Scene Flow Specifications: Encoding and Monitoring Rich Temporal Safety Properties of Autonomous Systems

To ensure the safety of autonomous systems, it is imperative for them to abide by their safety properties. The specification of such safety properties is challenging because of the gap between the input sensor space (e.g., pixels, point clouds) and the semantic space over which safety properties are specified (e.g. people, vehicles, road). Recent work utilized scene graphs to overcome portions of that gap, enabling the specification and synthesis of monitors targeting many safe driving properties for autonomous vehicles. However, scene graphs are not rich enough to express the many driving properties that include temporal elements (i.e., when two vehicles enter an intersection at the same time, the vehicle on the left shall yield...), fundamentally limiting the types of specifications that can be monitored. In this work, we characterize the expressiveness required to specify a large body of driving properties, identify property types that cannot be specified with current approaches, which we name *scene flow properties*, and construct an enhanced domain-specific language that utilizes symbolic entities across time to enable the encoding of the rich temporal properties required for autonomous system safety. In analyzing a set of 114 specifications, we find that our approach can successfully encode 110 (96%) specifications as compared to 87 (76%) under prior approaches, an improvement of 20 percentage points. We implement the specifications in the form of a runtime monitoring framework to check the compliance of 3 state-of-the-art autonomous vehicles finding that they violated scene flow properties over 40 times in 30 test executions, including 34 violations for failing to yield properly at intersections. Empirical results demonstrate the implementation is suitably efficient for runtime monitoring applications.

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**; **Software safety**; **Specification languages**; • **Computer systems organization** → *Robotics*; • **Theory of computation** → **Program specifications**.

Additional Key Words and Phrases: runtime verification, autonomous systems, safety properties, scene graphs

ACM Reference Format:
. 2025. Scene Flow Specifications: Encoding and Monitoring Rich Temporal Safety Properties of Autonomous Systems . In *Proceedings of ACM International Conference on the Foundations of Software Engineering (FSE '25).* ACM, New York, NY, USA, 24 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The inability of autonomous systems to meet their safety specifications has led to field failures and even fatalities [7, 8, 46, 65]. In one high-profile incident, a GMC Cruise autonomous vehicle (AV) collided with a fire truck responding to an emergency [58]. The company stated the AV "positively identified the emergency vehicle almost immediately", but had difficulty estimating the path it would take as it "was in the oncoming lane of traffic, which it had moved into to bypass the red light" [21]. Here, the AV failed to meet its safety specification to yield to the emergency vehicle, resulting in a collision.

Current methods for testing and verification with respect to safety specifications are inadequate to build a safety case for autonomous systems due to their inability to scale to the needed scope based on

Author's Contact Information:

FSE '25, June 23–27, 2025, Trondheim, Norway

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

the long-tail distribution of inputs such systems face and their rich requirements. One study estimated that current AV road-testing techniques would need to drive 11 billion miles to demonstrate human-level safety in terms of fatality levels [32]. For comparison, all AVs registered in California drove a combined 9 million miles in 2023 [51]—at that rate, a reliable safety case is over a millennium away.

Runtime verification has emerged as a potential tool to increase safety by monitoring for specification compliance in the field [47, 48]. In this paper we focus on monitoring for specification compliance, i.e. detecting when a specification has been violated. This has applications for safety in several dimensions. First, for the autonomous system itself (ego), the violation may be recoverable; for example, if an AV runtime verification system identifies that the vehicle has crossed into the opposing lane, it can take corrective action. Second, as autonomous systems are increasingly being deployed in fleets, runtime verification at scale can build a safety assurance case by effectively conducting large-scale field-testing with respect to the specifications being evaluated at runtime. Third, this problem setup is extensible—while the original specification of *"do not cross into the opposing lane"* may not render a violation until after the vehicle has entered a dangerous situation, a more restricted version with a safety buffer, e.g. *"do not come within 25cm of the opposing lane,"* would identify a violation with sufficient time to react. Finally, if a runtime monitor can identify not only violations by the ego system but by other systems as well, it can inform the ego system's actions. For example, if a monitor that tracks whether a vehicle yields properly identifies another vehicle acting out of turn, the ego vehicle can take precautionary action.

A robust runtime verification system must be able to reason over the complex environments in which autonomous systems operate and the temporally-rich safety properties that govern their behavior. No prior runtime verification approach succeeds in both dimensions, typically either using inadequate abstractions of the environment [4, 5], approximating safety-properties [50, 66], or both [47, 62]. We further discuss the limitations of prior work in Section 7.

Most closely related to this work is that of Toledo et al. that introduced the Scene Graph Safety Monitor (SGSM) approach [66] that leveraged scene graphs (SGs) to lift from sensors to the semantic space over which autonomous systems' specifications are written. SGs encode relevant entities in the environment as vertices in the graph, and capture pertinent spatial and semantic relationships between entities as edges in the graph. SGSM used a domain-specific language to query the graphs for propositions that could then be used in formulas expressed in linear temporal logic over finite traces (LTL_f) [18] to construct a monitor. However, this two-stage decoupling between the SG query and the LTL_f formula loses crucial temporal information about the connection between different entities and their relationships across time. As prior work applied SGSM to express properties of the driving code of the US state of Virginia [2], we explore this domain for comparison. The core shortcoming of prior work is that they are limited to describing only *scene properties* and are unable to encode and monitor *scene flow properties*.

Definition: A **scene** is a single-instant snapshot of the autonomous system's environment.

Definition: A **scene property** defines the behavior of the autonomous system (ego) based on its relations with entities in the scene, without accounting for how the relationships between ego and those entities change over time.

Definition: A **scene flow property** extends scene properties and defines the behavior of the autonomous system based on its relationships with entities in the scene and how those relationships change with the flow of time across scenes; in each scene, behavior can be described by current and previous relations to current and previous entities.

In Section 4, we introduce SceneFlow, a domain-specific language that enables encoding and monitoring scene flow properties using *symbolic entities*. For example, consider § 46.2-816 of the Virginia driving code that restricts following a vehicle too closely as depicted in the scenes in Figure 1. The scenes in Figure 1a show a violation of the property with the ego vehicle following *the same*

(a) Ego following a van too closely for two time steps (b) Ego performing a lane change to overtake

Fig. 1. Safe driving property: *"a motor vehicle shall not follow another vehicle, trailer, or semitrailer more closely than is reasonable..."* [2]. Specified in SceneFlow by the LTL$_f$ formula $\neg(tooCloseTo(e) \wedge \mathcal{X}\, tooCloseTo(e))$; where e refers to the vehicle being followed. **Left**: property is violated because the same vehicle is being followed over two time steps. **Right**: property is not violated; although ego is too close to *some* vehicle in both time steps, it is not the same vehicle. Prior approaches [66] could not differentiate between these situations.

van too closely in consecutive time steps. The scenes in Figure 1b also show ego following *a vehicle* too closely for two time steps, but it is not *the same vehicle*, so reporting a violation in this case would be erroneous. Since scene properties can only reason about relations in the current scene, they cannot distinguish between these cases; i.e. a vehicle versus the same vehicle—*scene flow properties* bridge this gap, enabling reasoning over current and past relationships to differentiate these cases. To understand the extent to which such distinctions are of practical importance, in Section 3 we studied all 207 sections of the Virginia driving code [2] and identified that of the 114 sections of the driving code that are applicable to autonomous systems, 20% require this level of temporal expressiveness.

Since prior work [66] cannot express these important safety properties, we developed SceneFlow, an approach for specifying and monitoring temporally-rich safety properties of autonomous systems. SceneFlow encodes the flow of information through time by leveraging *symbolic entities* that are bound to portions of the SG and where those bindings persist through time. In Figure 1a, the symbolic entity e is bound to the van which allows § 46.2-816 to be specified precisely. Moreover, in Figure 1b, if e is bound to the van in the first image, then it will not match the SUV that is *tooCloseTo* in the second image, thereby avoiding the erroneous report of a violation.

To showcase the expressiveness and utility of our framework, we demonstrate its application by monitoring NHTSA scenarios in the CARLA Leaderboard 2.0 [10]. The leaderboard contains scenarios defined with a variety of environments including freeways, urban areas, residential districts, and rural settings; a variety of weather conditions like daylight, sunset, fog, and night. Of particular interest, it contains multiple scenarios based on the NHTSA pre-crash scenario typology [1] including negotiations at traffic intersections, yielding to emergency vehicles, avoiding obstacles in the lane, and others. In addition to the NHTSA scenarios, we demonstrate our framework's ability to monitor three state-of-the-art research prototype autonomous vehicles, finding that they violated scene flow properties over 40 times in 30 test executions, including 34 violations for failing to yield properly at intersections. Although we demonstrate the utility of SceneFlow with respect to autonomous vehicles as a means to compare directly with prior work, the framework is general and applicable to many types of autonomous systems consuming complex sensor data that must abide by scene flow properties; we discuss additional use cases in Section 8.

The primary contributions of this paper are: (1) a substantial study specifying real-world requirements for autonomous systems revealing important limitations of prior work; (2) the development of SceneFlow, a novel specification language that addresses those limitations; (3) the development of a highly-optimized monitoring approach that yields orders of magnitude reductions in the cost of monitoring SceneFlow specifications; and (4) an evaluation of state-of-the-art autonomous driving systems demonstrating the breadth and practical effectiveness of SceneFlow monitoring.

2 Background

In this section we briefly summarize work in the area of SG generation, linear temporal logic, and the intersection of the two for checking safe driving properties.

2.1 Scene Graph Generation

Scene Graph Generation (SGG) aims at building a graph that encodes the semantic relationships between objects in a scene, and the interaction of those objects with their surroundings. SGs are highly demanded for visual understanding and reasoning tasks, leading this computer vision subfield to gain a lot of traction in recent years [12, 36].

An SGG maps a set of sensor inputs, I, to an SG, $sgg : I \mapsto SG$. SGs are directed graphs, with a vertex set V that represents the set of entities in a scene, and a set of edges $(u,v) \in E$ describing their relationships. Formally, $G = (V, E : V \mapsto V, kind : V \mapsto K, rel : E \mapsto R, att : V \cup E \mapsto A)$, with functions to access the entity *kind* of a vertex, the *rel*ation type encoded by an edge, and *att*ribute values of vertices and edges.

The SGG process can occur *bottom-up* or *top-down*. Bottom-up requires the identification of objects and their attributes, typically through an object detection network like Yolo [67] or Detectron [71], followed by the identification of the relationships between the detected objects [17, 31, 40, 75]. Top-down aims to detect and recognize the objects and their relationships at the same time [38, 39, 41, 72]. SGGs are being used in many different domains, including image generation [30, 61] and image captioning [27] where having structured semantic information about a scene, represented with an SG, can help improve performance. Other applications include Visual Question and Answering (VQA) where SGs capture the essential information of images, allowing graph-based VQA [73] methods to outperform traditional ones. Or 3D scene understanding [6, 33], that aims to construct 3D SGs, incorporating more accurate relationships in 3D space. More specialized SGGs have emerged for particular domains, such as autonomous vehicles [37, 44, 56], that capture the relevant semantics of driving scenes, e.g. the number of lanes, types of vehicles, and pedestrians.

2.2 Linear Temporal Logic

Linear Temporal Logic (LTL) [55] is a formal language designed for specifying and verifying the behavior of systems that evolve over time, e.g. embedded or cyber-physical systems [29, 54, 57]. It enables the definition of temporal relationships between events or states, using different operators to capture the progression of time. LTL operates over traces of Boolean values encoding the semantics of the events or states. The logical operators include: *And* ($\wedge$), *Or* ($\vee$), *Not* ($\neg$), among others, while the temporal operators are: *Next* ($\mathcal{X}$), *Until* ($\mathcal{U}$), *Always* ($\mathcal{G}$), and *Eventually* ($\mathcal{F}$).

