4-2024

REDriver: Runtime enforcement for autonomous vehicles

Yang SUN
Singapore Management University, yangsun.2020@phdcs.smu.edu.sg

Christopher M. POSKITT
Singapore Management University, cposkitt@smu.edu.sg

Xiaodong ZHANG

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Citation

REDriver: Runtime Enforcement for Autonomous Vehicles

Yang Sun
Singapore Management University
Singapore
yangsun.2020@phdcs.smu.edu.sg

Christopher M. Poskitt
Singapore Management University
Singapore
cposkitt@smu.edu.sg

Xiaodong Zhang
Xidian University
China
zhangxiaodong@xidian.edu.cn

Jun Sun
Singapore Management University
Singapore
junsun@smu.edu.sg

ABSTRACT

Autonomous driving systems (ADSs) integrate sensing, perception, drive control, and several other critical tasks in autonomous vehicles, motivating research into techniques for assessing their safety. While there are several approaches for testing and analysing them in high-fidelity simulators, ADSs may still encounter additional critical scenarios beyond those covered once they are deployed on real roads. An additional level of confidence can be established by monitoring and enforcing critical properties when the ADS is running. Existing work, however, is only able to monitor simple safety properties (e.g., avoidance of collisions) and is limited to blunt enforcement mechanisms such as hitting the emergency brakes. In this work, we propose REDriver, a general and modular approach to runtime enforcement, in which users can specify a broad range of properties (e.g., national traffic laws) in a specification language based on signal temporal logic (STL). REDriver monitors the planned trajectory of the ADS based on a quantitative semantics of STL, and uses a gradient-driven algorithm to repair the trajectory when a violation of the specification is likely. We implemented REDriver for two versions of Apollo (i.e., a popular ADS), and subjected it to a benchmark of violations of Chinese traffic laws. The results show that REDriver significantly improves Apollo's conformance to the specification with minimal overhead.

1 INTRODUCTION

Autonomous driving systems (ADSs) are the core of autonomous vehicles (AVs), integrating sensing, perception, drive control, and several other tasks that are necessary for automating their journeys. Given the safety-critical nature of ADSs [14, 18], it is imperative that they operate safely at all times, including in rare or unexpected scenarios that may not have been explicitly considered when the

system was designed. This has spurred a multitude of research into techniques for establishing confidence in an ADS, e.g., by modelling and verifying aspects of its design [23], by subjecting it to reconstructions of real-world accidents [6], or by testing it against automatically generated critical scenarios [27, 44, 50] in a high-fidelity simulator such as CARLA [15] or LGSVL [38].

These approaches all analyse an ADS *before* it is deployed on real roads, where it may still encounter additional scenarios beyond those that were covered. In fact, an analysis of accidents involving autonomous vehicles [31] suggests that the broader implementation of current AV technologies may not lead to a reduction in vehicle crash frequency. An additional level of confidence can thus be established if desirable properties are also monitored—even enforced—while the ADS is running. This is the idea of *runtime enforcement*, a technique that observes the execution of a system and then modifies it in a minimal way to ensure certain properties are satisfied. In AVs, runtime enforcement has been applied, for example, to monitor basic safety properties such as the avoidance of collisions, applying the emergency brake before they are violated [21]. Avoiding collisions, however, is not enough in general. ADSs are expected to satisfy a broader range of complicated properties concerning the overall traffic systems they operate in, such as national traffic laws that describe how vehicles should behave with respect to various junctions, signals, and (most precariously) other vehicles or pedestrians. Currently, no existing approach supports runtime enforcement of properties in this direction.

In this work, we aim to provide a general solution to the runtime enforcement problem for AVs. In particular, we propose REDriver, a general framework for runtime enforcement that can be integrated into ADSs with state-of-the-art modular designs, as exhibited by Apollo [4] and Autoware [2]. REDriver allows users to specify desirable properties of AVs using an existing and powerful domain-specific language (DSL) based on signal temporal logic (STL). This language supports properties ranging from the simplest, concerning collision avoidance, through to entire formalisations of national traffic laws [44]. REDriver monitors the planned trajectories and command sequences of the ADS at runtime and assesses them against the user's specifications. If the AV is predicted to potentially violate them in the near future (based on a quantitative semantics of STL), REDriver repairs the trajectories using a gradient-driven algorithm. Furthermore, it does so while minimising the "overhead" (or change) to the original journey. That is, by efficiently computing the gradient of each signal (with respect to the robustness degree
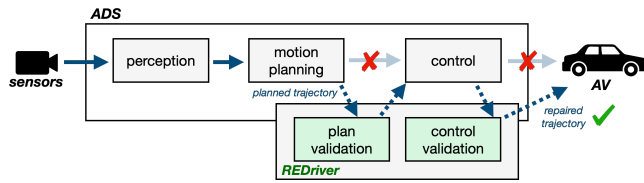
Figure 1: The architecture of an ADS with `REDriver`

Table 1: An example planned trajectory

Time	Position	Speed	Acc	Steer	Gear
0	(x: 0, y: 0)	7.01	-0.05	0	DRIVE
2	(x: 0, y: 13.34)	6.13	-0.48	0	DRIVE
4	(x: 0, y: 24.83)	5.44	-0.24	0	DRIVE
6	(x: 0, y: 35.85)	5.09	-0.18	0	DRIVE
8	(x: 0, y: 44.75)	3.89	-1.44	0	DRIVE

of the STL formula), we identify and modify the signal that is most likely to repair the trajectories.

