithm 1, the implementation tracks the start and end of every violation which can be used to determine the count and duration. This process is visualized in the timeline shown in Fig. 4 where the safety property DFA is evaluating until the violation is identified at time v_i, when the recovery criteria DFA begins evaluating until the end of the violation is identified at time r_i. The duration of the violation is thus $r_i - v_i$.

For example, ψ_1 specifies that the vehicle cannot be in the opposing lane and the safety property is encoded as $\mathcal{G}(\neg \mathit{isOppLane})$. The recovery criteria for ψ_1 is that the vehicle no longer be in the opposing lane, which is encoded as $\mathit{isOppLane}\ \mathcal{U}\ \neg \mathit{isOppLane}$; that is, evaluation continues as long as the AS continues to be in the opposing lane and only accepts when this is no longer the case. This paradigm generalizes; all specifications of the form $\mathcal{G}(\neg a)$ have a natural recovery criteria of $a\ \mathcal{U}\ \neg a$. Fig. 5 illustrates this case through the safety property DFA (top) and the recovery criteria DFA (bottom). During monitoring, the safety property DFA begins in state S1 and stays as long as *isOppLane* remains *True*. Once *isOppLane* is *False*, the safety property DFA transitions to S2. As this is the non-accepting trap state, execution immediately passes to the recovery criteria DFA and the start of a violation is recorded. The recovery criteria DFA starts in state R1 and will remain in this non-accepting state until *isOppLane* is *False*, at which point the recovery criteria DFA will transition to R2 and execution immediately passes to the safety property DFA and the end of the violation is recorded.

The default recovery criteria is *False*, meaning that the specification is irrecoverable. In this way, we can model the properties explored in prior work on SGSM as having a recovery criteria of *False* which is strictly weaker in terms of expressiveness [18]. Under previous work it was not possible to count the number and duration of violations, only whether or not a violation occurred at least once during the trace.

By contrast to irrecoverable specifications, some specifications are instantly recoverable, i.e. there is no sensible concept of "duration of violation". Consider ψ_9 which captures the specification that the AV should not run a stop sign. There is no sensible concept for the duration of running the stop sign—once the AV has passed the stop sign it cannot recover *for that stop sign*; however, it is still useful to track future violations, i.e. future stop signs. In these cases, the recovery criteria is trivial—a criteria of *True* means that the recovery is instantaneous and v_i equals r_i. As shown in Algorithm 1, this will result in the violation start and end being marked during the same iteration through the while loop since the recovery DFA is immediately accepting.

4.3. Reset Mapping Encoding

As noted in Section 4.1.2, the behavior of the safety property must make decisions about how to handle the initialization of the monitor to ensure the system and monitor are in a consistent state. This poses a challenge when resetting the property; a naive implementation could restart the safety property DFA at its start state as it would be during monitor start up. However, at the time of reset there is a known history that led to the violation or its recovery, and so restarting at the initial state of the safety property may not be correct. In the example of ψ_9, the developer may choose to be lenient in the encoding of the property to not enforce the desired behavior until a known state is reached, namely that there are no stop signs so that is guaranteed that the AV must stop for any *new* stop signs.

From the discussion on ψ_9 in Section 4.1.2, state S2 is the only accepting state that corresponds to the AV not being controlled by a stop sign. This is precisely the known state of the system when the violation occurs—a violation means that the AV is no longer controlled by a stop sign. Thus, the correct behavior is to reset to state S2 rather than S1 as we can reason about the semantics of the violation behavior to understand how the prior history informs future potential violations, ensuring that the system and monitor are in a consistent state. It is important to note that not all aspects of the history are useful. In this case, the history of the *hasStop* variable is critical to understanding the known state of the environment for future potential violations; however, the *isStopped* variable's history is not useful as we need to ignore the history of the violation in order to reset to track for future violations. From this understanding that violating ψ_9 guarantees a known history of the AV not being controlled by a stop sign, SGSM++ provides the user with a mechanism to specify what portion of the history should be applied to the future monitoring.

Algorithm 2 SGSM++ reset mapping encoding

```
 1: procedure findResetState($propDFA, resetDFA$)
 2:     $productDFA \leftarrow cartesianProduct(propDFA, resetDFA)$
 3:                                                  ▷ Remove the unsatisfiable transitions
 4:     for $transition \in productDFA.transitions()$ do     ▷ Over $propDFA \wedge resetDFA$
 5:         $predicate \leftarrow transition.predicate()$
 6:         if $unsat(predicate)$ then
 7:             $productDFA.removeTransition(transition)$
 8:                                                  ▷ Iteratively prune unreachable states
 9:     repeat
10:         $prevProductStates \leftarrow |productDFA.states()|$
11:         for $productState \in productDFA.states()$ do
12:             if $isUnreachable(productState)$ then
13:                 $productDFA.remove(productState)$
14:     until $prevProductStates == |productDFA.states()|$
15:                                                  ▷ Find accepting states
16:     $acceptingStates \leftarrow \{\}$
17:     for $(propState, resetState) \in productDFA.states()$ do
18:         if $resetState.isAccepting()$ then
19:             $acceptingStates.add(propState)$
20:     if $|acceptingStates| == 0$ then
21:         $RaiseError$: reset mapping over-constrains history
22:     else if $|acceptingStates| > 1$ then
23:         $RaiseError$: reset mapping under-constrains history
24:     else
25:         return $acceptingStates.pop()$
```

The user specifies the relevant history by providing a *reset mapping* as an LTL$_f$ formula that accepts on exactly the possible histories that inform the known state of the system at the end of the violation. The reset mapping is used to identify which state in the safety property DFA should be used as the reset state as shown in Algorithm 2. In Fig. 4, this is shown above the trace; it is assumed that the reset mapping is accepting at the time that the recovery criteria is accepting and that the history that led to this acceptance provides the information needed to reset the monitor into a state that is consistent with the system. Note that the default reset mapping $\neg\mathcal{F}$ `last`, which only accepts on the empty trace, always results in the reset state being the same as the original start state. This fits our intuitive understanding, as this says that there is no particular history that is relevant to the reset.

First, the cartesian product of the safety property DFA and the reset mapping DFA is computed (line 2). Then, edges which are logically unsatisfiable are removed (lines 4-7). Finally, all unreachable states are iteratively pruned from the resulting DFA (lines 9-14). To find the reset state of the safety property DFA, create an empty set (line 16); then for each state in the product graph that corresponds to an accepting state in the reset mapping graph, add the corresponding safety property state to the set (lines 17-19). If the provided reset mapping is valid, this will result in a set of exactly one state (lines 24-25); if the set is empty (line 20-21) or has more than one state (lines 22-23), then the provided reset mapping is not valid for the safety property because it either over-constrains or under-constrains the set of possible histories such that their application to the safety property state either resulted in no possible states or multiple possible states. This algorithm is run only once at compile time during monitor synthesis and the found reset state is stored for use during online monitoring.

In the case of ψ_9, the desired history captures that in the past ego was controlled by a stop sign until the latest time step when ego was no longer controlled by a stop sign. This is encoded as $\mathit{hasStop}\ \mathcal{U}\ (\neg \mathit{hasStop}\ \land$ `last`$)$; note the use of the `last` keyword to denote end of input as discussed in Section 3.1.1. Running this algorithm on ψ_9 using the reset mapping given above results in state S2 being identified as the reset state as expected.

5. Limitations of Expressiveness

Although the extension from SGSM to SGSM++ has increased the expressiveness of the approach and enabled the monitoring of additional properties and multiple violations, the set of all possible properties is large. We now reflect on the limitations of SGSM++ with respect to its expressiveness and accuracy.