Linear Temporal Logic over Finite traces (LTL_f) [18] is an extension of traditional LTL specifically tailored for tasks that operate over finite time horizons, such as discrete tasks that have a clear start and end point. Its ability to express temporal relationships over finite traces can encode finite temporal events like passing a vehicle or changing lanes, where correctness is tied to specific sequences of events. A property expressed in LTL_f can be transformed into a Deterministic Finite Automaton (DFA) [26, 78], which efficiently checks whether a finite trace satisfies or violates the property, making it well-suited for real-world applications in task planning and rule adherence. A DFA starts from a specific initial state determined by the LTL_f formula and then transitions through states based on the semantics of the formula. A trace ending in an (non) accepting state is (not) in the language of the LTL_f formula.

2.3 Scene Graph for Safety Monitoring

Previous work combined SGs and LTL_f into a framework called Scene Graph Safety Monitoring (SGSM), that enables the specification of driving properties for autonomous vehicles (AVs) [66]. As LTL_f operates over Boolean traces, SGSM developed a domain-specific language called SGL that

enabled the definition of atomic propositions (APs) over SGs and evaluated them to update the DFA state. SGL has three main operations: *relSet* which retrieves the set of vertices that have a specific edge relationship *from* the given set of vertices:

$$relSet : (V_1 \subseteq V, r \in R) \mapsto V_2 \subseteq V \mid V_2 = \{v_2 : v_1 \in V_1 \wedge (v_1, v_2) \in E \wedge rel((v_1, v_2)) = r\}$$

its complement *relSetR* which retrieves the set of vertices that have a specific edge relationship *to* the given set of vertices:

$$relSetR : (V_1 \subseteq V, r \in R) \mapsto V_2 \subseteq V \mid V_2 = \{v_2 : v_1 \in V_1 \wedge (v_2, v_1) \in E \wedge rel((v_2, v_1)) = r\}$$

and *filterByAttr* which selects a subset of the given vertices that have a given attribute:

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

In addition to those graph query operations, SGSM includes numeric comparison operators, Boolean logic, and set manipulation used to convert from vertex sets to Booleans.

SGSM and SceneFlow utilize SGs to describe the autonomous system's environment in a manner to enable reasoning about its compliance with safety properties. Thus, both techniques require that the SGG can identify, with sufficient accuracy and precision, relevant entities and relationships utilized within the regulations and, for SceneFlow, can track these entities over time.

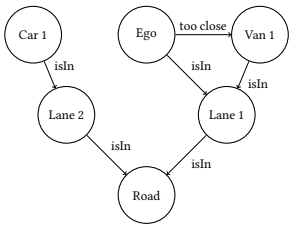
3 Expressivenes Required to Encode the Driving Code

The safety properties governing modern autonomous systems are rich and varied. In this section we perform a study to provide a characterization of such properties and identify expressiveness gaps. Despite SGSM's expressiveness to specify a wide variety of driving properties, as they showcase in their work, there are other properties that cannot be precisely specified but are instead approximated. Given that SGSM was studied by prior work in the context of its ability to express the driving code of the US state of Virginia, we follow their lead and use the Virginia driving code as the basis of analysis in this work. Let us examine one such case where SGSM cannot precisely express the desired property and instead must rely on an approximation. § 46.2-816 from the Virginia driving code states *"a motor vehicle shall not follow another vehicle, trailer, or semitrailer more closely than is reasonable..."* Following the SGSM paradigm, we state this property relative to the ego vehicle, i.e. as a property for the AV. This property can be over approximated by specifying that ego can never be closer than is reasonable[1], i.e. $\mathcal{G}(\neg tooClose)$ where *tooClose* is an AP defined by a graph query that identifies if ego has a "too close" relation with any other entities[2], i.e. $\|relSet(Ego, \text{"too close"})\| > 0$. Here we assume that the SGG defines a "too close" relation between entities. Driving code § 46.2-816 goes on to say *"... than is reasonable and prudent, having due regard to the speed of both vehicles and the traffic on, and conditions of, the highway at the time"*—the precise semantics of this relation would need to be formally encoded in the SGG and could, e.g., include a range of distances, account for the speed of the entities, or adjust for the road conditions.

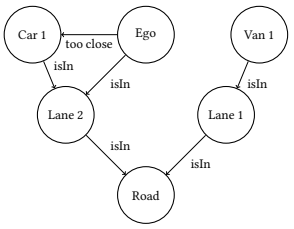
This approximation is sound in that it identifies all violations, but it is incomplete and leads to many false positives—there are many potential situations in which ego is temporarily closer to another vehicle than is reasonable, but does not do so over a period of time to be considered *following*. Attempts to increase the precision by approximating a temporal definition of following are futile; if "following" is defined as two time steps in a row, i.e. $\mathcal{G}(\neg(tooClose \wedge \mathcal{X}\, tooClose))$, then this introduces a new form of imprecision due to SGSM's inability to reason about which entities are involved in the APs. Consider a situation where the ego vehicle is close behind a car in one lane, and then changes lanes and is too close behind a different vehicle in the other lane, shown in Figure 1b with corresponding SGs in Figure 2. In

[1]This is explored in ψ_4 and ψ_5 in the original work on SGSM [66].

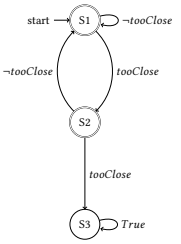
[2]In practice, this should also filter by entity type so as to check that ego is not "too close" to vehicles, trailers, or semitrailers specifically as described in the driving code. This is supported, but is omitted in the running example for brevity.



(a) SG for Fig. 1b at $t=1$: $tooClose = \|relSet(Ego, \text{“too close”})\| > 0 = \|\{\text{Van 1}\}\| > 0 = 1 > 0 = true$
DFA transitions from S1 to S2



(b) SG for Fig. 1b at $t=2$: $tooClose = \|relSet(Ego, \text{“too close”})\| > 0 = \|\{\text{Car 1}\}\| > 0 = 1 > 0 = true$
DFA transitions from S2 to S3 and rejects



(c) DFA for LTL_f of $\mathcal{G}(\neg(tooClose \wedge \mathcal{X}\, tooClose))$; rejects on the sequence of SGs.

Fig. 2. False positive example for § 46.2-816 encoding due to imprecise approximation of “following”.

this case, ego has not *followed* any one vehicle, yet the SGSM scene property encoding cannot identify this. In the first time step, shown on the left in Figure 2a, the AP for *tooClose* evaluates to *true* due to ego having the “too close” relationship with Van 1. In the second time step, shown on the right in Figure 2b, the AP for *tooClose* evaluates to *true* due to ego having the “too close” relationship with Car 1. As demonstrated through the DFA shown in Figure 2c, this sequence of events is rejected by this encoding; although ego did not follow any single vehicle, this information is lost in the Boolean evaluations over the graph as the information about *which vehicle* was being followed cannot be propagated through time. This is a fundamental limitation in the expressiveness of SGSM. Attempts to express such a concept in SGSM would be incomplete and inefficient; doing so would require enumerating separate specifications for each possible entity, e.g. an instantiation for $tooCloseCar1$ and another for $tooCloseVan1$. This is either incomplete by limiting the enumeration below what is encountered at runtime, or inefficient by tracking superfluous specifications. We now examine the impact of this limitation in practice.

Table 1. Necessity of scene flow information in expressing the Virginia driving code [2].
✗= No Support, ◐= Partial Support, ✓= Full Support. **Bold** = requires scene flow

For AV?	Express in SGSM [66]?	Requires flow?	#	Sections within § 46.2
✗	—	—	93	800, 800.2, 800.3, 801, 808, 808.2, 808.3, 809, 809.1, 810, 810.1, 811, 812, 813, 815, 816.1, 817, 818.2, 819, 819.1, 819.2, 819.3, 819.3:1, 819.4, 819.5, 819.6, 819.7, 819.8, 819.9, 819.10, 830.1, 830.2, 831, 832, 833.01, 840, 844, 853, 855, 860, 861, 866, 867, 868, 869, 872, 874.1, 876, 878, 878.3, 879, 880, 882, 882.1, 883, 891, 895, 896, 897, 898, 899, 900, 901, 902.1, 904.1, 906.1, 908, 911, 913, 915, 916, 916.2, 917.1, 917.2, 918, 919, 919.1, 920.1, 920.2, 921.1, 931, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 944.1
✓	✓	—	87	800.1, 802, 803.1, 805, 806, 807, 808.1, 814, 818, 818.1, 825, 827, 828, 828.1, 828.2, 830, 834, 835, 836, 838, 841, 845, 848, 849, 850, 851, 857, 859, 861.1, 862, 870, 871, 873, 873.1, 873.2, 874, 875, 877, 878.1, 878.2, 878.2:1, 881, 884, 885, 886, 887, 888, 889, 890, 892, 893, 894, 902, 903, 904, 906, 908.1, 908.1:1, 908.2, 908.3, 909, 910, 911.1, 912, 914, 915.1, 915.2, 916.1, 916.3, 917, 922, 923, 925, 926, 927, 928, 929, 930, 932, 932.1, 933, 822, 824, 826, 863, 846, 847
	◐	✓	**8**	**803, 804, 821, 833, 905, 907, 920, 924**
	✗	✓	**15**	**816, 820, 823, 829, 833.1, 837, 839, 842, 842.1, 843, 854, 856, 858, 865, 921**
	✗	✗	4	852, 864, 865.1, 868.1

We performed a full analysis of Chapter 8, “Regulation of Traffic”, of the Virginia driving code to categorize which sections are applicable to autonomous systems (AVs), which can be expressed

by SGSM (scene properties) and which ones, like the examples in Sections 1 and 2.3, require more expressiveness to capture relationships with specific entities over time (scene flow properties).

As illustrated in Table 1, of the 207 numbered sections in the code, 114 (55%) are applicable for typical autonomous systems[3]. Of these 114, 87 (76%) can be fully expressed by SGSM, while an additional 8 (7%) are partially[4] expressible by SGSM. We find that for 23 (20%) of the properties, SGSM is specifically limited due to its lack of support for flow properties, i.e. its inability to track relationships with specific entities over time; an approach extending SGSM with scene flow information would support 110 of the 114 properties (96%). The remaining 4 properties are inexpressible due to the imprecise language of the specification. While it is important for the driving code to contain catch-all sections such as § 46.2-864 that prohibits "reckless driving" defined as *"[operating] any motor vehicle at a speed or in a manner so as to endanger the life, limb, or property of any person"* [2], such provisions cannot be readily formalized regardless of the expressibility of the logic.

Qualitatively, we find that many critical safety properties in the driving code are not expressible by prior approaches. Chapter 8 Article 2 "Right-of-Way" contains 12 sections that describe under what situations different vehicles have the right-of-way when driving. Of these, five cannot be encoded under prior approaches because they require scene flow information in order to reason through time about who retains the right-of-way. For example, § 46.2-820 requires that *"when two vehicles approach ... an 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"* [2] while following sections discuss right-of-way for further scenarios, including yielding to emergency vehicles. The right-of-way established by the driving code allows all road users to proceed in a safe and orderly fashion by relieving the individual vehicles of the need to negotiate passage through shared spaces. However, this shared understanding of the right-of-way is only safe if all actors follow the procedure. Prior failures of autonomous systems to abide by the established right-of-way have led to serious accidents [58].

Limitations of expressiveness of prior approaches. We find that while prior approaches are capable of expressing substantial portions (76%) of the driving code examined, a limited (20%) but safety-critical, and in practice common, set of properties remain out of reach of prior approaches due to their inability to express *scene flow properties* that require reasoning about interactions with other entities and the environment through time.

4 Approach

Prior approaches are limited by their inability to express scene flow properties that require reasoning about the relationship between specific entities through the flow of time. We now introduce SceneFlow, a novel approach for expressing such properties over SGs by using a domain-specific language utilizing *symbolic entities* that allow for atomic propositions in LTL_f to reason about the same entity through time. We first describe the syntax and semantics of the domain-specific language, and then describe how the language is utilized to encode the relevant properties, and how these can be leveraged in a monitoring framework as shown in Figure 3 to identify property violations at runtime.