`REDriver` has been implemented for two versions of Apollo (i.e., versions 6.0 and 7.0, the latest at the time of experimentation). The implementation consists of a *plan validation* algorithm and a *control validation* algorithm that respectively observe and modify (if necessary) the outputs of the ADSs' motion planning and control modules. Note that the motion planning and control modules are black boxes to us. In particular, we enforce that these outputs (i.e., planned trajectories and command sequences) do not lead to violations—whenever possible—of a comprehensive formalisation of the Chinese traffic laws. This goes far beyond existing runtime enforcement approaches, which focus on simple safety properties (e.g., collision avoidance) and blunt enforcement mechanisms (e.g., hitting the emergency brakes). Figure 1 depicts how `REDriver` is integrated into the modular design of Apollo. In particular, we have added two new modules while ensuring that the existing modules and their inner logic remain unchanged. In the diagram, the perception, motion planning, and control boxes represent the existing Apollo modules, while the green plan validation and control validation boxes represent the new modules from `REDriver`. The arrow denotes the flow of signal transmission. We evaluated our implementation of `REDriver` against a benchmark of violation-inducing scenarios for Chinese traffic laws [44], finding that our runtime enforcement approach significantly reduces the likelihood of those violations occurring. Furthermore, `REDriver`'s overhead in terms of time and how often it intervenes is negligible.

2 BACKGROUND AND PROBLEM

In this section, we review the architecture of ADSs, the DSL for specifying safety properties, and then define our problem.

2.1 Overview of Autonomous Driving Systems

State-of-the-art open-source ADSs such as Apollo [3] and Autoware [2] have similar architectures. They are typically organised into loosely coupled modules that communicate via message-passing. Three of these modules are particularly relevant to our context, i.e., perception, motion planning, and control.

First, the perception module receives sensor readings (e.g., from a camera or LIDAR), processes them, and then feeds them to the

Table 2: An example predicted environment

Type	Time	Position	Speed	Acc	Steer
Car1	0	(x: 2.5, y: 5)	7.42	-0.05	-7.25
	2	(x: 1.67, y: 18.34)	6.37	-0.48	-11.10
	4	(x: 0, y: 29.88)	5.44	-0.24	0
	6	(x: 0, y: 40.87)	5.09	-0.18	0
	8	(x: 0, y: 49.76)	3.89	-1.44	0
Car2	0	(x: -2.5, y: 15)	0	0	0
	2	(x: -2.5, y: 15)	0	0	0
	4	(x: -2.5, y: 15)	0	0	0
	6	(x: -2.5, y: 15)	0	0	0
	8	(x: -2.5, y: 15)	0	0	0
Ped1	0	(x: 0.23, y: 48)	0	0	0
	2	(x: 0.23, y: 48)	0	0	0
	4	(x: 0.23, y: 48)	0	0	0
	6	(x: 0.23, y: 48)	0	0	0
	8	(x: 0.23, y: 48)	0	0	0
TL-ID	Time	Color	Blink	–	–
TL-0	0	GREEN	False	–	–
	2	YELLOW	False	–	–
	4	YELLOW	False	–	–
	6	YELLOW	False	–	–
	8	RED	False	–	–

motion planning module. Second, the motion planning module generates a *planned trajectory* based on the map, the destination, the sensor inputs, and the state of the ego vehicle, i.e., the one under the control of the ADS. Intuitively, the planned trajectory describes where the vehicle will be at future time points, and is computed based on a predicted environment that includes, for example, the predicted trajectories of other vehicles (NPCs, non-player characters), pedestrians, and traffic lights. For instance, Table 1 shows a planned trajectory for an ego vehicle with respect to the predicted environment shown in Table 2. Here, the ego vehicle slows down before approaching an intersection as the traffic light is changing to red. Every line in Table 1 represents a planned *waypoint*, i.e., the planned position, speed, acceleration, steer, and gear of the ego vehicle at a series of future time points. Note that an actual planned trajectory typically contains hundreds of waypoints. Similarly, every line in Table 2 corresponds to the predicted states of NPCs such as vehicles and pedestrians, as well as environmental parameters like traffic lights. Here, Car2 and Ped1 are predicted to be stationary, Car1 is predicated to change lanes, and the color of the traffic light ahead is predicted to change from green to yellow and eventually to red. Furthermore, in general there may be multiple planned trajectories for a given destination, and the planning module attempts to find the "best" one. Finally, the control module translates the planned trajectory into control commands (e.g., 'brake', and 'signal') so that the ego vehicle is likely to follow the planned trajectory, i.e., passing through the waypoints with the planned speed, acceleration, steering angle, and gear position. We refer to [2, 3] for details on how commands are generated.