5.1. Temporal Properties over Symbolic Entities

As described in Section 3.2, at each time step the online portion of the framework occurs in two phases: first the AP values are extracted from the current SG, and then these AP values are used to drive the DFA update. This strict boundary between the evaluation of the AP and any temporal information limits the expressiveness of SGSM++ due to not being able to propagate relevant information through time. Consider § 46.2-820 which states "[...] when two vehicles approach or enter an uncontrolled intersection at approximately the same time, the driver of the vehicle on the left shall yield the right-of-way to the vehicle on the right" [17]. In order to encode this specification, we must have information about when ego approaches an intersection and when another vehicle approaches an intersection in order to know if those two actions happen at the same time. A single instance of this interaction could be imagined in LTL$_f$ as:

$$\begin{gathered}(\neg \mathit{egoAtIntersection} \wedge \neg \mathit{otherVehicleAtIntersection}) \wedge \\ \mathcal{X}(\mathit{egoAtIntersection} \wedge \mathit{otherVehicleAtIntersection} \\ \wedge\ \mathit{egoOnLeftOfOtherVehicle}) \\ \rightarrow \\ \mathcal{X}\mathcal{X}(\mathit{egoAtIntersection}\ \mathcal{U}\ \neg \mathit{otherVehicleAtIntersection})\end{gathered}$$

That is, if ego and the other vehicle begin not at the intersection in the first time step, then at the second time step they are both at the intersection and ego is on the left, then until the other vehicle leaves the intersection, ego must continuously wait at the intersection. While the value for *egoAtIntersection* can be directly observed from ego's state and surrounding, determining *otherVehicleAtIntersection* under this context requires not just determining that *some* vehicle is at the intersection at each time step, but that *the same vehicle* is at the intersection for each of these evaluations. This notion of the same vehicle cannot be expressed in general by SGL using the graph querying functions described in Section 3.1.1—either the specification can be encoded to check for some vehicle by using the entity kind filter, e.g. $\mathit{otherCars} = \mathit{filterByAttr}(G \setminus \{\mathit{Ego}\}, \text{kind}, \lambda x : x = \text{car})$ to say that some car but not necessarily the same car each frame is at the intersection, or it can check for a specific car by using a unique identifier attribute, e.g. $\mathit{car01} = \mathit{filterByAttr}(\mathit{otherCars}, \text{uniqID}, \lambda x : x = \text{car01})$ to say that ego should yield to car01[1]. Checking for just some vehicle without guaranteeing that all parts of the equation refer to the same vehicle at all time steps is logically incorrect. Checking for a specific vehicle using a unique identifier will check the correct behavior, but only for the specific vehicle hard-coded into the property. As such, the specification cannot be checked in the general case for yielding to arbitrary vehicles and can only be checked for a predetermined set of identifiers. This comes from the inability to transfer this additional context information through time as only the truth values of the APs can be transferred through time using the LTL$_f$ formula. A more accurate encoding of the property can be imagined as follows—note that the vehicle quantifier binds across time.

$$\begin{aligned}\forall \mathit{vehicle} : \{(\neg \mathit{egoAtIntersection} \wedge \neg \mathit{atIntersection}(\mathit{vehicle}))\wedge \\ \mathcal{X}(\mathit{egoAtIntersection} \wedge \mathit{atIntersection}(\mathit{vehicle}) \\ \wedge\, \mathit{egoOnLeftOf}(\mathit{vehicle})) \\ \rightarrow \\ \mathcal{X}\mathcal{X}(\mathit{egoAtIntersection}\ \mathcal{U}\ \neg \mathit{atIntersection}(\mathit{vehicle}))\}\end{aligned}$$

In this way, such properties require a *symbolic entity* so as to not only evaluate that the property holds, but over which other vehicles. This may be achievable by instantiating separate monitors based on the product space of the quantifiers as this has been successful for other runtime monitoring tasks [43]. However, further work is required to demonstrate that this remains efficient to meet runtime constraints as prior work is exponential in the number of quantifiers. A key benefit of SGSM++ is that it runs in constant time and even with this limitation has demonstrated the ability to encode important safety properties. While we characterize this limitation, we leave its solution to future work.

5.2. Monitoring over Discrete Time

A fundamental limitation of the SGSM++ framework comes from its use of discrete evaluations of the properties and underlying APs in time. While a sufficiently rich logic could leverage continuous time, in practice SGGs can

[1]The ability to produce a consistent unique identifier is itself a hard problem with ongoing research, referred to as "object reidentification" [68, 69, 70] or "object tracking" [71, 72] in the literature.

only be implemented over discrete time. This can be mitigated by increasing the rate at which a property is evaluated, but this will always impact the semantics of the implemented monitor. For example, consider ψ_1 saying ego must not be in the opposing lane. Once an SG is observed where ego is in the opposing lane, the duration of violation is measured until an SG is observed where ego is not in the opposing lane. If ego returned to its own lane and then re-entered the opposing lane between the time two successive SGs were captured and evaluated, then what to the system was two violations appear to the monitor as one longer violation. This manifests not only in violation duration, but also in any internal temporal state.

5.3. Variable Duration Properties

In the LTL$_f$ encoding of properties, any durations required of the property are measured by counting a number of consecutive frames. Counting a duration of N frames thus requires N separate states as each state encodes its position in the sequence. This limits expressiveness as this number must be determined as a part of the property specification and cannot be dynamic in response to the system state. Consider § 46.2-849.B. from the Virginia Driving Code on turn signals that says "Wherever the lawful speed is more than 35 miles per hour, such signals shall be given continuously for a distance of at least 100 feet, and in all other cases at least 50 feet, before slowing down, stopping, turning, or partly turning" [17]. While SGSM could monitor for whether the turn signal is being given continuously until the turn, it cannot be used to track the required distance. A stricter version of the specification could be encoded that leverages the fact that at 35 miles per hour it takes 1.95 seconds to travel 100 feet, and less time than that at higher speeds. Thus, a property that monitors for 1.95 seconds would guarantee that no violations occur, but if ego is travelling at 70 miles per hour will enforce a duration that is twice as long as necessary. A richer logic is required to enable connecting the temporal aspects of the monitor with the dynamic aspects of the system; we leave such exploration to future work.

5.4. Precision and Recall of Scene Graph Generators

The foundation of SGSM entails the use of SGs to capture the relevant information about the AV and its environment which SGL++ then uses to extract the relevant APs to monitor the properties. While SGs are an extensible framework with the potential to encode arbitrary entities, relations, and attributes, in practice there are limits to the SGs due to the precision and recall of modern SGGs. To this end, the set of specifications that can be accurately checked are limited by the SGGs employed during monitoring. Modern SGGs rely on state-of-the-art object detection methods such as Detectron2 [54], etc. to identify entities. Such rich perception systems are still an active field of research, and current methods have limited precision, particularly when evaluating over less common classes [73]. A prior limited-scale study of SGGs over real data showed that for only 60% of images did the SG produced match a human annotation [19]. An inaccuracy in the SG could lead to an incorrect AP as evaluated by the monitor which could lead to erroneous or missed violations or violation recoveries. Such inaccuracies could arise from, e.g., an erroneous inclusion or exclusion of an entity, mislabeling of an attribute, or an incorrectly defined edge We hypothesize that adding a notion of uncertainty or confidence to both the properties and the SGG to enable, e.g., “the vehicle must stop if it is 50% sure there is a stop sign” would aid in this aspect; however, accurately judging confidence and integrating uncertainty across sensor and perception modalities remains a challenge that we leave for future work.

The set of specifications that can be checked by existing SGGs is also limited by the range of available entities, attributes, and relations of the SGG. Consider § 46.2-828.1 of the Virginia Driving Code which states “It shall be unlawful for [...] any motor vehicle intentionally to impede or disrupt a funeral procession” [17]; unless the SGG has a mechanism for perceiving and annotating a funeral procession in the SG, this remains out of reach for runtime monitoring. The space of entities, attributes, and relations described in the driving code and other sources for AV safety properties should serve to inform the development of future SGGs.

6. Study

We aim to answer the following research questions:

RQ#1: What driving properties can SGSM express?

RQ#2: Can SGSM find safety violations in AV systems?

RQ#3: Can SGSM++ identify the count and duration of violations?

6.1. Setup

To evaluate SGSM++'s performance in contrast to SGSM's ability to act as an automatic safety monitor, we need a common execution environment on which to run several AV systems to monitor.

6.1.1. Common Execution Platform

For running the study, we used the CARLA simulator for urban driving [21], which is widely-targeted for AV development due to its realistic environments, complex traffic simulation, and ability to model a variety of relevant road scenarios. CARLA holds a competition called the Autonomous Driving Leaderboard, which provides preconfigured scenarios to challenge the community to create systems that can drive autonomously. The challenge includes a variety of towns, 10 different scenarios, each one of them defining a different traffic situation, and a set of routes. We evaluated the 3 top-ranked systems [74] as of June 2022, using the provided evaluation routes for Town05, that includes 2-lane roads and 3-lane highways; 4-lane and T intersections; traffic lights, stop signs, crossing lanes; and pedestrians, cyclists, cars, and trucks. Particularly relevant to the properties examined later in the study, the evaluation routes pass 27 stop signs and 109 junctions.

We developed an SGG in the form of a Python module that interfaces with the CARLA API to extract the relevant entities, their attributes, and compute their relationships with each other and the road structure. The SGG uses ground truth information from CARLA to include all entities within a $50m$ by $50m$ area horizontally centered on ego and vertically offset to include $45m$ ahead of ego to be consistent with prior work [18, 19]. We adopt the default entity and relationship scheme from prior work on SGs for AVs [56, 19], enriched with additional information to include entities for the lanes, roads, and junctions and their relations based on the flow of traffic. Our unoptimized SGG and annotator take on average 288 ms to create a single SG* and our monitor takes 67 ms to evaluate all properties on it using an Intel Xeon Silver 4216 CPU @ 2.10GHz, 128 GB of RAM, and one Nvidia Titan RTX. While our simulation-based SGG uses ground truth information to eliminate the effects of sensor noise in our study, the current trajectory of SGG research in conjunction with the availability of HD maps for AV systems is promising for implementation of SGSM and SGSM++ on real-world systems.