4.1 Language Syntax and Basic Semantics

In this section we describe the syntax of SceneFlow and describe its basic semantics. The following sections further elaborate the semantics with respect to symbolic entities.

[3]The remaining 93 sections handle bureaucratic administration of the code or do not apply to typical autonomous systems, e.g. § 46.2-812 states *"No person shall drive ... for more than thirteen hours in any period of twenty-four hours"*.
[4]Partial indicating that the numbered section contains multiple clauses, some of which are expressible.

An expression in SceneFlow is an LTL$_f$ formula in which the propositions are *symbolic graph queries*. Building from the definition of an LTL$_f$ formula [18], an expression ϕ in SceneFlow is:

$$\phi ::= AP \mid (\neg\phi) \mid (\phi_1 \wedge \phi_2) \mid (\phi_1 \vee \phi_2) \mid (\phi_1 \implies \phi_2) \mid (\mathcal{G}\phi) \mid (\mathcal{F}\phi) \mid (\mathcal{X}\phi) \mid (\phi_1 \mathcal{U} \phi_2)$$

Where $\mathcal{G}$,$\mathcal{F}$,$\mathcal{X}$, and $\mathcal{U}$ are the standard LTL$_f$ operators discussed in Section 2.2. In this expression, AP is a symbolic graph query, $AP : SG \mapsto Boolean$, built up out of Boolean expressions (B) in turn built up out of expressions defining sets of SG vertices (S):

$$\begin{aligned}
AP &::= (\neg B) \mid (B_1 \wedge B_2) \mid (B_1 \vee B_2) \mid (B_1 \implies B_2) \mid (B_1 \oplus B_2) \\
B &::= def(e) \mid (\|S\| > N) \mid (\|S\| < N) \mid (\|S\| \geq N) \mid (\|S\| \leq N) \mid (\|S\| = N) \mid false \mid true \\
S &::= \{e\} \mid sg.V \mid (S_1 \cup S_2) \mid (S_1 \cap S_2) \mid (S_1 \setminus S_2) \mid (S_1 \triangle S_2) \mid relSet(S,r) \mid \\
&\quad relSetR(S,r) \mid filterByAttr(S,m,f) \mid ite(AP,S_1,S_2)
\end{aligned}$$

Where $\neg,\wedge,\vee,\implies$, and $\oplus$ are the logic not, and, or, implication, and exclusive or operators respectively; $\|\cdot\|,\cup,\cap,\setminus$, and $\triangle$ are the set size, union, intersection, difference, and symmetric difference operators respectively; $>,<,\geq,\leq$, and $=$ are the greater than, less than, greater than or equal, less than or equal, and equality test operators; $N \in \mathbb{N}$; ite is the if-then-else operator that evaluates to its second argument if its first argument is true, otherwise it evaluates to its third argument. Here e is an an identifier denoting a symbolic entity which will be described further in Section 4.2. In the semantics, S is a set of vertices in the SG and $sg.V$ is the set of all vertices in the graph ($S \subseteq sg.V$). We can use this syntax to define a standard expression that is common to many SceneFlow expressions: the special set $Ego = filterByAttr(sg.V, name, \text{“}ego\text{”})$, which is the set containing the lone vertex for referring to the ego vehicle in the SG.

Example: Note that SceneFlow subsumes the expressive power of the previous language utilized by SGSM [66], i.e. any expression in SceneFlow that does not utilize symbolic entities could be expressed in SGSM. For example, the expression of the example given in Section 2.3 is valid in both SGSM and SceneFlow and can be fully stated as:

$$\mathcal{G}(\neg((\|relSet(Ego, \text{“too close”})\| > 0)) \wedge (\mathcal{X}(\|relSet(Ego, \text{“too close”})\| > 0)))$$

4.2 Symbolic Entities

Motivating Example: A simple natural language description of the previous property would be *“it must never happen that in two consecutive time steps the set of entities that ego is too close is non-empty”*. As discussed, this does not match the original semantics of the driving code § 46.2-816. An improved but still simple natural language description for the driving code would be *“it must never happen that in two consecutive time steps, ego is too close to some vehicle, **E**”*. Examining the LTL$_f$ from before, this could be written as $\mathcal{G}(\neg(tooCloseTo\boldsymbol{E} \wedge \mathcal{X}\, tooCloseTo\boldsymbol{E}))$. In order to express $tooCloseTo\boldsymbol{E}$ in SGSM, the entity $\boldsymbol{E}$ must be described by a query over the graph. However, any single query over the graph is insufficient as the correct semantics of this property are for it to apply *over all such* $\boldsymbol{E}$ appearing over the trace of graphs. This is the problem that SceneFlow addresses through *symbolic entities*.

A symbolic entity enables the expression of an existential quantifier that refers to the same logical entity across time. Informally, using quantifiers over the possible vehicles, we could reformulate the previous as $\forall e : \mathcal{G}(\neg(tooCloseTo(e) \wedge \mathcal{X}\, tooCloseTo(e)))$, where $tooCloseTo(e)$ is a function over the quantified variable, ensuring that it refers to the same entity through time and checks all such entities.

4.2.1 State Semantics. If e_i are the symbolic entities referenced in a SceneFlow specification ϕ, then the semantics for that formula is: $\forall e_1 \in sg.V \cup \{\perp\} : \ldots : \forall e_n \in sg.V \cup \{\perp\} : \phi$, where the e_i can be bound to any single vertex or to a distinguished $\perp$ value which denotes that e_i is *undefined*.

In this section we first consider such a formula evaluated in a single state of a trace where it has access to $sg.V$ for the SG describing that state. From the grammar, $\{e\}$ may appear anywhere that

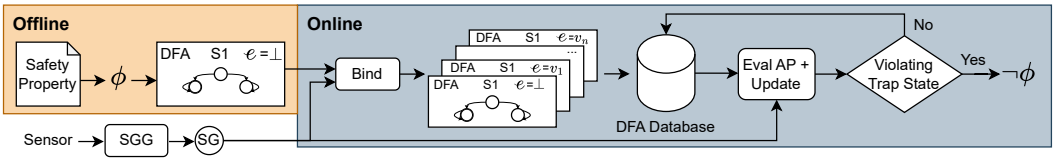


Fig. 3. Monitoring framework for leveraging SceneFlow

a set expression S may appear. The semantics of set operations, S, are defined using the standard set operators if $\{e\}$ is a singleton vertex set. If $e = \perp$ for any operand of a set operation, however, then the result of the expression is $\perp$. The exception being the *ite* function which uses the *optimistic* evaluation scheme discussed below for atomic propositions. Similarly a Boolean expression, B, that involves an operand of $\perp$ value evaluates to $\perp$. The exception to this being the function $def \equiv \lambda x : x \neq \perp$, used to determine whether a symbolic entity is defined. Atomic propositions, AP, are evaluated optimistically relative to $\perp$; i.e. if an operand having a $\perp$ value does not impact the truth value of the AP then it yields that truth value, otherwise it yields $\perp$. For example, $false \wedge \perp = false$ and $false \implies \perp = true$.

4.2.2 Trace Semantics. SceneFlow specifications are defined over sequences of states each defined by an SG. Within each state there is a well-defined value for $sg.V$, but different states may have different vertex sets with some entities appearing later in the trace and others leaving the scene as the trace progresses. Let $sg[i]$ be the SG from the ith step in the trace. The semantics of a formula with symbolic entites over a trace with varying SGs is defined as:

$$\forall i_1 \in [0,m] : \ldots : \forall i_n \in [0,m] : \forall e_1 \in sg[i_1].V \cup \{\perp\} : \ldots : \forall e_n \in sg[i_n].V \cup \{\perp\} : \phi \tag{1}$$

where m is the trace length. This definition allows variables to be defined to take on any possible vertex across the SGs in the trace.

The SGG process must respect the following semantics with respect to $sg.V$ and the intermediate $sg[i]$. Any entity that is contained in the SG at time i must also be present in all subsequent graphs; *and must refer to the same logical entity through time*, $\forall i \in [1,m] : sg[i-1].V \subseteq sg[i].V$. This is because if a symbolic entity e is bound to some vertex $e \in sg[i].V$, then e must also appear in all subsequent SGs so that the information about e is "remembered" through time even if it leaves the observed scene.

As discussed in Section 2.1, the SGG leverages its sensor inputs to identify entities and their physical and semantic relationships in the environment. Let $sensed[i]$ be the SG generated from the sensor input I_i at time i. There may be entities in $sg[i-1].V$ that are not present in $sensed[i].V$, i.e. there exist unseen entities, $unseen = sg[i-1].V \setminus sensed[i].V$, due to, e.g., occlusions, perception failure, or because the entity left the scene. In order to meet the prior invariant, $sg[i].V$ must contain $sg[i-1].V \cup sensed[i].V$ and ensure that the logical entities are aligned to the same vertex. However, simply adding $unseen$ to $sg[i]$ is insufficient—$sg[i]$ must also preserve the relevant relationships of vertices in $unseen$ from $sg[i-1]$. Any implementation of SceneFlow must determine what relationships and attributes of unseen entities should persist with the entity; i.e., what must be observed to be known, and what can be assumed based on prior information. We refer to relationships and attributes that will persist as *static* and those that will not as *dynamic*. For example, if the SG contains entities representing the lanes in the road, then relationships defining which lane can merge into which other lane can likely be assumed static as the road structure will not change even if it is not observed. However, the attribute of a traffic light determining its color is certainly dynamic and should not be assumed based on prior information. Thus, $sg[i].V$ must contain all entities in $unseen$, retaining their static attributes, and additionally $sg[i].E$ must contain all edges $\{(u,v) \in sg[i-1].E : u \in unseen \vee v \in unseen\}$, retaining their static relationships.

Evaluation of an LTL$_f$ formula involves checking a DFA that encodes its temporal structure but this definition implies that the state structure evolves over time as bindings are made. Rather than use a more expressive DFA, like an extended finite state machine with variables and guards, instead we generate *copies* of the DFA specialized to the bindings. As shown from the product of the quantifiers in Equation 1, SceneFlow creates a DFA for all possible bindings of all possible entities at every point in time. This allows us to directly leverage the LTL$_f$ DFA formulation. In the worst case, there are $\Pi_{i\in[0,m]}(\|sg[i]\|+1)^n$ such DFA for a property involving n symbolic entities over a trace of length m, but as we will discuss in Section 5 the vast majority of these can quickly be determined to have reached an accepting trap state at which point they can be ignored.

In practice rather than requiring full information about the trace, these semantics can be realized at each time step, enabling runtime monitoring. As shown in Figure 3, the Bind step generates DFA copies for all possible bindings of the symbolic entities in ϕ. Each DFA copy then evaluates the relevant APs over the SG to update its state. A SceneFlow formula ϕ holds if none of the DFA copies generated by this process reach a non-accepting trap state—states from which it is impossible to subsequently satisfy ϕ. All remaining DFAs from this process are retained to continue evaluation in the next time step, shown in the DFA database in Figure 3.

There are several points to observe about the role of $\perp$ in the evaluation of LTL$_f$ DFA. First, the evaluation of atomic propositions drives transitions through the LTL$_f$ DFA. If an *AP* labelling a transition evaluates to $\perp$ then the transition is not enabled. Second, it is possible that all transitions for a DFA are either $\perp$ or *false* and thus there are no valid state transitions. In this case, that DFA copy is discarded—the associated bindings, or lack of bindings, have no defined semantics with respect to the satisfaction of the expression. Finally, it is possible that a trace terminates with undefined symbolic entities; any DFA that has not yet reached a trap state when the trace terminates are discarded as they have not, and cannot, reach the violation condition.