There may be other modules in an ADS (e.g., the map module in Apollo) or the above-mentioned modules may be further divided into sub-modules (e.g., motion planning in Apollo is divided into routing, prediction, and planning). Nonetheless, the similar high-level design of existing ADSs implies we could potentially introduce

$$\varphi := \mu \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \cup_I \varphi_2$$

$$\mu := f(x_0, x_1, \cdots, x_k) \sim 0 \quad \sim :=> \mid \geq \mid < \mid \leq \mid \neq \mid =;$$

Figure 2: Specification language syntax, where φ, φ_1 and φ_2 are STL formulas, I is an interval, and f is a multivariate linear continuous function over language variables x_i

a module for runtime monitoring and enforcement which sits in-between existing modules, i.e., to intercept, analyse, and alter (if necessary) the inter-module messages. This way, runtime monitoring and enforcement can be introduced without changing the inner logic of existing modules. For instance, given the planned trajectory generated by the planning module shown in Table 1, if we decide that the trajectory could potentially lead to the violation of a certain property, we can simply modify the planned trajectory before forwarding it to the control module (to trigger a different control command generation).

2.2 Property Specification

To go beyond the simplest safety requirements (e.g., 'the ego vehicle does not collide'), we require a specification language that is able to express a rich set of properties that are relevant to autonomous vehicles and driving in general. In this work, we adopt the driver-oriented specification language of LawBreaker [44], which is based on signal temporal logic (STL), and has been demonstrated to be expressive enough to specify the traffic laws of China and Singapore. We highlight its key features, referring readers to [44] for details.

The high-level syntax of the language is shown in Figure 2. A time interval I is of the form $[l, u]$ where l and u are respectively the lower and upper bounds of the interval. Following convention, we write $\diamondsuit_I \varphi$ to denote $true \cup_I \varphi$; and $\square_I \varphi$ to denote $\neg \diamondsuit_I \neg\varphi$. Intuitively, $\cup$, $\square$, and $\diamondsuit$ are modal operators that are respectively interpreted as 'until', 'always', and 'eventually'. Note that the time interval is omitted when it is $[0, \infty]$. The propositions in this language are constructed using 17 variables and 16 functions that are relevant to AVs, some of which are shown in Tables 3. In general, μ can be regarded as a proposition of the form $f(x_0, x_1, \cdots, x_k) \sim 0$ where f is a multivariate linear continuous function and x_i for all i in $[0, k]$ is a variable supported in the language.

EXAMPLE 2.1. *Consider the following two (English translations) of) traffic rules from the Regulations for Road Traffic Safety of the People's Republic of China [9].*

(1) Article #38-(3): When a red light is on, vehicles are prohibited from passing. However, vehicles turning right can pass without hindering the passage of vehicles or pedestrians.

(2) Article #58-(3): When a vehicle is driving on a foggy day, the fog lights and hazard warning flashing should be on.

Table 3: Car and environment related variables

Signal	Type	Remarks
speed	Num	Speed of ego vehicle (m/s).
acc	Num	Acceleration of ego veh (m/s^2).
direction	Enum	forward, left, right
D(stopline)	Num	distance to the stopline ahead
D(junction)	Num	distance to the junction ahead
fogLight	Bool	whether the fog light is on
warningFlash	Bool	whether the warning flash light is on
PriorityV(n)	Bool	Whether there are vehicles with priority within n meters
PriorityP(n)	Bool	Whether there are pedestrians with priority within n meters
TL(color)	Enum	YELLOW, GREEN, RED, or BLACK
TL(blink)	Bool	if the traffic light ahead is blinking
fog	Num	degree of fog ranging from 0 to 1
snow	Num	degree of snow ranging from 0 to 1

The above can be formalised as follows.

$$law38_3 \equiv \square((TL(color) = red$$
$$\wedge (D(stopline) < 2 \vee D(junction) < 2)$$
$$\wedge \neg direction = right) \rightarrow (\diamondsuit_{[0,3]}(speed < 0.5))$$
$$\wedge (TL(color) = red \wedge (D(stopline) < 2$$
$$\vee D(junction) < 2) \wedge direction = right$$
$$\wedge \neg PriorityV(20) \wedge \neg PriorityP(20))$$
$$\rightarrow (\diamondsuit_{[0,2]}(speed > 0.5)))$$
$$law58_3 \equiv \square(fog \geq 0.5 \rightarrow (fogLight \wedge warningFlash))$$

where speed, direction, fogLight, *and* warningFlash *represent the speed, direction, fog light status, and warning flash light status of the vehicle;* TL() *returns the status of traffic light ahead;* D(object) *calculates the distance from the vehicle to the* object *ahead; and* PriorityV(n), PriorityP(n) *check whether there is a priority vehicle or pedestrian within* n *meters ahead. Note that several configurable constants (e.g., the distance 2 and the time interval $[0, 3]$) are introduced to reduce the vagueness of the law in practice [44].* □

A specification is evaluated with respect to a trace π of *scenes*, denoted as $\pi = \langle \pi_0, \pi_1, \pi_2 \ldots, \pi_n \rangle$, where each scene π_i is a valuation of the propositions at time step i and π_0 reflects the state at the start of a simulation. These traces can be constructed from the planned trajectory generated by the ADS (Section 3.1). We follow the standard semantics of STL (see e.g., [29]).