Table 1: Properties implemented in SGL [18] and SGL++

ψ	VA Code	English Summary of Property	LTL$_f$ Formula over SG propositions	# DFA States
ψ_1	§ 46.2-804	Ego vehicle cannot be in the opposing lane.	$\mathcal{G}(\neg \mathit{isOppLane})$	2
ψ_2	§ 46.2-802	Ego vehicle cannot be out of the road.	$\mathcal{G}(\neg \mathit{isOffRoad})$	2
ψ_3	§ 46.2-802	If ego vehicle is in the right-most lane, then ego vehicle should not steer to the right.	$\mathcal{G}(\mathit{isInRightLane} \wedge \neg \mathit{isJunction} \rightarrow \mathit{isNotSteerRight})$	2
ψ_4	§ 46.2-816	Ego vehicle should not be behind another entity in the same lane whithin 4 meters while travelling at a speed > S.	$\mathcal{G}(\mathit{isNearColl} \rightarrow \neg \mathit{isFasterThanS})$	2
ψ_5	§ 46.2-816	If ego vehicle is between 4 and 7 meters of the closest vehicle in the same lane and then comes within 4 meters of a vehicle in the same lane, throttle must not be positive.	$\mathcal{G}((\mathit{isSuperNear} \wedge \neg \mathit{isNearColl}) \wedge \mathcal{X}(\mathit{isNearColl}) \rightarrow \mathcal{X}(\mathit{isNoThrottle}))$	3
ψ_6	§ 46.2-888	If the ego vehicle is moving and there is no entity in the same lane as the ego vehicle within 7 meters, and there is no red traffic light or stop sign controlling the ego vehicle's lane, then the ego vehicle should not stop.	$\mathcal{G}(\neg \mathit{isStopped} \wedge \neg(\mathit{isSuperNear} \vee \mathit{isNearColl}) \wedge \neg \mathit{hasRed} \wedge \neg \mathit{hasStop} \wedge \mathcal{X}(\neg(\mathit{isSuperNear} \vee \mathit{isNearColl}) \wedge \neg \mathit{hasRed} \wedge \neg \mathit{hasStop}) \rightarrow \mathcal{X}(\neg \mathit{isStopped}))$	2
ψ_7	§ 46.2-804	If ego vehicle is not in a junction, then ego vehicle cannot be in more than one lane for more than T seconds (N samples).	$\neg \mathcal{F}\$[N][\mathit{isMultipleLanes} \wedge \neg \mathit{isJunction}]$	$N{+}1$
ψ_8	§ 46.2-833	Ego vehicle must exit junctions within T seconds (N samples).	$\neg \mathcal{F}\$[N][\mathit{isOnlyJunction}]$	$N{+}1$
ψ_9	§ 46.2-821	Once the ego vehicle detects a new stop signal controlling its lane, it must stop before passing the stop signal.	$\mathcal{G}((\neg \mathit{hasStop} \wedge \mathcal{X}(\mathit{hasStop})) \rightarrow (\mathcal{X}(\mathit{hasStop}\ \mathcal{U}\ (\mathit{isStopped} \vee \mathcal{G}(\mathit{hasStop})))))$	4

Table 2: Intermediate variables used in Atomic Propositions shown in Table 3

Name	SGL expression
egoLanes	$relSet(Ego, \text{isIn})$
egoRoads	$relSet(egoLanes, \text{isIn})$
egoJunctions	$relSet(egoRoads, \text{isIn})$
oppLanes	$relSet(egoLanes, \text{opposes})$
offRoad	$filterByAttr(egoLanes, \text{kind}, \lambda x : x = \text{offRoad})$
rightLanes	$relSet(egoLanes, \text{toRightOf})$
steerRight	$filterByAttr(Ego, \text{steer}, \lambda x : x > 0)$
inEgoLane	$relSetR(egoLanes, \text{isIn}) \setminus \{Ego\}$
nearColl	$relSet(inEgoLane, \text{near\_coll})$
superNear	$relSet(inEgoLane, \text{super\_near})$
egoFasterS	$filterByAttr(Ego, \text{speed}, \lambda x : x > S)$
noThrottle	$filterByAttr(Ego, \text{throttle}, \lambda x : x < \epsilon)$
tLights	$filterByAttr(G, \text{kind}, \lambda x : x = \text{trafficLight})$
redLights	$filterByAttr(tLights, \text{lightState}, \lambda x : x = \text{Red})$
trafLightLns	$relSet(redLights, \text{controlsTrafficOf})$
stopSigns	$filterByAttr(G, \text{kind}, \lambda x : x = \text{stopSign})$
stopSignLanes	$relSet(stopSigns, \text{controlsTrafficOf})$
egoStopped	$filterByAttr(Ego, \text{speed}, \lambda x : x < \epsilon)$
juncRoads	$relSetR(egoJunctions, isIn)$

6.1.2. AV Systems Evaluated

Each AV takes in a list of waypoints from the route and produces at each frame a control for steering, throttle, and brake; each system has different sensors and software. **Interfuser [75]** consists of a Deep Neural Network (DNN) with a transformer [76] architecture, and a controller that generates a set of actions for ego. It takes 3 images from 3 RGB cameras and a cropped center image to focus on distant traffic lights, a LiDAR point cloud, and the GPS coordinates and computes a set of waypoints, an object density map, traffic light state, stop sign presence, and if the vehicle is in a junction. These are fed into the controller to produce the output. **TCP [77]** takes in 1 image from an RGB camera, ego's speed, and the GPS coordinates and uses a DNN composed of a CNN-based image encoder using ResNet34 [78], and two GRU [79] branches for trajectory and control predictions. **LAV [80]** consists of a perception DNN, motion planner, and controller. The DNN consumes 3 images from 3 RGB cameras and a LiDAR point cloud, and outputs a BEV map which is fed to the planner along with the next waypoint

Table 3: Atomic Propositions

Atomic Prop.	SGL expression
isJunction	$\|egoJunctions\| > 0$
isOppLane	$\|oppLanes\| > 0$
isOffRoad	$\|offRoad\| > 0$
isInRightLane	$\|rightLanes\| = 0$
isNotSteerRight	$\|steerRight\| = 0$
isNearColl	$\|nearColl\| > 0$
isFasterThanS	$\|egoFasterS\| = 1$
isSuperNear	$\|superNear\| > 0$
isNoThrottle	$\|noThottle\| = 1$
isMultipleLanes	$\|egoLanes\| > 1$
hasRed	$\|trafLightLns \cap egoLanes\| > 0$
hasStop	$\|stopSignLanes \cap egoLanes\| > 0$
isStopped	$\|egoStopped\| = 1$
isOnlyJunction	$\|egoRoads \setminus juncRoads\| = 0$

coordinates to produce the next 10 future waypoints. The waypoints are passed to the controller along with a braking signal from a binary DNN classifier to compute the output.

6.2. RQ#1. SGSM Properties Evaluated

To evaluate SGSM's ability to encode safe driving properties relevant to AV systems, we selected 9 properties from the laws and best practices of the Virginia Driving Code [17]. Laws were selected to yield a set of properties within scope of current AV systems and diverse in both temporal aspects required to analyze the property compliance and richness of the SG structure required to evaluate the APs.

Table 1 shows the successful encoding of those properties, with their relevant statute, a short English summary, and their encoding using the APs over the SG* composed through the LTL$_f$ formula. Additionally, the number of states in the DFA is shown as a measure of temporal complexity. We note that precisely encoding the semantics of the law is challenging. Returning to the stop sign example, the APs are evaluated over the LTL$_f$ formula to track if *isStopped* is true at least once between *hasStop* becoming true and later becoming false, indicating that ego stopped while being controlled by the stop sign. This is a necessary but insufficient specification to meet the criteria under the law; notably, this does not check that the vehicle stopped *at the stop line* rather than before, nor does it enforce separate stops for successive stop signs along the same lane.