Example: Let us examine how the semantics of symbolic entities allow us to express the example of following too closely described before. By leveraging a symbolic entity e, the *tooCloseTo(e)*[5] function could be expressed as:

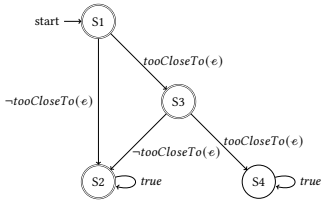
$$tooCloseTo(e) = \|relSet(Ego, \text{“too close”}) \cap \{e\}\| > 0$$

Filling this in for the full expression we have:

$$\neg((\|relSet(Ego, \text{“too close”}) \cap \{e\}\| > 0)) \wedge (\mathcal{X}(\|relSet(Ego, \text{“too close”}) \cap \{e\}\| > 0))$$

4.2.3 Example Trace. Let us revisit the execution of the example trace shown in Figure 2 in Section 2.3 that demonstrated how the SGSM encoding of the property led to a false-positive violation due to SGSM's inability to distinguish between the different entities. Figure 4 shows the SceneFlow encoding of the same property discussed above. The DFA, shown in Figure 4a, contains 4 states; the evaluation proceeds for all possible entity bindings at all times. Figure 4b shows the evaluation of each of the possible DFA copies as described above. At time $t=1$, there are three possible bindings for e, “Van 1”, “Car 1”, and $\perp$. The binding $e=$“Van 1” (ID=1) leads to $tooCloseTo(e)=true$ and advances the DFA to state S3. The binding $e=$“Car 1” (ID=2) leads to $tooCloseTo(e)=false$ and advances the DFA to state S2; since S2 is the accepting trap state, the DFA stops evaluating at this time step since it can never lead to a violation. The binding $e=\perp$ (ID=3) results in a DFA with no viable transitions and stops executing. At time $t=2$, three new DFAs are instantiated to evaluate the three potential bindings starting at this time step. Further, the remaining DFA that was instantiated at time $t=1$, ID=1, continues evaluation; with the updated SG in time $t=2$, the binding $e=$“Car 1” now leads to $tooCloseTo(e)=false$ and causes the DFA to transition from state S3 to the accepting trap state, S2. As demonstrated through all possible instantiations of the DFA, no instantiation leads to a violation, i.e. no DFA reaches state S4. If the trace ends at $t=2$, then ϕ accepts; however, if the trace were to continue, DFA 4 would continue to be active

[5]When describing a Boolean function over symbolic entities, by convention we omit $\{\cdot\}$ in the function call for brevity.



(a) DFA for LTL_f of $\neg(tooCloseTo(e) \wedge \mathcal{X}\, tooCloseTo(e))$

t	DFA ID	Entities	$tooCloseTo(e)$	State	Violation
1	1	e = "Van 1"	*true*	S3	No
1	2	e = "Car 1"	*false*	S2	No
1	3	$e = \perp$	$\perp$	—	No
2	1	e = "Van 1"	*false*	S2	No
2	4	e = "Van 1"	*false*	S2	No
2	5	e = "Car 1"	*true*	S3	No
2	6	$e = \perp$	$\perp$	—	No

(b) Evaluation of all symbolic entities over time

Fig. 4. DFA and evaluation of the SceneFlowSpec expression of the example from Figure 2.

and could identify violations later in the trace along with the additional DFAs that would be created. This demonstrates how the SceneFlow encoding of the property addresses the limitation of SGSM.

A single expression may contain multiple symbolic entities. For example, ϕ_3, further studied in Section 6 to encode the property from § 46.2-821 that the vehicle that arrives at the stop-sign-controlled intersection second must yield to the vehicle that arrived first. This is expressed as:

$$(((atInter(e_1,j)) \wedge \neg(atInter(e_2,j)) \wedge hasStop(e_2)) \wedge \mathcal{X}((atInter(e_1,j) \wedge atInter(e_2,j))))$$
$$\implies \mathcal{X}(\mathcal{X}(((atInter(e_2,j) \wedge \neg fullyInInter(e_2,j))\, \mathcal{U}\, \neg(atInter(e_1,j)))))$$

The semantics of these three entities are: e_1, the vehicle that arrived first at the intersection and has the right-of-way; e_2, the vehicle that arrived second at the intersection, is governed by a stop sign, and thus must yield to e_1; j, the intersection where these vehicles meet—it is important not only that the vehicles are at an intersection, they must both be at *the same* intersection. In the expression, the $atInter(e,j)$ function represents a graph query that checks if vehicle e is at intersection j, the $hasStopSign(e)$ function represents a graph query that checks if a vehicle e is governed by a stop sign, and the $fullyInInter(e,j)$ function represents a graph query that checks if vehicle e is fully in intersection j. In this way, the above can be understood as "if e_1 is at an intersection j and e_2 is not at intersection j, then in the next time step e_1 is still at intersection j and e_2 is newly at intersection j and is governed by a stop sign, then starting in the next time step, e_2 must wait to enter, i.e. be fully in, intersection j until e_1 is no longer at intersection j."

4.3 Property Encoding Patterns

Developers are better able to reason about high-level temporal patterns than the temporal logic expression of the patterns [16]. We developed several adaptations of the Property Specification Patterns (PSP) [23] to fit the semantics of SceneFlow used to specify properties of the Virginia driving code.

Latching Response Chains A two stimulus-one response chain in PSP [23] defines a sequence of two states as a precondition whose occurrence requires a third state – the response – to follow. In SceneFlow a common two-state stimulus comes in the form $(\neg B) \wedge \mathcal{X}(B)$ which defines a *latch* that identifies a specific instant in time when B becomes true. If B is expressed over a set of symbolic entities, e.g. $B = f(e_1,...,e_n)$, then the check for the transition from $\neg f(e_1,...,e_n)$ to $f(e_1,...,e_n)$ between time steps means that the same set of entities will not meet the precondition in future time steps. This pattern makes it possible to correctly check the postcondition only once for a specific binding of $e_1,...,e_n$ that experienced the precondition as follows:

$$\underbrace{(\neg f(e_1,...,e_n)) \wedge \mathcal{X} f(e_1,...,e_n)}_{\text{precondition requires 2 steps}} \implies \underbrace{\mathcal{X}\mathcal{X} postcondition}_{\text{begins evaluation at step 3}}$$

This pattern is a building block of several variant patterns described below.

The Bounded Guarantee Pattern One of the most common use cases for SceneFlow are properties that provide bounded guarantees about an entity's behavior. Consider a situation in which e_2 must yield to e_1. Yielding properties are defined in two stages for the pre and postconditions. The first stage expresses what it means for e_1 to have the right-of-way (the precondition), and the second stage expresses what it means for e_2 to yield (the postcondition) as follows:

$$(\neg atHoldPosition(e_2)) \wedge \mathcal{X}(hasRightOfWay(e_1) \wedge atHoldPosition(e_2)) \implies \mathcal{X}\mathcal{X}(atHoldPosition(e_2)\,\mathcal{U}\,\neg hasRightOfWay(e_1))$$

The precondition leverages a variant of the latching pattern discussed above to identify when e_2 must yield the right-of-way to e_1. The postcondition then uses the until operator $\mathcal{U}$ to describe that e_2 must continue to yield to e_1 until e_1 no longer has the right-of-way.

Table 2 instantiates this pattern to express three driving code properties: ϕ_2,ϕ_3, and ϕ_4. Moreover, variants of bounded guarantee that require certain conditions to be met throughout the duration of the precondition are used to express two more properties: ϕ_5 and ϕ_6.

Time-Bounded Relationship Pattern Another common property in SceneFlow is the time-bounded relationship pattern that states that two entities may not continuously be associated by a given relationship for longer than a certain duration. Consider some function $timeBoundedR(e_1,e_2)$ that is *true* if entity e_1 has some relationship, R, to e_2 that must only exist for a bounded period of time. This property is expressed using a variant of the latching precondition as:

$$(\neg timeBoundedR(e_1,e_2)) \wedge \mathcal{X}(timeBoundedR(e_1,e_2)) \implies \mathcal{X}(\neg\$[N][timeBoundedR(e_1,e_2)])$$

Here, $N \in \mathbb{Z}^+$, and $\$[N][AP]$ is the discrete metric operator explored in prior work that successively applies the $\mathcal{X}$ operator joined by conjunction, e.g. $\$[2][A] = A \wedge \mathcal{X}A$ [66]. Table 2 instantiates this pattern twice to express properties: ϕ_1 and ϕ_8.

The Concurrence Pattern A final pattern that expresses that a transition and a property must happen concurrently. Consider a hypothetical property that says that when an entity exits an intersection, it must exit into the rightmost lane. This would be expressed as:

$$(inInter(e_1)\,\mathcal{U}\,\neg inInter(e_1)) \implies (inInter(e_1)\,\mathcal{U}\,(\neg inInter(e_1) \wedge rightMostLane(e_1))$$

Table 2 instantiates this pattern to express property ϕ_7.

4.4 Limitations

Though SceneFlow enables the encoding and specification of a large proportion of safety properties, including 96% of the relevant Virginia driving code sections per Section 3, there remain limitations that present avenues for future work. First, real-world requirements contain ambiguity and imprecision. For example, the remaining 4% of the driving code studied cannot be encoded due to broadness of catch-all provisions on, e.g., reckless driving. Further, other aspects of the driving code require specific parameter choices to encode, e.g., defining a specific definition of "too close" for § 46.2-816; although the "too close" relationship is readily represented in an SG, SceneFlow relies on the SGG to determine whether such a relationship exists. These requirements were developed for human use, targeting human drivers and human law enforcement; future work may seek to develop AV-specific requirements or enable systems to make determinations specified in natural language [28]. More broadly, future work should investigate the impact of imprecise and inaccurate SGGs on monitoring performance. Another limitation comes from the use of discrete-time LTL_f which uses less precise timing than richer temporal logics. Consider a property that requires a specific duration; e.g. ϕ_1 and ϕ_8 explored in the study in Section 6.1. LTL_f must approximate any duration based on the framerate of the system. However, since all sensors and thus SGGs operate in discrete time, SceneFlow does not introduce a new source of imprecision and the monitor is sound and is precise within the framerate. As such, future work should focus on methods to increase the framerate to decrease the imprecision.

5 Implementation

To study the utility of SceneFlow to express properties and synthesize monitors to detect violations, we developed a Python implementation to study 8 properties in various scenarios. Here we give technical details about the implementation; Section 6 discusses the results of its successful application.

While SceneFlow is general to many applications, our implementation targets the AV domain through the CARLA [22] driving simulator, where we explore its utility under several driving contexts to demonstrate its broad applicability. We utilize an SGG that leverages CARLA's Python API to generate SGs using ground-truth simulator data; the graphs were generated to match the graph abstraction used in SGSM [66]. Leveraging the simulator also allows the SGG to consistently identify the same logical entity through time. In practice, an SGG implemented over sensed data would need to perform this automatically; this is an area with ongoing research, referred to as "object reidentification" [3, 68, 74] or "object tracking" [20, 49]. Leveraging ground-truth SGs allows us to analyze the expressiveness and utility of SceneFlow independently of the SGG component.

At instantiation, the implementation computes the DFA for each property from its LTL_f expression using the LTL_f2DFA Python package [26]. Then, the monitor at each time step: builds the SG from the current environment using the SGG, abiding by the invariants described in Section 4.2.2; evaluates all DFA instances using the SG as described next in Section 5.1.2; and, if any DFA reaches a violating trap state, logs the violation including the binding of the relevant symbolic entities.

5.1 Optimizations

From Section 4.2.2, the worst-case quantity of DFAs that must be evaluated is bounded by $(\|sg.V\|)^{nm}$ for an expression with n symbolic entities and a trace of length m; we now discuss two key optimizations that substantially reduce this burden to be practicable: type information and lazy binding. Section 6.4 explores the efficiency of SceneFlow for practical use in a runtime monitoring framework.