2.3 The Runtime Enforcement Problem for AVs

Given an ADS and a user-specified property φ, our goal is to solve the *runtime enforcement* problem for AVs by monitoring traces π of the ADS against φ at runtime, and altering its behavior when a violation is likely in the near future. Here, altering the ADS's behavior means adjusting its planned trajectory and consequently the control commands. Solving this problem could systematically improve the safety of ADSs when encountering unusual situations on the road. We formulate our problem as follows:

DEFINITION 1 (PROBLEM DEFINITION). *Given a runtime planned trajectory γ, runtime control commands ζ, a specification of ADS behavior φ, and a trace π of the AV in a scenario. Let γ', ζ', and π' denote the adjusted planned trajectory, modified control commands,*

and resulting trace of the AV after these adjustments. Our problem is:

$$Maximise : \frac{\rho(\varphi, \pi') - \rho(\varphi, \pi)}{|\gamma' - \gamma| + |\zeta' - \zeta|}.$$

□

Intuitively, we seek to maximize the improvement in adhering to the desired behavior, while considering the magnitude of changes made to the planned trajectory and control commands. Here, the function ρ serves as the quantitative semantics of a trace concerning the specification. Its purpose is to provide a numerical assessment that calculates the distance to a violation of the specification.

3 OUR APPROACH

REDriver, our runtime enforcement approach, consists of three broad steps. First, *plan validation*, in which it evaluates the planned trajectory against the specification to determine if there is a risk of violation. Second, *trajectory repair*, in which the planned trajectory is modified so as to avoid the violation. Finally, *control validation*, in which the commands generated by the control module are further evaluated to ensure the specification is satisfied. As shown in Figure 1, these steps seamlessly integrate into the modular design of ADSs: REDriver sits between the modules, intercepting and altering the messages they exchange. Note that we assume that the sensor data received by the ADS is accurate.

3.1 Plan Validation

Given a specification φ and a planned trajectory from the motion planning module of the ADS, REDriver first determines whether the trajectory is likely to violate φ. To achieve this, REDriver first constructs a trace π from the planned trajectory, i.e., by evaluating all variables and functions relevant to φ at every time point with respect to the planned trajectory and the predicted environment. For instance, given the planned trajectory in Table 1 (and the predicted environment in Table 2), Table 4 shows the constructed trace.

One practical complication is that some variables relevant to φ cannot be obtained from the planned trajectory as they are only known after command generation (see Section 3.3). For example, the values of fogLight (i.e., whether the fog light is on) and warningFlash can only be determined once the respective commands are generated. For such situations, we use typed 'placeholder' variables $x_{i,j}$ in the scenes of the trace for each time step i and position j. We define an *assignment* α to be a function mapping the typed variables $x_{i,j}$ to the value domains. Then, for traces π containing those variables, π satisfies φ if and only if there exists an assignment α such that $\pi[\alpha(x_{i,j})/x_{i,j}]$ satisfies φ for every variable $x_{i,j}$ in π. Practically, finding a suitable assignment α is straightforward: all variables for assignment α have only a few possible discrete values (e.g., the light is on or off), and thus brute force search is sufficient and inexpensive.

Next, REDriver computes how 'close' the ego vehicle will come to violating φ. Note that our goal is to proactively react when a violation is likely in the near future. This is because the ego vehicle operates in an open environment (e.g., with other vehicles and pedestrians) and thus reacting too late may be too risky if the predicted environment turns out to be wrong (e.g., a sudden move of a pedestrian). To measure how close a trace π is to violating

Table 4: Trace obtained from the trajectory in Table 1

planning signals	0	2	4	6	8
speed	7.01	6.13	5.44	5.09	3.89
direction	0	0	0	0	0
D(stopline)	44	30.66	19.17	8.15	-0.75
D(junction)	44	30.66	19.17	8.15	-0.75
fogLight	$x_{0,0}$	$x_{2,0}$	$x_{4,0}$	$x_{6,0}$	$x_{8,0}$
warningFlash	$x_{0,1}$	$x_{2,1}$	$x_{4,1}$	$x_{6,1}$	$x_{8,1}$
Prediction Signals	0	2	4	6	8
TL(color)	1	0	0	0	2
fog	0.6	0.6	0.6	0.6	0.6
PriorityV(20)	false	false	false	false	false
PriorityP(10)	false	false	false	true	true

φ, we adopt a quantitative semantics [13, 29, 34] that produces a numerical *robustness* degree.