As ψ_4, ψ_7, and ψ_8 contain a threshold parameter, we instantiate 3 versions of each, for a total of 15 monitors. For ψ_4, $S \in \{5, 10, 15\}\frac{m}{s}$ was chosen to represent parking-lot, urban, and suburban driving speeds. For ψ_7, empirical studies found that lane changes take $4.6s$ on average with a std dev of $2.3s$ and max of $13.3s$ [81]; thus we select $T \in \{5, 10, 15\}s$ to represent the average, 2 std dev, and beyond max. For ψ_8, we select $T \in \{5, 10, 15\}s$ as the time to clear the intersection as a left turn across a 4 lane road at $10mph$ takes $5s$, and we allow for a buffer factor of $1 - 3\times$.

We note that while some parameters can be expressed in SGL, others are reliant on the parameterization of the underlying SGG. In ψ_4, ψ_5, and ψ_6, we use 4 and 7 meters as the distance thresholds because these correspond to the 'near collision' and 'super near' relationship used by prior AV SGGs [56, 19]. Further, the underlying laws do not provide concrete values, e.g. the law from ψ_5 says "[...] a motor vehicle shall not follow [a vehicle] **more closely than is reasonable and prudent** [...]" (emphasis added) [17].

Table 4: 20 Sections Randomly Selected from the Virginia Driving Code [17]

Applicable to ego	Expressible by SGSM	Count	Sections
No	N/A	10	808, 819.3:1, 819.9, 831, 866, 873, 876, 882.1, 895, 926
Yes	Yes	7	817, 826, 834, 836, 862, 902, 903
	No	3	816.1, 854, 921

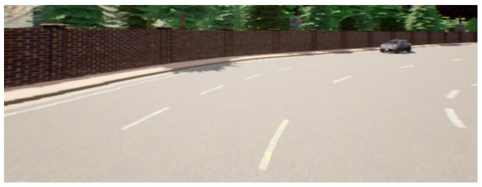
When examining the driving code [17] we find that our framework, equipped with additional entities and attributes, can already encode many additional rules. For example, § 46.2-803, 805, and 807 are all variations on the theme of § 46.2-804 about where the vehicle can operate checked by ψ_1 for different situations, e.g. in traffic circles. Similarly, § 46.2-833, 835, and 836 describe how vehicles must respond to traffic lights. These can be encoded similarly to § 46.2-821 for stopping at stop signs as checked by ψ_9.

However, to obtain a more quantitative grasp of the expressiveness of the DSL, we randomly sampled 20 sections, shown in Table 4, of the 207 sections of the driving code chapter on the regulation of traffic [17]. Of these, we found that 10 applied to AVs while the other 10 were either targeting other non-AV entities, e.g. pedestrians, or covered bureaucratic administration of the code. Of the 10 applicable to AVs, we find that 7 can be encoded through

SGSM, though some require richer SGs than examined in our implementation. For example, § 46.2-817, 834, and 902 concern the AV responding to signals from law-enforcement officers directing traffic. If the SGG could identify law-enforcement officers as entities in the SG and interpret signals from the officer as a relationship between the officer and ego, then SGSM can encode these sections. Of the 3 sections that cannot be encoded in SGSM, one section, § 46.2-816.1, cannot be encoded directly because it does not contain sufficient specificity—the section targets "careless or distracted" driving leading to injury; fully and formally specifying this section is beyond the scope of SGSM and likely generally intractable. The remaining two sections that cannot be encoded concern passing (§ 46.2-854) and following (§ 46.2-921) other vehicles. These sections cannot be encoded due to the limitation of symbolic entities described in Section 5.1 as passing or following a specific vehicle requires tracking that same vehicle through time. Overall, this analysis highlights the expressiveness of SGSM and its utility for the task of runtime monitoring of safety properties, with SGSM able to encode 70% of the applicable properties in this sample.

RQ# 1 Findings: SGSM is able to successfully encode a wide variety of safety properties, with this study demonstrating the successful encoding of 9 properties from the driving code. Further analysis shows that this generalizes to additional properties based on entities and their relationships, showing the potential of SGSM to express many safety driving properties.

6.3. RQ#2. Violations Observed


(a) Interfuser violates ψ_1. Missed road curve, crossed into opp. lane.

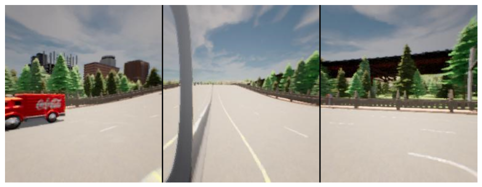

(b) LAV violates ψ_2. Left turn missed lane and drove into median.

Figure 6: Interfuser and LAV safety violations identified with SGSM [18].

As described in RQ#1, we derive 15 properties from the Virginia Driving Code and use SGSM to implement a monitor for each property. We ran each

AV system through the 10 evaluation scenarios of the CARLA leaderboard and separately evaluated all 15 properties at a rate of $2Hz$.

Table 7.1 (left-most set of columns) shows how many of the 10 routes contained at least one violation for each AV system for each of the properties. Note that since the properties are defined as global properties, i.e. once a violation occurs it reaches a trap state, we can track only the first violation, for a maximum of 10 possible violations per AV system per property. We find that the number of violations ranges from 51 for TCP to 72 for Interfuser (over 150 possible violations). Fig. 6a shows an instance of Interfuser violating ψ_1; as the road curved to the right, Interfuser did not steer right enough and drifted into the opposing lane. Fig. 6b shows LAV violating ψ_2, turning left through a junction too sharply, exiting the junction into the median between two lanes. While this is not off of the road bed, the SGG denotes it as off road because it is not part of a defined lane of traffic. Fig. 2 shows TCP violating ψ_9 over a series of frames. TCP approaches a junction with a marked stop line, but it does not stop and enters the junction.

The property violation statistics also give insights into the driving style of the AVs. None of the AVs violated ψ_4, meaning that they maintained sufficient follow distance from lead vehicles. However, we also see that Interfuser and TCP violated ψ_6 over more than half the routes, i.e., they stopped in the middle of the roadway. While we do observe 9 cases where this stoppage is unjustifiable, in 4 other cases we observe that the AV is stopping due to a stopped vehicle ahead of it but farther than the 7 meters prescribed in ψ_6, and in the remaining 4 cases there is a traffic light that is transitioning out of red. This highlights the difficulty in concretizing the parameters used in the specification given the imprecise definitions in the driving manual; 7 meters may be acceptable depending on circumstances. This is further shown in the performance across the parameterizations of ψ_7 and ψ_8. As T increases, the specification is more relaxed which leads to fewer violations; e.g. TCP reduces from 8 routes with violations to 0 under ψ_8 when T is increased from 5 to 10. Although TCP eliminates all violations, Interfuser and LAV do not improve as rapidly. This may point to different AV's optimizations; they likely did not optimize for junction crossing times, and instead may have prioritized moving cautiously through a junction leading to slower transits.

We note that while ψ_3 has an extremely high violation rate with 100% of routes yielding at least one violation, this may point to a weakness in the implementation of ψ_3 rather than of the AVs tested. As discussed in Table 1, ψ_3 says that if ego is in the rightmost lane then it should not steer to the right. The underlying goal of this property is that ego should stay on the roadway and since the rightmost lane necessarily means there is no additional roadway to the right. Thus, the property requires ego to not turn right at all in these cases. However, this is very restrictive and neglects the myriad of cases when turning right could be correct, e.g., if the road is curving to the right. Future refinements of this implementation may consider an adjustment of the property to instead require that ego steer no sharper right than the road is curving right; however, the existing SGG does not annotate the roadway with a notion of curvature, so this information is not currently available at execution time. This highlights the importance to align the property semantics, encoding, and SGG to ensure that reported violations accurately reflect violations of the underlying driving property.

Overall, this highlights three features of SGSM. First, it showcases how it enables the specification and monitoring of driving properties that included entities like *lanes*, *vehicles*, and *traffic signals*; their attributes like *speed* and *color*; and their relations like *is in*, *controls*, and *opposes*. Second, it shows how SGSM can be parameterized to support a rich set of property types, from stateless to temporal, over propositions that are easily accessible through the scene graph. Third, it provides evidence of SGSM's generality as per its direct application to monitor three distinct systems.

RQ# 2 Findings: SGSM is able to operate as a safety monitor to identify property violations at runtime in a blackbox manner. Unlike prior approaches, SGSM is able to operate end-to-end, from sensors to actions, without assuming that certain high-level data is available. As applied to monitor three state-of-the-art research prototype AV systems, SGSM identified that the AVs violated 71% of the encoded safety properties.