5.1.1 Type Information. The implementation allows for the inclusion of *type information* for the symbolic entities in the description of ϕ. In the SGG utilized in our implementation, each vertex has a special attribute describing the type of the entity, e.g. lane, car, bus. When attempting to bind e_i, only those $\{v \in sg.V : type(v) = type(e_i)\}$ are considered, greatly reducing the space of possible DFAs. The DFAs that are not chosen can be thought of as immediately moving to the accepting trap state—since the binding does not meet the type precondition it trivially cannot lead to a violation. In addition to semantic type information defining the logical class the entity belongs to, the implementation also allows for a second dimension of type information describing whether or not the entity must be observed at the time of binding. Recall from Section 4.2.2 that the set of entities observed at time i, $sensed[i].V$, may not be the complete set of entities that have been seen, $sg[i].V$. Depending on the semantics of the property, it may be sound to only allow for entities to be bound to those that are currently being observed. For example, a vehicle only needs to begin tracking yielding to another vehicle if it can observe the other vehicle at that time. All symbolic entities explored in the study utilize this type definition. This optimization can be particularly useful for long-running traces where the number of entities observed at any particular time is much lower than the number of entities that have ever been seen in the past, i.e. $\|sensed[i]\| \ll \|sg[i].V\|$.

5.1.2 Lazy Binding. In addition to type information, the implementation attempts to delay the binding of an entity as long as possible while monitoring. This allows the evaluation to consider many equivalent entity bindings at once. Conceptually, this process is similar to techniques like CEGAR [14], where a state space is explored quickly by grouping equivalent abstractions and iteratively refining the abstraction if it becomes unsound; however, rather than counter-examples guiding the refinement process, entity bindings are "refined" by trying all concretizations when doing

so could lead to differing outcomes at that step. At each time step, SceneFlow instantiates a new copy of the DFA with all symbolic entities unbound, i.e. $\perp$. Then, the new DFA and any still-running DFAs from previous time steps are evaluated over the current *sg* to determine their next state. For each DFA, if the execution would proceed the same regardless of the binding of an entity, then this allows the binding of $\perp$ to serve as the canonical representation until such time as the binding would differentiate the behavior of the DFA, effectively allowing the implementation to evaluate many equivalent parts of the search-space through the canonical representation. If evaluation would be different, the DFA evaluation branches to consider $e \in sg.V \cup \{\perp\}$. This branching can occur in two possible ways. At each time step, the DFA evaluation attempts to use the current binding, or lack thereof, to evaluate its state transitions; recall that the evaluation is optimistic and attempts to identify a valid state transition if possible. If a valid state transition can be found with the current bindings, no new bindings are made and the execution continues to the next step. If a state transition cannot be found, this means that all potential state transitions were either *false* or $\perp$, and since the set of possible transitions for a DFA must always be complete, at least one transition evaluated to $\perp$. In such case, the set of symbolic entities that are currently unbound and referenced in any transition that evaluated to $\perp$ are identified, and new DFA copies are created to bind those entities. In addition, any evaluation of $def(\perp)$ causes the evaluation to branch; this may alter evaluation in cases of, e.g. $ite(def(e),S_1,S_2)$.

In addition to reducing the number of DFAs being evaluated, the implementation also shares the evaluation of graph queries between different copies of the DFAs. If a DFA transition contains an *AP* expressed over some set of symbolic entities, *E*, then any DFAs that have the same set of bindings with respect to *E* form an equivalence class over that *AP* and the *AP* is evaluated only once per equivalence class; this can greatly reduce the number of queries, particularly in cases where a DFA has a low number of symbolic entities per *AP* relative to the total number of symbolic entities.

These optimizations are critical for the practical application of SceneFlow. To quantify the improvement, to evaluate one property expressed with three symbolic entities for the data collected for the study in Section 6 which analyzed 33 traces (max length 3583, combined length 44455; max 813 entities, combined 13976 entities), an unoptimized version of SceneFlow would need to evaluate on the order of 10^{28000} DFAs. By contrast, our implementation utilized on the order of 10^8.

6 Study

We aim to answer the following research questions to demonstrate the utility of SceneFlow[6].

RQ#1: What driving properties can SceneFlow express beyond prior approaches?
RQ#2: Can SceneFlow identify property violations in specific scenarios?
RQ#3: Can SceneFlow identify property violations in state-of-the-art research AV systems?
RQ#4: Is SceneFlow efficient enough to permit runtime monitoring?

RQ#1 aims to study the expressiveness of SceneFlow to capture the scene flow properties discussed in Section 3. RQ#2 explores SceneFlow's ability to identify violations of these properties by evaluating specific scenarios chosen to exhibit these properties. RQ#3 then explores the broader applicability of SceneFlow to monitor three state-of-the-art AV systems in their test environments; replicating the setup of the experiment in SGSM [66]. Finally, RQ#4 explores the efficiency of the implementation of SceneFlow to determine its viability for runtime monitoring.

6.1 RQ#1: Successful encoding of scene flow properties

SceneFlow is expressive enough to specify all 23 scene flow properties identified in Section 3, and it can do so for both the ego vehicle and all other vehicles simultaneously. We now examine how SceneFlow enables the expression of these properties. Table 2 demonstrates the successful encoding of 8 of

[6]We make our code and results available at: https://anonymous.4open.science/r/SceneFlowLang.

Table 2. Successful encoding in SceneFlow of 8 properties from the Virginia driving code [2]. Symbolic entity variables refer to their type: e is any vehicle, b is any bike, ℓ is any lane, and j is any intersection.

ϕ	§46.2	English description of Property	LTL_f formula
ϕ_1	816	Ego should not follow other vehicles too closely	$\neg(tooClose(e_2,e_1)\wedge sameLane(e_1,e_2)\wedge behind(e_2,e_1)\wedge\neg stopped(e_1))\wedge$ $\mathcal{X}(tooClose(e_2,e_1)\wedge sameLane(e_1,e_2)\wedge behind(e_2,e_1)\wedge\neg stopped(e_1))$ $\Longrightarrow$ $\neg(\$[T][tooClose(e_2,e_1)\wedge sameLane(e_1,e_2)\wedge behind(e_2,e_1)\wedge\neg stopped(e_1)])$
ϕ_2	820	Ego should yield the right-of-way to the vehicle on its right if both arrive at approximately the same time	$((\neg atInter(e_1,j))\wedge\neg(atInter(e_2,j))\wedge hasStop(e_2)\wedge hasStop(e_1))\wedge$ $\mathcal{X}((atInter(e_1,j)\wedge atInter(e_2,j)\wedge toRightOf(e_2,e_1)))$ $\Longrightarrow$ $\mathcal{X}(\mathcal{X}(((atInter(e_2,j)\wedge\neg fullyInInter(e_2,j))\,\mathcal{U}\,\neg(atInter(e_1,j)))))$
ϕ_3	821	Ego should yield the right-of-way to the vehicles at an uncontrolled intersections if they arrived at it earlier	$(((atInter(e_1,j))\wedge\neg(atInter(e_2,j))\wedge hasStop(e_2))\wedge\mathcal{X}((atInter(e_1,j)\wedge atInter(e_2,j))))$ $\Longrightarrow$ $\mathcal{X}(\mathcal{X}(((atInter(e_2,j)\wedge\neg fullyInInter(e_2,j))\,\mathcal{U}\,\neg(atInter(e_1,j)))))$
ϕ_4	829	Ego should yield the right-of-way to emergency vehicles at a signaled intersection	$(\neg(atInter(e_2)))\wedge\mathcal{X}((atInter(e_1,j)\wedge hasEmergencyLights(e_1)\wedge atInter(e_2,j)\wedge notEqual(e_1,e_2)))$ $\Longrightarrow$ $\mathcal{X}(\mathcal{X}(((atInter(e_2,j)\wedge\neg fullyInInter(e_2,j))\,\mathcal{U}\,\neg(atInter(e_1,j)))))$
ϕ_5	839	Ego should overtake a bicycle at a reasonable speed and at least 3 ft to the left of it	$(behind(e_1,b)\wedge safeDistance_D(e_1,b)\wedge\neg(behind(b,e_1))\wedge\mathcal{F}((behind(b,e_1)\vee\neg(safeDistance_D(e_1,b)))))$ $\Longrightarrow$ $\mathcal{X}((safeDistance_D(e_1,b)\,\mathcal{U}\,behind(b,e_1)))$
ϕ_6	843	Ego should not drive to the opposing lane when overtaking another vehicle unless that lane is free of oncoming traffic for a sufficient distance ahead to permit the overtaking	$(behind(e_1,e_2)\wedge opposingClear(e_1,\ell)\wedge\neg(front(e_2,e_1))\wedge onlyIn(e_1,\ell)\wedge$ $\mathcal{F}(((front(e_2,e_1)\wedge onlyIn(e_1,\ell))\vee\neg(opposingClear(e_1,\ell)))))$ $\Longrightarrow$ $\mathcal{X}((opposingClear(e_1,\ell)\,\mathcal{U}\,(front(e_2,e_1)\wedge onlyIn(e_1,\ell))))$
ϕ_7	846	Ego should keep the lane it is driving on after leaving an intersection.	$onlyIn(e,\ell_1)\wedge\mathcal{X}(fullyInInter(e))\wedge\mathcal{X}(\mathcal{X}(((fullyInInter(e)\wedge\neg(onlyIn(e,\ell_2)))\,\mathcal{U}\,onlyIn(e,\ell_2))))$ $\Longrightarrow$ $onlyIn(e,\ell_1)\wedge\mathcal{X}(fullyInInter(e))\wedge\mathcal{X}(\mathcal{X}(((fullyInInter(e)\wedge\neg(onlyIn(e,\ell_2)))\,\mathcal{U}\,(onlyIn(e,\ell_2)\wedge match(\ell_1,\ell_2)))))$
ϕ_8	921	Ego should not follow any emergency vehicle traveling with the sirens on closer than 500 ft	$\neg(tooCloseToEmergency(e_2,e_1)\wedge sameLane(e_1,e_2)\wedge behind(e_2,e_1)\wedge isEmergencyVehicle(e_1))\wedge$ $\mathcal{X}(tooCloseToEmergency(e_2,e_1)\wedge sameLane(e_1,e_2)\wedge behind(e_2,e_1)\wedge isEmergencyVehicle(e_1))$ $\Longrightarrow$ $\neg(\$[T][tooCloseToEmergency(e_2,e_1)\wedge sameLane(e_1,e_2)\wedge behind(e_2,e_1)\wedge isEmergencyVehicle(e_1)])$

the 23 scene flow properties identified in Section 3, so chosen to give a representative sample of the expressiveness of the language. For example, when corresponding properties exist for both stop signs and traffic lights, we explore only the stop sign property. To highlight the novel application of symbolic entities enabled by SceneFlow, each property is described by its temporal logic formula with all AP represented as functions over the relevant symbolic entities. Let us examine one such function defined over two symbolic entities, $atInter(e_1,j)$ which is *true* iff entity e_1 is at intersection j. This is used in ϕ_2, ϕ_3, and ϕ_4 to determine an entity is at the intersection to track yielding. This function is expressed as:

$$atInter(e_1,j) = \|relSet(relSet(relSet(\{e_1\},\text{“}isIn\text{”}),\text{“}isIn\text{”}),\text{“}isIn\text{”})\cap\{j\}\| = 1$$

Relying on the semantics of the SGG that encodes the semantics that entities have an “isIn” relationship with the lanes they occupy, which have an “isIn” relationship with the roads they occupy, which have an “isIn” relationship with the intersections they occupy. Thus, the query finds the set of intersections e_1 is in, and, as a method for checking that the set includes j, checks that the intersection of that set with the set containing j has size 1. For space, we defer the rest of the graph queries used for each of these functions to the online repository.