DEFINITION 2 (QUANTITATIVE SEMANTICS). *Given a trace π and a formula φ, the quantitative semantics is defined as the robustness degree $\rho(\varphi, \pi, t)$, computed as follows. Recall that propositions μ are of the form $f(x_0, x_1, \cdots, x_k) \sim 0$.*

$$\rho(\mu, \pi, t) = \begin{cases} -\pi_t(f(x_0, x_1, \cdots, x_k)) & if \sim is \leq or < \\ \pi_t(f(x_0, x_1, \cdots, x_k)) & if \sim is \geq or > \\ |\pi_t(f(x_0, x_1, \cdots, x_k))| & if \sim is \neq \\ -|\pi_t(f(x_0, x_1, \cdots, x_k))| & if \sim is = \end{cases}$$

where t is the time step and $\pi_t(e)$ is the valuation of expression e at time t in π.

$$\rho(\neg\varphi, \pi, t) = -\rho(\varphi, \pi, t)$$
$$\rho(\varphi_1 \wedge \varphi_2, \pi, t) = \min\{\rho(\varphi_1, \pi, t), \rho(\varphi_2, \pi, t)\}$$
$$\rho(\varphi_1 \vee \varphi_2, \pi, t) = \max\{\rho(\varphi_1, \pi, t), \rho(\varphi_2, \pi, t)\}$$
$$\rho(\varphi_1 \cup_I \varphi_2, \pi, t) = \sup_{t_1 \in t+I} \min\{\rho(\varphi_2, \pi, t_1), \inf_{t_2 \in [t, t_1]} \rho(\varphi_1, \pi, t_2)\}$$

where $t + I$ is the interval $[l + t, u + t]$ given $I = [l, u]$. □

Note that the smaller $\rho(\varphi, \pi, t)$ is, the closer π is to violating φ. If $\rho(\varphi, \pi, t) \leq 0$, φ is violated. We write $\rho(\varphi, \pi)$ to denote $\rho(\varphi, \pi, 0)$; $\pi \vDash \varphi$ to denote $\rho(\varphi, \pi, t) > 0$; and $\pi \nvDash \varphi$ to denote $\rho(\varphi, \pi, t) \leq 0$. Note that time is discrete in our setting.

EXAMPLE 3.1. *Let $\varphi = \Box(speed < 90)$, i.e., the speed limit is 90km/h. Suppose π is $\langle(speed \mapsto 0, \dots), (speed \mapsto 0.5, \dots), \cdots (speed \mapsto 85, \dots)\rangle$ where the ego vehicle's max speed is 85km/h at the last time step. We have $\rho(\varphi, \pi) = \rho(\varphi, \pi, 0) = \min_{t \in [0, |\pi|]}(90 - \pi_t(speed)) = 5$. Suppose instead that φ is the specification from Example 2.1 and π is the trace from Table 4. The robustness value is $\rho(\varphi, \pi) = 0$, i.e., φ is violated as the ego vehicle fails to stop before the stop line when the traffic light turns red.* □

3.2 Trajectory Repair

If the robustness value of φ with respect to a trace π is below a certain threshold θ, there is a risk of violating φ in the future, even if $0 < \rho(\varphi, \pi) \leq \theta$ (given that there is uncertainty in the predicted environment). This threshold is determined experimentally in our work (Section 4): intuitively, it characterises how 'cautious' the ADS is. In order to enforce φ, i.e., proactively prevent its possible violation, REDriver *repairs* the planned trajectory before sending

it to the control module of the ADS so as to change the commands that will be generated.

Our trajectory repair method consists of three steps. First, we identify the earliest time step when the robustness value falls below the threshold. Second, we compute the gradient (through auto-differentiation [22]) of each variable at the identified time step with respect to the robustness degree. Based on the result, we then modify the variable to increase the robustness degree. Finally, we modify the planned trajectory accordingly and feed it into the control module. In the following, we present each step in detail.

Determine the time step. Given a trace $\pi = \langle(t_0, s_0), \cdots, (t_n, s_n)\rangle$, we write π^k to denote the prefix $\langle(t_0, s_0), (t_1, s_1), \cdots, (t_k, s_k)\rangle$. Given π such that $\rho(\varphi, \pi) < \theta$, we aim to identify a time step k such that: (1) $\rho(\varphi, \pi^k) < \theta$; and (2) there does not exist a time step l such that $l < k$ and $\rho(\varphi, \pi^l) < \theta$. Intuitively, k is the earliest time step when the robustness value falls below the threshold. We identify the time step k using a sequential search, i.e., we start from $k = 0$ and keep increasing k until we find a k such that $\rho(\varphi, \pi^k) < \theta$.