6.4. RQ#3: SGSM++ Violation Counts and Durations

Table 5 and Table 6 show the recovery criteria and reset mapping respectively for each of the properties studied which enable us to investigate the count and duration of violations. In their simplest form, each of the studied properties have recovery criteria that emit a DFA with no more than two states; however, the framework is general to handle arbitrarily complex recovery criteria. For example, consider a more advanced version of ψ_1 that focused on aggressive or reckless driving. Virginia Driving Code §46.2-868.1

Table 5: Recovery Criteria encoded with SGSM++

Property	SGL++ Recovery Criteria
ψ_1	$\mathit{isOppLane}\ \mathcal{U}\ \neg \mathit{isOppLane}$
$\psi_1^{\$[30]}$	$\mathcal{F}\$[30][\mathit{isOppLane}]$
$\psi_1^{\$[60]}$	$\mathcal{F}\$[60][\mathit{isOppLane}]$
ψ_2	$\mathit{isOffRoad}\ \mathcal{U}\ \neg \mathit{isOffRoad}$
ψ_3	$\neg \mathit{isNotSteerRight}\ \mathcal{U}\ \mathit{isNotSteerRight}$
ψ_4	$(\mathit{isNearCol} \rightarrow\ \mathit{isFasterThanS})\ \mathcal{U}\ \neg(\mathit{isNearCol} \rightarrow\ \mathit{isFasterThanS})$
ψ_5	$\neg \mathit{isNoThrottle}\ \mathcal{U}\ \mathit{isNoThrottle}$
ψ_6	$\mathit{isStopped}\ \mathcal{U}\ \neg \mathit{isStopped}$
ψ_7	$\neg(\neg \mathit{isMultipleLanes} \vee \mathit{isOnlyJunction})\ \mathcal{U}$ $(\neg \mathit{isMultipleLanes} \vee \mathit{isOnlyJunction})$
ψ_8	$\mathit{isOnlyJunction}\ \mathcal{U}\ \neg \mathit{isOnlyJunction}$
ψ_9	True

Table 6: Reset Mapping encoded with SGSM++

Property	SGL++ Reset Mapping
$\psi_1, \psi_1^{\$[30]}, \psi_1^{\$[60]}, \psi_2, \psi_3, \psi_4, \psi_5, \psi_6, \psi_7, \psi_8$	$\neg \mathcal{F}\ \texttt{last}$
ψ_9	$\mathit{hasStop}\ \mathcal{U}(\neg \mathit{hasStop}\ \vee\ \texttt{last})$

states that "[...] guilty of aggressive driving if [...] violates one or more of the following: §46.2-802 (ψ_2) [...] with the intent to harass, intimidate, injure or obstruct." Further, §46.2-852 states that "[driving] recklessly or at a speed or in a manner so as to endanger the life, limb, or property of any person shall be guilty of reckless driving." In this case, a stronger recovery criteria may be desired that, e.g., ensures that the vehicle has not crossed into the opposing lane for a certain duration before it could be considered no longer in violation. This would allow for the encoding of aggressive or reckless driving in the form of, e.g., repeated swerving across the center line as one violation. The recovery condition given by $\mathcal{F}\$[N][\neg \mathit{isOppLane}]$ will cause the violation to end only when the AV has been not in the opposing lane for N consecutive time steps. We investigate this property as $\psi_1^{\$[N]}$ for $N = 30$ (15 seconds) and $N = 60$ (30 seconds).

Note that the recovery criteria sets the minimal number of steps required for recovery and thus the minimal duration of all violations. This minimal duration is exactly the length of the shortest path between the initial state and the accepting state of the DFA emitted by the recovery criteria. In the case of the automatic recovery given by the condition *True*, this distance is zero since the initial state is also the accept state. For the single condition criteria described in Section 4.2 given by $a\,\mathcal{U}\,\neg a$, this distance is one since, on the step that the property was violated, the condition a was *True*, so it must take at least one step for a to be *False*; this can be seen visually in the lower half of Fig. 5 for ψ_1. In the more involved case about reckless driving, the distance and thus minimal duration is N. This minimal duration is important to consider when interpreting the results and comparing across properties, e.g. for large values of N a violation of duration N may appear to be poor behavior in absolute terms but actually be optimal for that particular property violation.

Whereas Table 7.1 shows the number of routes with at least one violation, Table 7.2 (middle-left) shows the total number of violations across all routes. Further, Tables 7.3 (middle-right) and 7.4 (right-most) show the total duration over all violations, and maximum duration of any single violation respectively in terms of the number of frames over which the violation persisted. Each frame is 0.5 seconds.

First, by comparing Table 7.1 and Table 7.2, we see that almost all properties are violated more than once on at least one route since the total count of violations is greater than the number of routes that had a violation. For some cases, this is markedly so as in ψ_3. As noted in Section 6.3, ψ_3's implementation is stricter than the underlying goal property; this is further made clear by the violation counts shown with over half (823/1405=59%) of all violations observed across all properties coming from ψ_3.

Second, by examining the count and duration we can gain an even clearer picture of the differing driving styles, strengths, and weaknesses of the different AV systems. As noted in Section 6.3, Interfuser and TCP violate ψ_6 at least once in over half of the routes, indicating that they stopped for no reason. While the number of routes with a violation makes Interfuser and TCP seem similar with violations in 9 routes and 6 routes respectively, the count of violations paints a much clearer picture with Interfuser having 141 violations to only 24 for TCP, indicating a clear pattern of violation for Interfuser while TCP's may have been separate isolated incidents. Further, examining the duration of violations in Tables 7.3 and 7.4 shows that when TCP did violate ψ_6 it always recovered on the next frame; meanwhile, Interfuser spent almost 6000 frames, just shy of a third of the 18133 total frames (5812/18133=32%) stopped for no reason, an average of over 40 frames per violation. This behavior appears to be a known weakness of Interfuser—examining its internal code, its controller[2] contains logic that tracks how long it has been stopped for and forces the AV to drive forward if it exceeds a predefined threshold of 60 seconds which is very close to our observed maximum violation duration of 117 frames or 58.5 seconds. This behavior is also related to the longest duration violation observed, where Interfuser violates $\psi_7^{T=5}$ for 1110 frames or 9 minutes and 15 seconds. Examining the data, we see that Interfuser stopped between lanes, and although it moved slightly at least every 58.5 seconds per the data for ψ_6, it did not complete the transition between lanes—the test ended while the system was still in violation.

An additional valuable use case for this data is the identification of weaknesses common across SUTs. Whereas Table 7.1 shows that the SUTs fail to stop at a stop sign at least once in 70% of the routes, the results in Table 7.2 show that the SUTs failed to stop at a combined 63 stop signs, with each SUT missing at least 20. As discussed in Section 6.1.1, the test routes have the SUT transit 27 stop signs in total; thus, the SUTs collectively failed to stop at 63/81=78% of all stop signs. This information helps to clarify that it is not simply a few problematic intersections that cause the SUTs to fail once or twice per route, but instead the SUTs demonstrate pervasive and consistent difficulty in stopping at stop signs.

These data also provide us with a basis to evaluate the more complex reset criteria studied for ψ_1 that can be used for judging, e.g. reckless driving. Recall that ψ_1 checks if ego is in the opposing lane. Recovering from ψ_1 requires ego to not be in the opposing lane for a single frame, while $\psi_1^{\$[30]}$ and $\psi_1^{\$[60]}$ require ego to not be in the opposing lane for 30 and 60 frames, or 15 and 30 seconds, respectively. Examining the total number of violations shown in Table 7.2 for LAV, we see that for the increasingly strict reset criteria, the number of violations goes from 8 to 7 to 6. The drop from 8 violations to 7 violations between ψ_1 and $\psi_1^{\$[30]}$ indicates that there were two separate violations that were more than 1 frame apart but less than 30 frames apart. Likewise, the drop from 7 to 6 between $\psi_1^{\$[30]}$ and $\psi_1^{\$[60]}$ indicates violations that are greater than 30 but less than 60 frames apart. As noted in Section 4.2, the minimum recovery time of $\psi_1^{\$[30]}$ is 30 frames; however, looking at the maximum recovery time for LAV in Table 7.4, it is 75 frames, this is partially due to the second violation causing the 30 second count to reset, leading to a much longer duration of violation. This effectively

[2]https://github.com/opendilab/InterFuser/.../interfuser_controller.py#L262

demonstrates SGSM++'s ability to both encode and measure these complex counting and duration properties.

RQ# 3 Findings: SGSM++ successfully extends the functionality of SGSM by enriching the understanding of violation to include both the count and duration of violation. This allows for fine-grained analysis to identify, e.g., that not only did the three SUTs cross into the opposing lane in 15 different tests, they did so a combined 19 separate times for a total duration of 39 seconds and a maximum duration of 9.5 seconds. This significantly improves the practical utility of SGSM++ to be used as a safety monitor in practice.