Note that ϕ_1 and ϕ_8 contain a parameter, T, for the amount of time spent too close to the vehicle that will be considered following. We explore two parameterizations, $T\in\{10{,}50\}$, which correspond to 0.5 and 2.5 seconds as the monitor was evaluated at $20Hz$ in RQ#2. Although 0.5 seconds is unreasonably strict, we include it in the study to demonstrate the functionality of the property as all vehicles studied were too conservative to violate the more relaxed property. Similarly, ϕ_5 is based on § 46.2-839 that prescribes a passing distance of at least 3 feet when overtaking a bicycle, which was implemented by checking that the distance, center-to-center, was at least 2 meters. In the study, all vehicles respected this distance and thus we implemented ϕ_5^*, a variation that increases the safe distance to 7 meters to exhibit violations. This produces 12 parameterizations of the 8 properties.

Leveraging symbolic entities to monitor all road users. The use of symbolic entities not only allows for the reasoning about the ego vehicle's adherence to scene flow properties, but it additionally allows for reasoning about *all* entities' adherence to the properties simultaneously. No formula in Table 2 references ego, instead relying on symbolic entities which can refer to

(a) Scenario S2: Ego arrives at the intersection first (left), yet the ambulance takes the right-of-way (right). Ego abides by ϕ_4, ambulance violates ϕ_3.

(b) Scenario S3: Ego begins to overtake too closely (ϕ_5^*, left), but does not finish before a vehicle comes in the opposing lane (ϕ_6, right).

Fig. 5. Example violations identified in RQ#2. Best viewed on a screen.

ego or any other vehicle. For example, ϕ_3 uses two symbolic entities to check that e_2 appropriately yields to e_1 at an intersection if e_1 arrived first. Through this single definition, SceneFlow automatically checks that all possible combinations of vehicles yielded appropriately to each other. This has potential applications in the field—if an autonomous vehicle detects that another road user is not appropriately yielding, it may need to alter its behavior in response to drive more cautiously. In Table 2, a violation has the semantics that the last-numbered entity is in violation, e.g. in ϕ_3 a violation means e_2 failed to yield to e_1.

Table 3. Property violations in scenarios studied. Properties with no violations omitted.

	ϕ_1^{10}		ϕ_3		ϕ_5^*		ϕ_6	
	e	o	e	o	e	o	e	o
S1	1	-	-	3	-	-	-	-
S2	-	-	-	1	-	-	-	-
S3	2	-	-	-	2	-	1	-
Total	3	-	-	4	2	-	1	-

6.2 RQ#2: Monitoring NHTSA Scenarios

Given CARLA's ability to define different driving conditions, its developers have released two AV challenges: leaderboard 1.0 [9] and leaderboard 2.0 [10] to evaluate the driving proficiency of autonomous systems in realistic traffic scenarios. The leaderboard 2.0 challenge includes several scenarios constructed based on the NHTSA pre-crash typology [1]. In RQ#2 we select three of these scenarios for study, examining scenarios that were crafted to exhibit the behaviors monitored, i.e. that meet the properties' preconditions. Then, in RQ#3 we replicate the study from SGSM to monitor three top-performing systems from leaderboard 1.0 with respect to the scene flow properties.

We selected three scenarios from leaderboard 2.0 to exhibit specific scenarios checked by the properties in Table 2. These scenarios are pre-recorded driving logs provided by CARLA to exhibit a specific behavior; as such, we expected all properties to be satisfied. Scenario S1, called *VehicleTurningRouteLeft*, has ego approach a busy T-intersection to turn left; this targets ϕ_2 and ϕ_3 about respecting right-of-way at the intersection. Scenario S2, called *OppositeVehicleTakingPriority*, has ego arrive at an intersection first, but then an ambulance takes the right-of-way and runs the stop sign; this targets ϕ_4 about yielding to emergency vehicles regardless of timing. Scenario S3, called *HazardAtSideLaneTwoWays*, has ego on a two-lane road following behind two bikes and must cross into the opposing lane to pass them; this targets ϕ_5 and ϕ_6 about overtaking safely.

Table 3 shows the number of violations found per property. As discussed in Section 6.1, the properties are expressed to check violations from all entities, not just the ego vehicle; this is denoted in the two columns for each property with column "e" showing ego's violations and column "o" showing violations by other vehicles. For scenario S1, we find three violations of ϕ_3 targeted by this scenario where in each case another vehicle did not wait its turn at the intersection. Upon examining the trace, these cases all stem from the vehicle coming to a stop several meters behind the stop line at the intersection, thus not meeting the criteria of being *inInter* to claim their spot in the order. For scenario S2, shown in Figure 5a, we see that although ego reached the intersection first, it appropriately yielded to the ambulance, leading to no violation of ϕ_4. However, S2 does show one violation of ϕ_3—the ambulance does not yield to ego. This highlights the importance of identifying

Fig. 6. LAV [13] enters intersection from right-most lane and exits into left-most lane, violating ϕ_7.

Table 4. Property violations in state-of-the-art AV. Properties with no violations omitted.

	ϕ_2		ϕ_3		ϕ_5^*		ϕ_7	
	e	o	e	o	e	o	e	o
TCP [70]	-	-	16	28	-	-	2	-
LAV [13]	-	-	14	23	1	-	7	-
InterFuser [60]	-	1	4	18	1	-	-	1
Total	-	1	34	69	2	-	9	1

interplay between requirements as the isolated text of § 46.2-821 does not contemplate this scenario. This also showcases the utility of SceneFlow monitoring other road users; even if ego did not know the other vehicle was an ambulance, the monitor identifying it stealing the right-of-way could be used to ensure ego takes appropriate precaution to stop to avoid a collision. Finally, scenario S3, shown in Figure 5b, identifies violations of both ϕ_5^* (left) and ϕ_6 (right). In this maneuver, ego attempts to overtake the bike, but in beginning the maneuver invades the extended safety buffer of ϕ_5^*. Then, once ego has passed the bike but is still in the opposing lane, a vehicle appears in that lane heading toward ego, leading to a violation of ϕ_6. We note that both of these properties are parameterized by the amount of buffer that must be afforded, both to the bike and in the opposing lane, and thus the scenario design may have targeted different thresholds. These scenarios demonstrate SceneFlow's ability to successfully monitor for and identify violations of the relevant scene flow properties.

6.3 RQ#3: Monitoring AVs for scene flow properties

We now replicate the settings of the experiment carried out to validate SGSM [66], monitoring the systems under tests' ability to meet the specified scene flow properties, with results shown in Table 4. As discussed in Section 2.3, ψ_4 and ψ_5 implemented for SGSM are over-approximations of ϕ_1. Similarly, while ψ_9 for SGSM monitored that the vehicle stopped for each stop sign, it did not consider yielding as in ϕ_2, ϕ_3, and ϕ_4. Further, ψ_8 for SGSM monitored that the vehicle must exit the intersection in a timely fashion, but not that it exit into the correct lane as in ϕ_7.

We find that these systems are particularly susceptible to not respecting the right-of-way of vehicles that arrived at the intersection before them, with all three systems exhibiting at least one violation and a combined total of 34 violations. Other road users share this same limitation with 69 violations; here, the system under test could leverage SceneFlow's ability to identify when other vehicles act out of turn to take precautionary action. Further, two of the systems exhibit multiple violations of ϕ_7, not turning through the intersection properly as illustrated in Figure 6.

We remark here that these systems were not specifically designed to meet these specifications as these rich temporal properties were not considered in the leaderboard ranking. This further highlights the importance of encoding and monitoring for these properties as enabled by SceneFlow to ensure that autonomous systems that are developed abide by the full set of safe driving properties.

Utility of SceneFlow to express safe driving properties. SceneFlow can express 100% of scene flow properties, detect violations in recorded NHTSA scenarios, and find faults in state-of-the-art autonomous vehicles in simulation, including 34 instances across 30 tests where they failed to yield the right-of-way at an intersection.

6.4 RQ#4: Efficiency for Runtime Monitoring

For SceneFlow to provide utility as a runtime monitoring technique, it must be amenable to efficient execution. While the baseline technique of SGSM has a constant-time complexity [66], the runtime

complexity of SceneFlow depends on the number of entities observed. This raises the question of whether an implementation of SceneFlow is efficient enough for runtime monitoring.

As discussed in Section 6.1, the properties evaluated heretofore track not only whether the ego vehicle violated the property, but also whether any other vehicles violated the property as well by using a symbolic entity to refer to both the entity being monitored and all other entities relevant to the property. This incurs substantial runtime cost as each additional symbolic entity expands the state space; replacing one symbolic entity with the ego vehicle reduces the complexity. Given this trade off, we now focus on monitoring only for properties that track violations by the ego vehicle; full discussion of monitoring for all vehicles is available in the online repository. Another trade off arises in the method of evaluating the properties—evaluating all properties simultaneously in parallel requires less time than running serially, at the expense of requiring additional computational resources. We consider both fully parallelized evaluation and serial evaluation of the 12 parameterizations explored above.

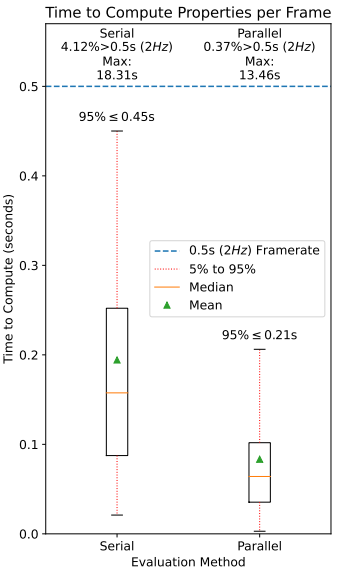


Fig. 7. Box plot of evaluation time per frame for violations by ego.

Following from Section 6.3, we evaluate the ability of our SceneFlow implementation to meet the real-time constraint imposed by the experiment of SGSM in which scene graphs were collected at 2Hz. Setting aside the concern for how long it takes to generate the scene graph which affects both SGSM and SceneFlow, this imposes a maximum of 0.5 seconds per frame to evaluate all properties. To measure the time it takes to evaluate the properties, we ran the evaluation 10 times per configuration on an AMD EPYC 9454, recording the time to evaluate the properties per frame. To eliminate threading effects, the time under parallelization was measured by evaluating each property individually and taking the maximum for the frame.

Figure 7 shows box plots of the amount of time taken for serial and parallel evaluation per frame. The whiskers of the plot show the times for the fastest 5% and 95%. The blue dashed line shows the 0.5s constraint, with the text above detailing how many data points failed to meet this $2Hz$ criteria. Serial evaluation performs slowest; although the median and 95% time per frame were within the constraint, 4.12% of frames took longer than 0.5s, with a maximum time of 18.31s. Switching to parallel evaluation provides a marked improvement, with 95% of frames finishing within 0.21s, and only 0.37% of frames exceeding the constraint with a maximum time of 13.46s.

The discrepancy between the median and maximum evaluation times highlights a key difficulty in evaluating scene flow properties—the evaluation time depends on the quantity of entities in the scene, which is unbounded in the real world. As the number of entities in the scene grows, the time it takes to soundly evaluate the properties may grow beyond the real-time constraint; i.e., stopping evaluation at the real-time constraint may miss violations. Future work should explore methods to prioritize entities and properties for evaluation to minimize false negatives. Consider for example a busy intersection with many vehicles waiting their turn at the stop sign per ϕ_3; while sound evaluation requires considering all vehicles in line, in practice evaluation that prioritizes considering vehicles closer to the intersection would likely lead to few missed violations.

Overall, the implementation of SceneFlow is suitable for real-time operation. Additionally, further optimizations of the research-prototype Python implementation (online), including using a compiled and optimized language and leveraging custom hardware, will improve performance.

Efficiency of SceneFlow for runtime monitoring. The implementation of SceneFlow is suitable for runtime monitoring. When evaluating properties in parallel monitoring for violations by the ego vehicle, 99.63% of frames meet the real-time constraint.

6.5 Threats to Validity

We have demonstrated the successful application of SceneFlow to both express scene flow properties and identify violations of these properties through a runtime monitoring approach leveraging SGs. However, the external validity of our results is limited by our use of simulation. While working in simulation allowed us to collect high-quality SGs to study the efficacy of SceneFlow in isolation, further study on its application to real-world systems is needed to better understand the generality of the approach. Our external validity is further affected by our focus on autonomous vehicles and driving properties; while SceneFlow is broadly applicable as discussed in Section 8, this remains to be empirically validated. The internal validity of our results is impacted by our implementation of SceneFlow and the safety properties studied and expressed through SceneFlow. We have carried out extensive validation efforts to that end and release our data and code to mitigate this threat.