EXAMPLE 3.2. *Let $\varphi = law38_3$ from Example 2.1 and π denote the trace from Table 4. Suppose the threshold θ is 10. Then, as shown in Example 3.1, $\rho(law38_3, \pi) = 0$ and is thus below the threshold. Then, we apply the above-mentioned algorithm to identify the time step. The following are computed in sequence.*

$$\rho(\varphi, \pi^0) = 42, \cdots, \rho(\varphi, \pi^2) = 28.66, \cdots,$$
$$\rho(\varphi, \pi^4) = 17.17, \cdots, \rho(\varphi, \pi^6) = 6.15$$

Thus, the time step k that we are looking for is 6 (as $6 < \theta$). □

Calculate the gradient. Next, we find out how the variables at time step k should be modified so that the robustness degree of the resulting trace can be improved. We thus define a differentiation function that calculates the gradient of each relevant variable with respect to the robustness degree. Intuitively, when the gradient of a variable x at time k is positive (resp. negative), we can increase the robustness degree by increasing (resp. decreasing) the value of x.

Recall that the robustness degree of φ is computed using discrete functions min and max (Definition 2) that are hard to differentiate [46]. Hence, we adopt a continuous robustness measure as defined in [20, 35] which replaces min and max in Definition 2 with continuous functions $\widetilde{max}$ and $\widetilde{min}$ as follows:

$$\widetilde{max}\{x_0, x_1, \ldots, x_m\} = \frac{1}{a}\ln(\sum_{i=1}^m e^{ax_i})$$
$$\widetilde{min}\{x_0, x_1, \ldots, x_m\} = -\widetilde{max}(-x_0, -x_1, \ldots, -x_m)$$

where a is a constant that controls the accuracy of $\widetilde{max}$ and $\widetilde{min}$. The larger a is, the closer $\widetilde{max}$ (resp. $\widetilde{min}$) is to max (resp. min). We set a to be 10, following [20, 35]. We denote the continuous robustness degree as $\tilde{\rho}(\varphi, \pi)$. The following proposition from [20, 35] establishes the soundness of approximating $\rho(\varphi, \pi)$ with $\tilde{\rho}(\varphi, \pi)$.

PROPOSITION 3.3. *Let φ be an STL formula, π be a trace, and ε be a real value larger than 0. Then, there exists a value a_1 such that $|\tilde{\rho}(\varphi, \pi, i) - \rho(\varphi, \pi, i)| < \varepsilon$ holds for all $a > a_1$.* □

Next, we define a differentiation function $D(\varphi, \pi, x^k)$ that returns a given variable x's gradient with respect to $\rho(\varphi, \pi)$ at time k.

$$D(\varphi, \pi, x^k) = \frac{\partial\tilde{\rho}(\varphi, \pi, 0)}{\partial x^k}$$

The following shows how $D(\varphi, \pi, x^k)$ is computed.

DEFINITION 3. *Given an STL formula φ and trace π, function $D(\varphi, \pi, x^k)$ is defined as follows:*

$$\frac{\partial\tilde{\rho}(\mu, \pi, t)}{\partial x^k} = \begin{cases} 0 & \text{if } k \neq t \\ \frac{df'(x_0, x_1, \cdots, x_n)}{dx^k} & \text{otherwise} \end{cases}$$

where $\frac{df'(x_0, x_1, \cdots, x_k)}{dx^k}$ is the derivative of function f' with respect to x^k. Furthermore, let $\frac{\partial\widetilde{max}(\{x_0, x_1, \ldots, x_m\})}{\partial x^k}$ be defined as $\frac{e^{ax}}{\sum_{i=1}^m e^{ax_i}}$, and let $\frac{\partial\widetilde{min}(\{x_0, x_1, \ldots, x_m\})}{\partial x^k}$ be defined as $\frac{e^{-ax}}{\sum_{i=1}^m e^{-ax_i}}$.

$$\frac{\partial\tilde{\rho}(\neg\varphi, \pi, t)}{\partial x^k} = -\frac{\partial\tilde{\rho}(\varphi, \pi, t)}{\partial x^k}$$

$$\frac{\partial\tilde{\rho}(\varphi_1 \wedge \varphi_2, \pi, t)}{\partial x^k} = \frac{\partial\widetilde{min}\{\tilde{\rho}(\varphi_1, \pi, t), \tilde{\rho}(\varphi_2, \pi, t)\}}{\partial\tilde{\rho}(\varphi_1, \pi, t)} \cdot \frac{\partial\tilde{\rho}(\varphi_1, \pi, t)}{\partial x^k}$$
$$+ \frac{\partial\widetilde{min}\{\tilde{\rho}(\varphi_1, \pi, t), \tilde{\rho}(\varphi_2, \pi, t)\}}{\partial\tilde{\rho}(\varphi_2, \pi, t)} \cdot \frac{\partial\tilde{\rho}(\varphi_2, \pi, t)}{\partial x^k}$$