6.5. Threats of validity

In this study we showed the feasibility of implementing SGSM and its utility for checking safety property specifications based on driving rules. The external validity of our results, however, is bounded by our use of simulation to create the SGs using ground truth data. Working in simulation enabled us to construct an SGG module that generates high-quality SG representations of the world to judge the cost-effectiveness of the framework as a whole, but we recognize that it will be necessary to consider SGGs using various sensor types and in the wild. The CARLA simulator used in our study may differ from other deployment scenarios, including other simulators based on its particular design goals; further analysis in varied simulation environments is needed to understand the generality of the framework. Moreover, CARLA suffers from the simulation-reality gap [82], so deploying the approach in the real world will be necessary to assess its true potential in the field. The results do provide evidence of the SGSM viability for monitoring richer driving properties, but its violation detection effectiveness will depend on the quality of the systems under test and the scenarios under which those systems are exercised. We explored 3 systems competing under a CARLA benchmarking challenge and a set of predefined scenarios. Pushing SGSM towards commercial systems and richer scenarios will also contribute to generalize the findings. Similarly, more complex domain properties including those involving the behavior of multiple entities over time should be specified and checked to further evaluate the expressiveness and generality of SGSM.

The internal validity of our results is mainly affected by our implementation of SGSM and by our interpretation and encoding of the properties. We

Table 7: Full Study Results for the SUTs Interfuser [75], TCP [77], and LAV [80].

SUT	7.1: # Routes with ≥ 1 Violation (RQ#2) [75]	[77]	[80]	Sum	7.2: Total # Violations (RQ#3) [75]	[77]	[80]	Sum	7.3: Total Duration (# Frames) (RQ#3) [75]	[77]	[80]	Sum	7.4: Max Duration (# Frames) (RQ#3) [75]	[77]	[80]	Max
ψ_1	3	6	6	15	4	7	8	19	10	18	50	78	4	5	19	19
$\psi_1^{\$[30]}$	See RQ#3			15	4	7	7	18	126	221	243	590	33	34	75	75
$\psi_1^{\$[60]}$	See RQ#3			15	4	7	6	17	246	431	376	1053	63	64	148	148
ψ_2	0	0	1	1	0	0	2	2	0	0	17	17	0	0	11	11
ψ_3	10	10	10	30	127	239	457	823	2299	2016	4416	8731	238	97	145	238
$\psi_4^{S=5}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\psi_4^{S=10}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\psi_4^{S=15}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ψ_5	3	2	3	8	3	2	5	10	18	27	13	58	14	25	4	25
ψ_6	9	6	3*	18	141	24	11	176	5812	24	11	5847	117	1	1	117
$\psi_7^{T=5}$	10	5	8	23	22	9	18	49	2237	160	344	2741	1110	55	65	1110
$\psi_7^{T=10}$	5	3	6	14	9	3	9	21	2103	105	187	2395	1100	45	55	1100
$\psi_7^{T=15}$	5	3	5	13	7	3	7	17	2027	75	110	2212	1090	35	45	1090
$\psi_8^{T=5}$	10	8	10	28	43	25	66	134	1258	55	396	1709	608	8	40	608
$\psi_8^{T=10}$	5	0	6	11	5	0	8	13	1090	0	55	1145	598	0	30	598
$\psi_8^{T=15}$	5	0	1	6	5	0	1	6	1040	0	20	1060	588	0	20	588
ψ_9	7	7*	7	21	20	23	20	63	0	0	0	0	0	0	0	0
Sum	72	50	66	188	394	349	625	1368	18266	3132	6238	27636	5563	369	658	5727
Max	10	10	10	30	141	239	457	823	5812	2016	4416	8731	1110	97	148	1110

We identified and fixed an error in the infrastructure for scene graph generation used in prior work [18], which is why these numbers differ from the earlier report (3 used to be 2, and 7* used to be 8).

worked extensively to review the findings from utilizing SGSM to understand and validate that the implemented properties over the SGs faithfully encoded the desired semantics of the specification. Yet, SGs are always an approximation of the real environment attenuated by the SGG through sensors, perception, and other implementation artifacts. As such, the scene graph could include additional entities, not include other entities, or mis-relate certain entities relative to the real environment. For example, we identified a failure of the SG generation wherein the SGG consistently fails to accurately capture the stop sign for one of the intersections. As such, SGSM cannot monitor for violations of ψ_9 at that intersection, and the relevant results of Section 6.4 may under count the stop sign violations by 1 for each of the SUTs. We leave to future work investigating, characterizing, and improving the robustness of SGSM with respect to the effects of these approximations. To mitigate this threat we have released an artifact containing the relevant system and data.

6.5.1. Potential for Field Deployments of SGSM++

Two principle factors impact the potential application of SGSM++ in the real world as a runtime monitor: accuracy and efficiency of its implementation. Our experimental design relied on using the CARLA simulator and its Python API; this allowed our study to produce high-quality SGs and generate the SGs in a manner decoupled from the simulator's internal timing. As discussed, these factors allowed us to examine the expressiveness and utility of SGSM++ in this initial exploration and limit our ability to reason about the potential impacts of SGSM++'s accuracy and timing. The accuracy of SGSM++ is solely affected by the accuracy of the SGG used. The ability of SGSM++ to meet the real-time requirements for field application are affected by the time required for the SGG to create the SG and the time for SGSM++ to evaluate the property over the SG.

We first comment on the accuracy and efficiency of current SGGs. Prior research has conducted initial studies on the accuracy and timing of current research-prototype SGGs on real-world camera images [19]. This exploration found that only 60% of SGs were fully accurate in a small-scale study. Further, the time required for the SGG varied greatly based on the size of the image, ranging from just under 1 second per image for low-resolution images under 100,000 pixels to just over 5 seconds per image for HD images around 2 million pixels on a system with 32 cores and 4 GTX1080Ti GPUs [19]. Overall, this low accuracy, slow performance, and required hardware present substantial obstacles for the implementation of SGSM++ using current SGGs. However, the SGG studied was a research prototype that was not intended for real-time application; we hope that our results will spur future research into optimizing SGGs for real-time application.

As for the time SGSM++ takes to process the SGs and evaluate the properties, as discussed in Section 6.1.1, initial evaluations showed that each SG took less than 100ms to process in our un-optimized Python implementation. This is very promising for real-time application of SGSM++ for runtime monitoring as it indicates that the approach adds a proportionally small overhead compared to generating the SGs themselves. We believe that improvements in the SGG, as shown in recent benchmarks [30, 31], and optimizations in the monitor implementation could push these times to be practicable for application as a real-time monitor.

7. Conclusion

Providing assurances that AVs abide by safe driving properties is key to their successful deployment. However, specifying and monitoring such properties is challenging as they involve reasoning about not only the AV but also its relationship with other entities in the real environment, and such information is not readily accessible. Our previous work introduced the Scene Graph Safety Monitoring (SGSM) framework to better support the specification of safe driving properties and their automatic synthesis into an AV runtime monitor to detect and characterize property violations. In this work, we provide further analysis and formalization of SGSM and extend the framework to produce SGSM++, which captures the semantics of resetting a property violation, allowing the monitor to count the quantity and duration of violations. The study shows the expressiveness of the DSL for specifying 9 real driving properties including the ability to reset these properties for continuous monitoring and the potential for generalization to a broad range of safe driving properties. The study further demonstrates the generality of the monitoring mechanism through its application to 3 off-the-shelf AV systems where it uncovers various driving violations. We find that these AV systems together violate 71% of the properties at least one time, including almost 1400 unique violations over 30 test executions, with violations lasting up to 9.25 minutes; additionally, the AVs fail to stop at stop signs in 78% of cases.

Acknowledgements

This work was supported in part by the National Science Foundation through grant #2129824 and #2312487, the U.S. Army Research Office under grant number W911NF-24-1-0089, and Lockheed Martin Advanced Technology Labs. Trey Woodlief was supported by a University of Virginia SEAS Fellowship. The authors acknowledge Research Computing at The University of Virginia for providing computational resources and technical support that have contributed to the results reported within this publication.

References

[1] R. Bellan, Cruise inches into waymo's territory in the phoenix area, accessed on 02.07.2024 (Aug 2023).
URL `https://techcrunch.com/2023/08/08/cruise-inches-into-waymos-territory-in-the-phoenix-area/`

[2] R. Bellan, Cruise and waymo win robotaxi expansions in san francisco, accessed on 02.07.2024 (Aug 2023).
URL `https://techcrunch.com/2023/08/10/cruise-and-waymo-win-robotaxi-expansions-in-san-francisco/`

[3] A. Marshall, Uber video shows the kind of crash self-driving cars are made to avoid, accessed on 02.07.2024 (Mar 2018).
URL `https://www.wired.com/story/uber-self-driving-crash-video-arizona/`

[4] N. Board, Collision between vehicle controlled by developmental automated driving system and pedestrian. nat. transpot. saf. board, washington, dc, Tech. rep., USA, Tech. Rep. HAR-19-03, 2019. URL https://www. ntsb. gov/investigations … (2019).