7 Related Work

Prior work has recognized the importance of ensuring autonomous systems abide by their safety requirements. However, no prior work is suitable for runtime monitoring of scene flow properties. We now briefly present work on specifying and monitoring properties for autonomous systems.

7.1 Safety Property Specification

A recent survey on leveraging formal methods to comply with driving rules for autonomous driving [48] highlights the importance of formalizing driving behaviors. Recent works have studied driving codes from different countries, aiming to encode portions of their rules using formal methods. For instance, Esterle et al. [24] analyzes the German concretization of the Vienna convention on road traffic, encoding portions in LTL. Zhang et al. [77] studied the US Department of Motor Vehicles (DMV) driver manual and encoded some driving rules using their custom framework, AVChecker. Kochanthara et al. [34] analyze the Dutch highway manual and analyze whether those requirements are met at the design level of two AV systems. Nonetheless, none of them assessed what percentage of the driving rules they were able to encode using formal specifications. We perform a full analysis of the relevant Virginia driving code [2] and found that 96% can be encoded using SceneFlow. Sun et al [63, 64] use STL to fully encode Chinese traffic laws; however, this effort only encodes the temporal aspect without regard for extracting atomic propositions from sensor data. The `PriorityV` Boolean variable [64] assumes an external oracle of whether another vehicle has priority at an intersection whereas SceneFlow enables reasoning about such a vehicle directly (ϕ_3). Complementary to this effort, there are on-going works that explore the usage of rulebooks [11, 15], which impose a partial order on the set of properties to prevent, e.g., the conflict between ϕ_3 and ϕ_4. We leave extensions applying rulebooks to SceneFlow for future work.

7.2 Safety Monitors for Autonomous Systems

Prior work has developed monitors for AV subcomponents like adaptive cruise control [76], collision avoidance [42, 43, 45], trajectory prediction [25], or lane changing and overtaking [59, 69]. In contrast, other works have focused on end-to-end AV systems including using Signal Temporal Logic (STL) [19, 63, 64, 76], Linear Temporal Logic (LTL) [47, 62] and First Order Logic (FOL) [50] to monitor different systems. Two particularly relevant works introduce spatial relationships between different entities in

a scene using graph representations [50] and metric spaces respectively [47]. Further, reinforcement learning shielding has been explored to increase the robustness of the agent behavior by learning [4] and enforcing [35] safety properties using temporal logic; similar approaches have sought to integrate runtime monitoring with the system's decision making to ensure compliance [64]. Nonetheless, these techniques suffer from either one or both of the following two main limitations. First, despite using temporal properties, they lack a mechanism to reason about the same entity through time—a core requirement of scene flow properties we tackle through SceneFlow. Second, prior work takes for granted the evaluation of the atomic propositions (APs), ignoring the fact that a mapping between sensor inputs and the AP values is needed. To overcome the second issue, Anderson et al. [5] introduced spatial regular expressions to match different patterns over a sequence of images, but it is limited by only reasoning about 2D bounding boxes, which over-approximate the shape of objects and are imprecise for 3D reasoning. In this work we use SGGs, detailed in Section 2.1, to convert the sensor inputs into SGs, which we then use for evaluating the APs; SGGs are an active area of research and continually improving [52, 53].

8 Beyond Autonomous Vehicles and Driving Properties

The development of SceneFlow was motivated by the limited expressiveness of previous work to capture common safe driving properties. However, SceneFlow is applicable to any autonomous system that must abide to specifications over a complex spatio-temporal context captured through rich multidimensional sensors. For example, pick-and-place robots in a warehouse, using LiDAR and cameras, would use SGs to capture the distribution of the objects to manipulate and the surfaces where they sit, and properties specified through SceneFlow would constrain how and in what sequence those objects must be manipulated in order to avoid breakages. Surgical robots assisting doctors would use SGs to recognize organs, surgeons' hands, surgery instruments, and properties specified through SceneFlow would control that the right instruments are used in the right order and on the right organs. Finally, drones in a swarm would use SGs to capture their position and line of sight to peer drones, and specify properties in SceneFlow to enforce swarm formation maneuvers and formation adjustments in the presence of external entities.

9 Conclusion

In this work we have: (1) provided a characterization of the space of safe driving properties and identified expressiveness gaps in existing specification languages, (2) designed a domain-specific language SceneFlow that addresses the significant gap of scene flow properties through the novel use of symbolic entities, and (3) implemented a highly-optimized monitoring approach showing the application of SceneFlow in practice operating under various simulation scenarios and target systems demonstrating the potential of the approach to detect complex but common property violations involving multiple entities with rich relationships manifested over time. As part of the future work, we aim to apply SceneFlow in the field with a full end-to-end pipeline including SGGs captured from sensor data, for more complex situations including for swarm and platoon deployments, and for additional autonomous systems such as those discussed in Section 8.

10 Data Availability

Our artifact, including the SceneFlow implementation, the study data, a replication package, and details about the properties examined, are available at https://anonymous.4open.science/r/SceneFlowLang.

References

[1] [n. d.]. Pre-Crash Scenario Typology for Crash Avoidance Research.

[2] [n. d.]. Virginia Code Title 46.2 Chapter 8 - Motor Vehicles, Regulation of Traffic.

[3] Muna O Almasawa, Lamiaa A Elrefaei, and Kawthar Moria. 2019. A survey on deep learning-based person re-identification systems. *IEEE Access* 7 (2019), 175228–175247.

[4] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. 2018. Safe reinforcement learning via shielding. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[5] Jacob Anderson, Georgios Fainekos, Bardh Hoxha, Hideki Okamoto, and Danil Prokhorov. 2023. Pattern matching for perception streams. In *International Conference on Runtime Verification*. Springer, 251–270.

[6] Iro Armeni, Zhi-Yang He, Amir Zamir, Junyoung Gwak, Jitendra Malik, Martin Fischer, and Silvio Savarese. 2019. 3D Scene Graph: A Structure for Unified Semantics, 3D Space, and Camera. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 5663–5672. https://doi.org/10.1109/ICCV.2019.00576

[7] NTS Board. 2019. *Collision between vehicle controlled by developmental automated driving system and pedestrian. Nat. Transpot. Saf. Board, Washington, DC.* Technical Report. USA, Tech. Rep. HAR-19-03, 2019. URL https://www. ntsb. gov/investigations ....

[8] Neal E Boudette and Niraj Chokshi. 2021. U.S. Will Investigate Tesla's Autopilot System Over Crashes With Emergency Vehicles. *New York Times* (Aug 2021). https://www.nytimes.com/2021/08/16/business/tesla-autopilot-nhtsa.html

[9] CarlaSimulator. [n. d.]. CARLA Leaderboard. https://leaderboard.carla.org/#leaderboard-10. Accessed: 2024-07-19.

[10] CarlaSimulator. [n. d.]. CARLA Leaderboard 2.0. https://leaderboard.carla.org/. Accessed: 2024-09-09.

[11] Andrea Censi, Konstantin Slutsky, Tichakorn Wongpiromsarn, Dmitry Yershov, Scott Pendleton, James Fu, and Emilio Frazzoli. 2019. Liability, Ethics, and Culture-Aware Behavior Specification using Rulebooks. In *2019 International Conference on Robotics and Automation (ICRA)*. 8536–8542. https://doi.org/10.1109/ICRA.2019.8794364

[12] Xiaojun Chang, Pengzhen Ren, Pengfei Xu, Zhihui Li, Xiaojiang Chen, and Alexander G. Hauptmann. 2021. A Comprehensive Survey of Scene Graphs: Generation and Application. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), 1–1. https://doi.org/10.1109/TPAMI.2021.3137605

[13] Dian Chen and Philipp Krähenbühl. 2022. Learning from all vehicles. In *CVPR*.

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer, 154–169.

[15] Anne Collin, Artur Bilka, Scott Pendleton, and Radboud Duintjer Tebbens. 2020. Safety of the Intended Driving Behavior Using Rulebooks. In *2020 IEEE Intelligent Vehicles Symposium (IV)*. 136–143. https://doi.org/10.1109/IV47402.2020.9304588

[16] Christoph Czepa and Uwe Zdun. 2018. On the understandability of temporal properties formalized in linear temporal logic, property specification patterns and event processing language. *IEEE Transactions on Software Engineering* 46, 1 (2018), 100–112.

[17] Bo Dai, Yuqi Zhang, and Dahua Lin. 2017. Detecting Visual Relationships with Deep Relational Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 3298–3308. https://doi.org/10.1109/CVPR.2017.352

[18] Giuseppe De Giacomo, Moshe Y Vardi, et al. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces.. In *Ijcai*, Vol. 13. 854–860.

[19] Ankush Desai, Tommaso Dreossi, and Sanjit A Seshia. 2017. Combining model checking and runtime verification for safe robotics. In *International Conference on Runtime Verification*. Springer, 172–189.

[20] Deepak Kumar Dewangan and Satya Prakash Sahu. 2020. Real time object tracking for intelligent vehicle. In *2020 first international conference on power, control and computing technologies (ICPC2T)*. IEEE, 134–138.

[21] Greg Dietrerich. 2023. Further update on emergency vehicle collision. https://www.getcruise.com/news/blog/2023/further-update-on-emergency-vehicle-collision/ Accessed on 05.05.2024.

[22] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. *CoRR* abs/1711.03938 (2017). arXiv:1711.03938 http://arxiv.org/abs/1711.03938

[23] Matthew B Dwyer, George S Avrunin, and James C Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*. 411–420.

[24] Klemens Esterle, Luis Gressenbuch, and Alois Knoll. 2020. Formalizing Traffic Rules for Machine Interpretability. In *2020 IEEE 3rd Connected and Automated Vehicles Symposium (CAVS)*. 1–7. https://doi.org/10.1109/CAVS51000.2020.9334599

[25] Alec Farid, Sushant Veer, Boris Ivanovic, Karen Leung, and Marco Pavone. 2023. Task-relevant failure detection for trajectory predictors in autonomous vehicles. In *Conference on Robot Learning*. PMLR, 1959–1969.

[26] Francesco Fuggitti. 2019. *LTLf2DFA*. https://doi.org/10.5281/zenodo.3888410

[27] Lizhao Gao, Bo Wang, and Wenmin Wang. 2018. Image Captioning with Scene-graph Based Semantic Concepts. In *Proceedings of the 2018 10th International Conference on Machine Learning and Computing* (Macau, China) *(ICMLC '18)*. Association for Computing Machinery, New York, NY, USA, 225–229. https://doi.org/10.1145/3195106.3195114

[28] Carl Hildebrandt, Trey Woodlief, and Sebastian Elbaum. 2024. ODD-diLLMma: Driving Automation System ODD Compliance Checking using LLMs. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 13809–13816.

[29] Hengle Jiang, Sebastian Elbaum, and Carrick Detweiler. 2013. Reducing failure rates of robotic systems though inferred invariants monitoring. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 1899–1906.