$$\frac{\partial\tilde{\rho}(\varphi_1 \vee \varphi_2, \pi, t)}{\partial x^k} = \frac{\partial\widetilde{max}\{\tilde{\rho}(\varphi_1, \pi, t), \tilde{\rho}(\varphi_2, \pi, t)\}}{\partial\tilde{\rho}(\varphi_1, \pi, t)} \cdot \frac{\partial\tilde{\rho}(\varphi_1, \pi, t)}{\partial x^k}$$
$$+ \frac{\partial\widetilde{max}\{\tilde{\rho}(\varphi_1, \pi, t), \tilde{\rho}(\varphi_2, \pi, t)\}}{\partial\tilde{\rho}(\varphi_2, \pi, t)} \cdot \frac{\partial\tilde{\rho}(\varphi_2, \pi, t)}{\partial x^k}$$

$$\frac{\partial\tilde{\rho}(\varphi_1 \, \mathsf{U_I} \, \varphi_2, \pi, t)}{\partial x^k} = \sum_{t' \in t+\mathrm{I}} \left(\frac{\partial\tilde{\rho}(\varphi_1 \, \mathsf{U_I} \, \varphi_2, \pi, t)}{\partial\tilde{\rho}(\varphi_1, \pi, t')} \cdot \frac{\partial\tilde{\rho}(\varphi_1, \pi, t')}{\partial x^k} \right.$$
$$\left. + \frac{\partial\tilde{\rho}(\varphi_1 \, \mathsf{U_I} \, \varphi_2, \pi, t)}{\partial\tilde{\rho}(\varphi_2, \pi, t')} \cdot \frac{\partial\tilde{\rho}(\varphi_2, \pi, t')}{\partial x^k} \right)$$

where $\frac{\partial\tilde{\rho}(\varphi_1 \, \mathsf{U_I} \, \varphi_2, \pi, t)}{\partial\tilde{\rho}(\varphi_1, \pi, t')}$ is the derivative of $\tilde{\rho}(\varphi_1 \mathsf{U_I} \varphi_2, \pi, t)$ with respect to $\tilde{\rho}(\varphi_1, \pi, t')$, and is defined as:

$$\sum_{t_1 \in t+\mathrm{I} \wedge t_1 \geq t'} \left(\frac{\partial\widetilde{max}\{\widetilde{min}\{\tilde{\rho}(\varphi_2, \pi, t_1), \inf_{t_2 \in [t, t_1]} \tilde{\rho}(\varphi_1, \pi, t_2)\} \mid t_1 \in t+\mathrm{I}\}}{\partial\widetilde{min}\{\tilde{\rho}(\varphi_2, \pi, t_1), \inf_{t_2 \in [t, t_1]} \tilde{\rho}(\varphi_1, \pi, t_2)\}} \cdot \right.$$
$$\left. \frac{\partial\widetilde{min}\{\tilde{\rho}(\varphi_2, \pi, t_1), \inf_{t_2 \in [t, t_1]} \tilde{\rho}(\varphi_1, \pi, t_2)\}}{\partial\inf_{t_2 \in [t, t_1]} \tilde{\rho}(\varphi_1, \pi, t_2)} \cdot \frac{\partial\widetilde{min}\{\tilde{\rho}(\varphi_1, \pi, t_2) \mid t_2 \in [t, t_1]\}}{\partial\tilde{\rho}(\varphi_1, \pi, t')} \right)$$

where $\frac{\partial\tilde{\rho}(\varphi_1 \, \mathsf{U_I} \, \varphi_2, \pi, t)}{\partial\tilde{\rho}(\varphi_2, \pi, t')}$ is defined as:

$$\frac{\partial\widetilde{max}\{\widetilde{min}\{\tilde{\rho}(\varphi_2, \pi, t_1), \inf_{t_2 \in [t, t_1]} \tilde{\rho}(\varphi_1, \pi, t_2)\} \mid t_1 \in t+\mathrm{I}\}}{\partial\widetilde{min}\{\tilde{\rho}(\varphi_2, \pi, t'), \inf_{t_2 \in [t, t']} \tilde{\rho}(\varphi_1, \pi, t_2)\}} \cdot$$
$$\frac{\partial\widetilde{min}\{\tilde{\rho}(\varphi_2, \pi, t'), \inf_{t_2 \in [t, t']} \tilde{\rho}(\varphi_1, \pi, t_2)\}}{\partial\tilde{\rho}(\varphi_2, \pi, t')}$$

□

Given the time step k previously identified, we apply the above definition to compute $D(\varphi, \pi^k, x^k)$ for every variable x. The purpose of the differentiation function D is to determine the "responsibility" of each signal in violating the specification. In other words, consider the computation of robustness as a function of multiple variables where D determines the gradient of each variable. We remark that our implementation of $D(\varphi, \pi^k, x^k)$ is based on automatic differentiation techniques [22]. Intuitively, we store the intermediate values while computing the robustness degree, and then compute the gradients based on reverse accumulation.