[5] B. Templeton, Tesla in taiwan crashes directly into overturned truck, ignores pedestrian, with autopilot on, ForbesAccessed on 02.07.2024 (Jun 2020).
URL `https://www.forbes.com/sites/bradtempleton/2020/06/02/tesla-in-taiwan-crashes-directly-into-overturned-truck-ignores-pedestrian-with-autopilot-on/?sh=20a7458f58e5link`

[6] N. E. Boudette, N. Chokshi, U.s. will investigate tesla's autopilot system over crashes with emergency vehicles, New York TimesAccessed on 02.07.2024 (Aug 2021).
URL https://www.nytimes.com/2021/08/16/business/tesla-autopilot-nhtsa.html

[7] R. Bellan, A waymo self-driving car killed a dog in 'unavoidable' accident, accessed on 02.07.2024 (Jun 2023).
URL https://techcrunch.com/2023/06/06/a-waymo-self-driving-car-killed-a-dog-in-unavoidable-accident/

[8] T. Victor, K. Kusano, T. Gode, R. Chen, M. Schwall, Safety performance of the waymo rider-only automated driving system at one million miles, Tech. rep., accessed on 02.07.2024 (February 2023).
URL https://storage.googleapis.com/sdc-prod/v1/safety-report/Waymo-Safety-Methodologies-and-Readiness-Determinations.pdf

[9] L. Zhang, Cruise's safety record over 1 million driverless miles, accessed on 02.07.2024 (Apr 2023).
URL https://getcruise.com/news/blog/2023/cruises-safety-record-over-one-million-driverless-miles/

[10] H. Araujo, M. R. Mousavi, M. Varshosaz, Testing, validation, and verification of robotic and autonomous systems: A systematic review, ACM Trans. Softw. Eng. Methodol. 32 (2) (mar 2023). doi:10.1145/3542945.
URL https://doi.org/10.1145/3542945

[11] N. Mehdipour, M. Althoff, R. D. Tebbens, C. Belta, Formal methods to comply with rules of the road in autonomous driving: State of the art and grand challenges, Automatica 152 (2023) 110692. doi:https://doi.org/10.1016/j.automatica.2022.110692.
URL https://www.sciencedirect.com/science/article/pii/S0005109822005568

[12] K. Watanabe, E. Kang, C.-W. Lin, S. Shiraishi, Runtime monitoring for safety of intelligent vehicles, in: Proceedings of the 55th annual design automation conference, 2018, pp. 1–6.

[13] J. Stamenkovich, L. Maalolan, C. Patterson, Formal assurances for autonomous systems without verifying application software, in: 2019 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED UAS), IEEE, 2019, pp. 60–69.

[14] A. Kane, O. Chowdhury, A. Datta, P. Koopman, A case study on runtime monitoring of an autonomous research vehicle (arv) system, in: Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings, Springer, 2015, pp. 102–117.

[15] M. Mauritz, F. Howar, A. Rausch, Assuring the safety of advanced driver assistance systems through a combination of simulation and runtime monitoring, in: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II 7, Springer, 2016, pp. 672–687.

[16] K. Leach, C. S. Timperley, K. Angstadt, A. Nguyen-Tuong, J. Hiser, A. Paulos, P. Pal, P. Hurley, C. Thomas, J. W. Davidson, et al., Start: A framework for trusted and resilient autonomous vehicles (practical experience report), in: 2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2022, pp. 73–84.

[17] Virginia code title 46.2 chapter 8 - motor vehicles, regulation of traffic.

[18] F. Toledo, T. Woodlief, S. Elbaum, M. B. Dwyer, Specifying and monitoring safe driving properties with scene graphs, in: 2024 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2024.

[19] T. Woodlief, F. Toledo, S. Elbaum, M. B. Dwyer, S3c: Spatial semantic scene coverage for autonomous vehicles, in: 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), ACM, 2024.

[20] G. De Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence, Association for Computing Machinery, 2013, pp. 854–860.

[21] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, V. Koltun, CARLA: An open urban driving simulator, in: Proceedings of the 1st Annual Conference on Robot Learning, 2017, pp. 1–16.

[22] A. Desai, T. Dreossi, S. A. Seshia, Combining model checking and runtime verification for safe robotics, in: International Conference on Runtime Verification, Springer, 2017, pp. 172–189.

[23] E. Zapridou, E. Bartocci, P. Katsaros, Runtime verification of autonomous driving systems in carla, in: International Conference on Runtime Verification, Springer, 2020, pp. 172–183.

[24] R. Castelino, K. Rothemann, A. Lamm, A. Hahn, Connected vehicle perception monitoring: A runtime verification approach for enhanced autonomous driving safety, in: Proceedings of the 10th International Conference on Vehicle Technology and Intelligent Transport Systems - Volume 1: VEHITS, INSTICC, SciTePress, 2024, pp. 402–409. `doi:10.5220/0012696400003702`.

[25] C. Morse, L. Feng, M. Dwyer, S. Elbaum, A framework for the unsupervised inference of relations between sensed object spatial distributions and robot behaviors, in: 2023 IEEE International Conference on Robotics and Automation (ICRA), 2023, pp. 901–908. `doi:10.1109/ICRA48891.2023.10161071`.

[26] A. Matos Pedro, T. Silva, T. Sequeira, J. a. Lourenço, J. a. C. Seco, C. Ferreira, Monitoring of spatio-temporal properties with nonlinear sat solvers, Int. J. Softw. Tools Technol. Transf. 26 (2) (2024) 169–188. `doi:10.1007/s10009-024-00740-7`.
URL `https://doi.org/10.1007/s10009-024-00740-7`

[27] B. Yalcinkaya, H. Torfah, A. Desai, S. A. Seshia, Ulgen: A runtime assurance framework for programming safe cyber-physical systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2023).

[28] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, U. Topcu, Safe reinforcement learning via shielding, in: Proceedings of the AAAI conference on artificial intelligence, Vol. 32, 2018.

[29] B. Könighofer, J. Rudolf, A. Palmisano, M. Tappler, R. Bloem, Online shielding for reinforcement learning, Innovations in Systems and Software Engineering (2022) 1–16.

[30] PapersWithCode, Scene graph generation on visual genome, accessed on 08.20.2024 (2023).
URL `https://paperswithcode.com/sota/scene-graph-generation-on-visual-genome?metric=mean%20Recall%20%4020`

[31] PapersWithCode, Panoptic scene graph generation on psg dataset, accessed on 08.20.2024 (2023).
URL `https://paperswithcode.com/sota/panoptic-scene-graph-generation-on-psg`

[32] A. Farid, S. Veer, B. Ivanovic, K. Leung, M. Pavone, Task-relevant failure detection for trajectory predictors in autonomous vehicles, in: Conference on Robot Learning, PMLR, 2023, pp. 1959–1969.

[33] C. Luo, R. Wang, Y. Jiang, K. Yang, Y. Guan, X. Li, Z. Shi, Runtime verification of robots collision avoidance case study, in: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Vol. 1, IEEE, 2018, pp. 204–212.

[34] H. Wu, D. Lyu, Y. Zhang, G. Hou, M. Watanabe, J. Wang, W. Kong, A verification framework for behavioral safety of self-driving cars, IET Intelligent Transport Systems 16 (5) (2022) 630–647.

[35] M. Schwammberger, Distributed controllers for provably safe, live and fair autonomous car manoeuvres in urban traffic, 2021.
URL `https://api.semanticscholar.org/CorpusID:237298372`

[36] R. Wang, Y. Wei, H. Song, Y. Jiang, Y. Guan, X. Song, X. Li, From offline towards real-time verification for robot systems, IEEE Transactions on Industrial Informatics 14 (4) (2018) 1712–1721.

[37] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, G. Rosu, Rosrv: Runtime verification for robots, in: Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5, Springer, 2014, pp. 247–254.

[38] S. Kochanthara, T. Singh, A. Forrai, L. Cleophas, Safety of perception systems for automated driving: A case study on apollo, ACM Transactions on Software Engineering and Methodology 33 (3) (2024) 1–28.

[39] H. Torfah, C. Xie, S. Junges, M. Vazquez-Chanlatte, S. A. Seshia, Learning monitorable operational design domains for assured autonomy, in: International Symposium on Automated Technology for Verification and Analysis, Springer, 2022, pp. 3–22.

[40] F. Yang, S. S. Zhan, Y. Wang, C. Huang, Q. Zhu, Case study: Runtime safety verification of neural network controlled system, in: International Conference on Runtime Verification, Springer, 2024, pp. 205–217.