[30] Justin Johnson, Agrim Gupta, and Li Fei-Fei. 2018. Image Generation from Scene Graphs. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1219–1228. https://doi.org/10.1109/CVPR.2018.00133

[31] Jaewon Jung and Jongyoul Park. 2018. Visual Relationship Detection with Language prior and Softmax. In *2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS)*. 143–148. https://doi.org/10.1109/IPAS.2018.8708855

[32] Nidhi Kalra and Susan M. Paddock. 2016. *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* RAND Corporation, Santa Monica, CA. https://doi.org/10.7249/RR1478

[33] Ue-Hwan Kim, Jin-Man Park, Taek-jin Song, and Jong-Hwan Kim. 2020. 3-D Scene Graph: A Sparse and Semantic Representation of Physical Environments for Intelligent Agents. *IEEE Transactions on Cybernetics* 50, 12 (2020), 4921–4933. https://doi.org/10.1109/TCYB.2019.2931042

[34] Sangeeth Kochanthara, Tajinder Singh, Alexandru Forrai, and Loek Cleophas. 2024. Safety of Perception Systems for Automated Driving: A Case Study on Apollo. *ACM Trans. Softw. Eng. Methodol.* 33, 3, Article 64 (mar 2024), 28 pages. https://doi.org/10.1145/3631969

[35] Bettina Könighofer, Julian Rudolf, Alexander Palmisano, Martin Tappler, and Roderick Bloem. 2022. Online shielding for reinforcement learning. *Innovations in Systems and Software Engineering* (2022), 1–16.

[36] Hongsheng Li, Guangming Zhu, Liang Zhang, Youliang Jiang, Yixuan Dang, Haoran Hou, Peiyi Shen, Xia Zhao, Syed Afaq Ali Shah, and Mohammed Bennamoun. 2024. Scene Graph Generation: A comprehensive survey. *Neurocomput.* 566, C (mar 2024), 25 pages. https://doi.org/10.1016/j.neucom.2023.127052

[37] Jiachen Li, Haiming Gang, Hengbo Ma, Masayoshi Tomizuka, and Chiho Choi. 2022. Important Object Identification with Semi-Supervised Learning for Autonomous Driving. In *2022 International Conference on Robotics and Automation (ICRA)* (Philadelphia, PA, USA). IEEE Press, 2913–2919. https://doi.org/10.1109/ICRA46639.2022.9812234

[38] Yikang Li, Wanli Ouyang, Xiaogang Wang, and Xiao'Ou Tang. 2017. ViP-CNN: Visual Phrase Guided Convolutional Neural Network. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 7244–7253. https://doi.org/10.1109/CVPR.2017.766

[39] Yikang Li, Wanli Ouyang, Bolei Zhou, Kun Wang, and Xiaogang Wang. 2017. Scene Graph Generation from Objects, Phrases and Region Captions. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 1270–1279. https://doi.org/10.1109/ICCV.2017.142

[40] Wentong Liao, Bodo Rosenhahn, Ling Shuai, and Michael Ying Yang. 2019. Natural Language Guided Visual Relationship Detection. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 444–453. https://doi.org/10.1109/CVPRW.2019.00058

[41] Hengyue Liu, Ning Yan, Masood Mortazavi, and Bir Bhanu. 2021. Fully Convolutional Scene Graph Generation. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 11541–11551. https://doi.org/10.1109/CVPR46437.2021.01138

[42] Aaron Lohner, Francesco Compagno, Jonathan Francis, and Alessandro Oltramari. 2024. Enhancing Vision-Language Models with Scene Graphs for Traffic Accident Understanding. arXiv:2407.05910 [cs.CV] https://arxiv.org/abs/2407.05910

[43] Chenxia Luo, Rui Wang, Yu Jiang, Kang Yang, Yong Guan, Xiaojuan Li, and Zhiping Shi. 2018. Runtime verification of robots collision avoidance case study. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 204–212.

[44] Arnav Vaibhav Malawade, Shih-Yuan Yu, Brandon Hsu, Harsimrat Kaeley, Anurag Karra, and Mohammad Abdullah Al Faruque. 2022. Roadscene2vec: A Tool for Extracting and Embedding Road Scene-Graphs. *Know.-Based Syst.* 242, C (apr 2022), 12 pages. https://doi.org/10.1016/j.knosys.2022.108245

[45] Arnav Vaibhav Malawade, Shih-Yuan Yu, Brandon Hsu, Deepan Muthirayan, Pramod P. Khargonekar, and Mohammad Abdullah Al Faruque. 2022. Spatiotemporal Scene-Graph Embedding for Autonomous Vehicle Collision Prediction. *IEEE Internet of Things Journal* 9, 12 (2022), 9379–9388. https://doi.org/10.1109/JIOT.2022.3141044

[46] Aarian Marshall. 2018. Uber video shows the kind of crash self-driving cars are made to avoid. https://www.wired.com/story/uber-self-driving-crash-video-arizona/

[47] André Matos Pedro, Tomás Silva, Tiago Sequeira, João Lourenço, João Costa Seco, and Carla Ferreira. 2024. Monitoring of spatio-temporal properties with nonlinear SAT solvers. *Int. J. Softw. Tools Technol. Transf.* 26, 2 (feb 2024), 169–188. https://doi.org/10.1007/s10009-024-00740-7

[48] Noushin Mehdipour, Matthias Althoff, Radboud Duintjer Tebbens, and Calin Belta. 2023. Formal methods to comply with rules of the road in autonomous driving: State of the art and grand challenges. *Automatica* 152 (2023), 110692.

https://doi.org/10.1016/j.automatica.2022.110692

[49] Anton Milan, Laura Leal-Taixé, Ian Reid, Stefan Roth, and Konrad Schindler. 2016. MOT16: A benchmark for multi-object tracking. *arXiv preprint arXiv:1603.00831* (2016).

[50] Christopher Morse, Lu Feng, Matthew Dwyer, and Sebastian Elbaum. 2023. 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)*. 901–908. https://doi.org/10.1109/ICRA48891.2023.10161071

[51] Office of Public Affairs. 2024. Autonomous Vehicle Permit Holders Report A Record 9 Million Test Miles In California In 12 Months. https://www.dmv.ca.gov/portal/news-and-media/news-releases/autonomous-vehicle-permit-holders-report-a-record-9-million-test-miles-in-california-in-12-months/.

[52] PapersWithCode. 2023. Panoptic Scene Graph Generation on PSG Dataset. https://paperswithcode.com/sota/panoptic-scene-graph-generation-on-psg Accessed on 08.20.2024.

[53] PapersWithCode. 2023. Scene Graph Generation on Visual Genome. https://paperswithcode.com/sota/scene-graph-generation-on-visual-genome?metric=mean%20Recall%20%4020 Accessed on 08.20.2024.

[54] Srinivas Pinisetty, Partha S Roop, Steven Smyth, Nathan Allen, Stavros Tripakis, and Reinhard Von Hanxleden. 2017. Runtime enforcement of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 1–25.

[55] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 46–57. https://doi.org/10.1109/SFCS.1977.32

[56] Aayush Prakash, Shoubhik Debnath, Jean-Francois Lafleche, Eric Cameracci, Gavriel State, Stan Birchfield, and Marc T. Law. 2021. Self-Supervised Real-to-Sim Scene Generation. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. 16024–16034. https://doi.org/10.1109/ICCV48922.2021.01574

[57] Thomas Reinbacher, Matthias Függer, and Jörg Brauer. 2014. Runtime verification of embedded real-time systems. *Formal methods in system design* 44 (2014), 203–239.

[58] Abhirup Roy and Hyunjoo Jin. 2023. California regulator probes crashes involving GM's Cruise robotaxis. https://www.reuters.com/business/autos-transportation/gms-cruise-robotaxi-collides-with-fire-truck-san-francisco-2023-08-19/

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

[60] Hao Shao, Letian Wang, Ruobing Chen, Hongsheng Li, and Yu Liu. 2023. Safety-enhanced autonomous driving using interpretable sensor fusion transformer. In *Conference on Robot Learning*. PMLR, 726–737.

[61] Guibao Shen, Luozhou Wang, Jiantao Lin, Wenhang Ge, Chaozhe Zhang, Xin Tao, Yuan Zhang, Pengfei Wan, Zhongyuan Wang, Guangyong Chen, Yijun Li, and Ying-Cong Chen. 2024. SG-Adapter: Enhancing Text-to-Image Generation with Scene Graph Guidance. arXiv:2405.15321 [cs.CV]

[62] Joseph Stamenkovich, Lakshman Maalolan, and Cameron Patterson. 2019. Formal assurances for autonomous systems without verifying application software. In *2019 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED UAS)*. IEEE, 60–69.

[63] Yang Sun, Christopher M Poskitt, Jun Sun, Yuqi Chen, and Zijiang Yang. 2022. LawBreaker: An approach for specifying traffic laws and fuzzing autonomous vehicles. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[64] Yang Sun, Christopher M Poskitt, Xiaodong Zhang, and Jun Sun. 2024. REDriver: Runtime Enforcement for Autonomous Vehicles. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[65] Brad Templeton. 2020. Tesla In Taiwan Crashes Directly Into Overturned Truck, Ignores Pedestrian, With Autopilot On. *Forbes* (Jun 2020). https://www.forbes.com/sites/bradtempleton/2020/06/02/tesla-in-taiwan-crashes-directly-into-overturned-truck-ignores-pedestrian-with-autopilot-on/?sh=20a7458f58e5link

[66] Felipe Toledo, Trey Woodlief, Sebastian Elbaum, and Matthew B. Dwyer. 2024. Specifying and Monitoring Safe Driving Properties with Scene Graphs. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*. 15577–15584. https://doi.org/10.1109/ICRA57147.2024.10610973

[67] Ao Wang, Hui Chen, Lihao Liu, Kai Chen, Zijia Lin, Jungong Han, and Guiguang Ding. 2024. YOLOv10: Real-Time End-to-End Object Detection. *arXiv preprint arXiv:2405.14458* (2024).

[68] Hongbo Wang, Jiaying Hou, and Na Chen. 2019. A survey of vehicle re-identification based on deep learning. *IEEE Access* 7 (2019), 172443–172469.

[69] Huihui Wu, Deyun Lyu, Yanan Zhang, Gang Hou, Masahiko Watanabe, Jie Wang, and Weiqiang Kong. 2022. A verification framework for behavioral safety of self-driving cars. *IET Intelligent Transport Systems* 16, 5 (2022), 630–647.

[70] Penghao Wu, Xiaosong Jia, Li Chen, Junchi Yan, Hongyang Li, and Yu Qiao. 2022. 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.

[71] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. 2019. Detectron2. https://github.com/facebookresearch/detectron2.

[72] Danfei Xu, Yuke Zhu, Christopher B. Choy, and Li Fei-Fei. 2017. Scene Graph Generation by Iterative Message Passing. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 3097–3106. https://doi.org/10.1109/CVPR.2017.330

[73] Zhuoqian Yang, Zengchang Qin, Jing Yu, and Tao Wan. 2020. Prior Visual Relationship Reasoning For Visual Question Answering. In *2020 IEEE International Conference on Image Processing (ICIP)*. 1411–1415. https://doi.org/10.1109/ICIP40778.2020.9190771

[74] Mang Ye, Jianbing Shen, Gaojie Lin, Tao Xiang, Ling Shao, and Steven CH Hoi. 2021. Deep learning for person re-identification: A survey and outlook. *IEEE transactions on pattern analysis and machine intelligence* 44, 6 (2021), 2872–2893.

[75] Ruichi Yu, Ang Li, Vlad I. Morariu, and Larry S. Davis. 2017. Visual Relationship Detection with Internal and External Linguistic Knowledge Distillation. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 1068–1076. https://doi.org/10.1109/ICCV.2017.121

[76] Eleni Zapridou, Ezio Bartocci, and Panagiotis Katsaros. 2020. Runtime verification of autonomous driving systems in CARLA. In *International Conference on Runtime Verification*. Springer, 172–183.

[77] Qingzhao Zhang, David Ke Hong, Ze Zhang, Qi Alfred Chen, Scott Mahlke, and Z. Morley Mao. 2021. A Systematic Framework to Identify Violations of Scenario-dependent Driving Rules in Autonomous Vehicle Software. *Proc. ACM Meas. Anal. Comput. Syst.* 5, 2, Article 15 (jun 2021), 25 pages. https://doi.org/10.1145/3460082

[78] Shufang Zhu, Geguang Pu, and Moshe Y Vardi. [n. d.]. First-Order vs. Second-Order Encodings for LTLf-to-Automata. ([n. d.]).