[41] J. Grieser, M. Zhang, T. Warnecke, A. Rausch, Assuring the safety of end-to-end learning-based autonomous driving through runtime monitoring, in: 2020 23rd Euromicro Conference on Digital System Design (DSD), IEEE, 2020, pp. 476–483.

[42] J. Anderson, G. Fainekos, B. Hoxha, H. Okamoto, D. Prokhorov, Pattern matching for perception streams, in: International Conference on Runtime Verification, Springer, 2023, pp. 251–270.

[43] A. Balakrishnan, J. Deshmukh, B. Hoxha, T. Yamaguchi, G. Fainekos, Percemon: online monitoring for perception systems, in: Runtime Verification: 21st International Conference, RV 2021, Virtual Event, October 11–14, 2021, Proceedings 21, Springer, 2021, pp. 297–308.

[44] D. Grundt, A. Köhne, I. Saxena, R. Stemmer, B. Westphal, E. Möhlmann, Towards runtime monitoring of complex system requirements for autonomous driving functions, arXiv preprint arXiv:2209.14032 (2022).

[45] M. Zipfl, N. Koch, J. M. Zöllner, A comprehensive review on ontologies for scenario-based testing in the context of autonomous driving, in: 2023 IEEE Intelligent Vehicles Symposium (IV), 2023, pp. 1–7. `doi:10.1109/IV55152.2023.10186681`.

[46] F. Klueck, Y. Li, M. Nica, J. Tao, F. Wotawa, Using ontologies for test suites generation for automated and autonomous driving functions, in: 2018 IEEE International Symposium on Software Relia-

bility Engineering Workshops (ISSREW), 2018, pp. 118–123. doi:10.1109/ISSREW.2018.00-20.

[47] F. Wotawa, J. Bozic, Y. Li, Ontology-based testing: An emerging paradigm for modeling and testing systems and software, in: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2020, pp. 14–17. doi:10.1109/ICSTW50294.2020.00020.

[48] S. Ulbrich, T. Nothdurft, M. Maurer, P. Hecker, Graph-based context representation, environment modeling and information aggregation for automated driving, in: 2014 IEEE Intelligent Vehicles Symposium Proceedings, 2014, pp. 541–547. doi:10.1109/IVS.2014.6856556.

[49] M. Hülsen, J. M. Zöllner, C. Weiss, Traffic intersection situation description ontology for advanced driver assistance, in: 2011 IEEE Intelligent Vehicles Symposium (IV), 2011, pp. 993–999. doi:10.1109/IVS.2011.5940415.

[50] M. Buechel, G. Hinz, F. Ruehl, H. Schroth, C. Gyoeri, A. Knoll, Ontology-based traffic scene modeling, traffic regulations dependent situational awareness and decision-making for automated vehicles, in: 2017 IEEE Intelligent Vehicles Symposium (IV), 2017, pp. 1471–1476. doi:10.1109/IVS.2017.7995917.

[51] I. Majzik, O. Semeráth, C. Hajdu, K. Marussy, Z. Szatmári, Z. Micskei, A. Vörös, A. A. Babikian, D. Varró, Towards system-level testing with coverage guarantees for autonomous vehicles, in: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE, 2019, pp. 89–94.

[52] J. Johnson, R. Krishna, M. Stark, L.-J. Li, D. A. Shamma, M. S. Bernstein, L. Fei-Fei, Image retrieval using scene graphs, in: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, pp. 3668–3678. doi:10.1109/CVPR.2015.7298990.

[53] X. Chang, P. Ren, P. Xu, Z. Li, X. Chen, A. G. Hauptmann, A comprehensive survey of scene graphs: Generation and application, IEEE Transactions on Pattern Analysis and Machine Intelligence (2021) 1–1doi:10.1109/TPAMI.2021.3137605.

[54] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, R. Girshick, Detectron2, `https://github.com/facebookresearch/detectron2` (2019).

[55] J. Redmon, A. Farhadi, Yolov3: An incremental improvement, arXiv (2018).

[56] A. V. Malawade, S.-Y. Yu, B. Hsu, H. Kaeley, A. Karra, M. A. Al Faruque, roadscene2vec: A tool for extracting and embedding road scene-graphs, Knowledge-Based Systems 242 (2022) 108245.

[57] J. Li, H. Gang, H. Ma, M. Tomizuka, C. Choi, Important object identification with semi-supervised learning for autonomous driving (2022) 2913–2919.

[58] A. Prakash, S. Debnath, J. Lafleche, E. Cameracci, G. State, S. Birchfield, M. T. Law, Self-supervised real-to-sim scene generation, in: 2021 IEEE/CVF International Conference on Computer Vision (ICCV), IEEE Computer Society, Los Alamitos, CA, USA, 2021, pp. 16024–16034. `doi:10.1109/ICCV48922.2021.01574`.
URL `https://doi.ieeecomputersociety.org/10.1109/ICCV48922.2021.01574`

[59] A. Silberschatz, H. Korth, S. Sudarshan, Database systems concepts, McGraw-Hill, Inc., 2005.

[60] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, D. Vrgoč, Foundations of modern query languages for graph databases, ACM Computing Surveys (CSUR) 50 (5) (2017) 1–40.

[61] D. Jackson, Alloy: a lightweight object modelling notation, ACM Transactions on software engineering and methodology (TOSEM) 11 (2) (2002) 256–290.

[62] T. Reinbacher, M. Függer, J. Brauer, Runtime verification of embedded real-time systems, Formal methods in system design 44 (2014) 203–239.

[63] S. Pinisetty, P. S. Roop, S. Smyth, N. Allen, S. Tripakis, R. V. Hanxleden, Runtime enforcement of cyber-physical systems, ACM Transactions on Embedded Computing Systems (TECS) 16 (5s) (2017) 1–25.

[64] H. Jiang, S. Elbaum, C. Detweiler, Reducing failure rates of robotic systems though inferred invariants monitoring, in: 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2013, pp. 1899–1906.

[65] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), ieee, 1977, pp. 46–57.

[66] S. Zhu, G. Pu, M. Y. Vardi, First-order vs. second-order encodings for ltlf-to-automata.

[67] F. Fuggitti, Ltlf2dfa (March 2019). `doi:10.5281/zenodo.3888410`.

[68] M. O. Almasawa, L. A. Elrefaei, K. Moria, A survey on deep learning-based person re-identification systems, IEEE Access 7 (2019) 175228–175247.

[69] M. Ye, J. Shen, G. Lin, T. Xiang, L. Shao, S. C. Hoi, Deep learning for person re-identification: A survey and outlook, IEEE transactions on pattern analysis and machine intelligence 44 (6) (2021) 2872–2893.

[70] H. Wang, J. Hou, N. Chen, A survey of vehicle re-identification based on deep learning, IEEE Access 7 (2019) 172443–172469.

[71] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, K. Schindler, Mot16: A benchmark for multi-object tracking, arXiv preprint arXiv:1603.00831 (2016).

[72] D. K. Dewangan, S. P. Sahu, Real time object tracking for intelligent vehicle, in: 2020 first international conference on power, control and computing technologies (ICPC2T), IEEE, 2020, pp. 134–138.

[73] S. Kothawade, S. Ghosh, S. Shekhar, Y. Xiang, R. Iyer, Talisman: targeted active learning for object detection with rare classes and slices using submodular mutual information, in: European Conference on Computer Vision, Springer, 2022, pp. 1–16.

[74] C. Team, I. A. A. Lab, E. A. Foundation, AlphaDrive, Autonomous driving on carla leaderboard, accessed on 02.07.2024.
URL `https://paperswithcode.com/sota/autonomous-driving-on-carla-leaderboard`

[75] H. Shao, L. Wang, R. Chen, H. Li, Y. Liu, Safety-enhanced autonomous driving using interpretable sensor fusion transformer, in: Conference on Robot Learning, PMLR, 2023, pp. 726–737.

[76] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).

[77] P. Wu, X. Jia, L. Chen, J. Yan, H. Li, Y. Qiao, Trajectory-guided control prediction for end-to-end autonomous driving: A simple yet strong baseline, Advances in Neural Information Processing Systems 35 (2022) 6119–6132.

[78] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[79] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using rnn encoder-decoder for statistical machine translation, arXiv preprint arXiv:1406.1078 (2014).

[80] D. Chen, P. Krähenbühl, Learning from all vehicles, in: CVPR, 2022.

[81] T. Toledo, D. Zohar, Modeling duration of lane changes, Transportation Research Record 1999 (1) (2007) 71–78.

[82] N. Jakobi, P. Husbands, I. Harvey, Noise and the reality gap: The use of simulation in evolutionary robotics, in: European Conference on Artificial Life, Springer, 1995, pp. 704–720.