EXAMPLE 3.4. *Given the trace π of Table 4, the following shows how to calculate the gradient of speed with respect to $\varphi_0 = \square(speed > 5)$ at time step 6:*

$$D(\varphi_0, \pi^6, speed^6) = \frac{\partial \rho(\varphi, \pi^6, 0)}{\partial \rho(speed > 5, \pi^6, 6)} \cdot \frac{\partial \rho(speed > 5, \pi^6, 6)}{\partial speed^6}$$

$$= \frac{e^{-10*0.09}}{e^{-10*2.01} + e^{-10*1.13} + e^{-10*0.44} + e^{-10*0.09}} \cdot 1 = 0.97$$

Similarly, continuing Example 3.2, the gradients are computed as follows:

$$D(law38_3, \pi^6, speed^6) = 8.39 \times 10^{-08}$$

$$D(law38_3, \pi^6, D(stopline)^6) = 0.5$$

$$D(law38_3, \pi^6, D(junction)^6) = 0.5$$

$$D(law38_3, \pi^6, direction^6) = -4.74 \times 10^{-19}$$

$$D(law38_3, \pi^6, TL(color)^6) = -9.48 \times 10^{-19}$$

$$D(law38_3, \pi^6, PriorityV(20)^6) = -9.77 \times 10^{-28}$$

$$D(law38_3, \pi^6, PriorityN(20)^6) = 2.15 \times 10^{-23}$$

The gradients for variable $D(stopline)$ and $D(junction)$ at time step 6 are positive, which means that we can effectively increase the robustness value $\rho(law38_3, \pi^6)$ by increasing $D(stopline)^6$ or $D(junction)^6$. □

PROPOSITION 3.5. *Let $D(\varphi, \pi, x^k)$ be the result of gradient calculation as shown in Definition 3. When $D(\varphi, \pi, x^k)$ is positive (or negative), there exists an interval $(0, \Delta)$ such that increasing (or decreasing) x^k within this interval increases in the value of $\tilde{\rho}(\varphi, \pi)$.*

PROOF. First, if φ is a Boolean Expression μ, $\tilde{\rho}(\mu, \pi)$ can be represented by a continuous function $f'(x_0, \cdots, x_n)$ as shown in Definition 2. Given that $f'(x_0, \cdots, x_n)$ is confined to linear or absolute value functions, the proposition holds for μ.

Then, assuming the proposition holds, the proposition holds if we can prove the proposition holds for each and every way φ can be constructed, i.e., $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, and $\varphi_1 \cup_I \varphi_2$.

If φ is in the format of $\neg\varphi_1$, we have $\tilde{\rho}(\neg\varphi_1, \pi) = -\tilde{\rho}(\varphi_1, \pi)$, and $D(\neg\varphi_1, \pi, x^k) = -D(\varphi_1, \pi, x^k)$. Thus, by negating the modification, we can ensure that the proposition holds for $\neg\varphi_1$.

If φ is in the format of $\varphi_1 \vee \varphi_2$, $\tilde{\rho}(\varphi_1 \vee \varphi_2, \pi) = \frac{1}{a} \ln(e^{ax_1} + e^{ax_2})$, and $D(\varphi_1 \vee \varphi_2, \pi, x^k) = \frac{e^{ax_1}}{e^{ax_1}+e^{ax_2}} \cdot D(\varphi_1, \pi, x^k) + \frac{e^{ax_2}}{e^{ax_1}+e^{ax_2}} \cdot D(\varphi_2, \pi, x^k)$. Here, $x_1 = \tilde{\rho}(\varphi_1, \pi)$, $x_2 = \tilde{\rho}(\varphi_2, \pi)$, $a \to \infty$. Suppose the proposition holds for $\tilde{\rho}(\varphi_1, \pi)$ within interval $(0, \Delta_1)$, and holds for $\tilde{\rho}(\varphi_2, \pi)$ within interval $(0, \Delta_2)$. If $x_1 \neq x_2$, suppose $x_1 > x_2$, then we have $\frac{e^{ax_1}}{e^{ax_1}+e^{ax_2}} \to 1$, $\frac{e^{ax_2}}{e^{ax_1}+e^{ax_2}} \to 0$, and $D(\varphi_1 \vee \varphi_2, \pi, x^k) \to D(\varphi_1, \pi, x^k)$. The proposition holds for interval $(0, \Delta_1)$. If $x_1 = x_2$, then $D(\varphi_1 \vee \varphi_2, \pi, x^k) > 0$ indicates $D(\varphi_1, \pi, x^k) + D(\varphi_2, \pi, x^k) > 0$. Even if one of $D(\varphi_1, \pi, x^k)$ and $D(\varphi_2, \pi, x^k)$ is negative, the value of $e^{ax_1} + e^{ax_2}$ still increases, leading to the increase of $\tilde{\rho}(\varphi_1 \vee \varphi_2, \pi)$. Let Δ' be a number larger than 0 and smaller than $min\{\Delta_1, \Delta_2\}$. The proposition holds for $(0, \Delta')$.

If φ is in the format of $\varphi_1 \wedge \varphi_2$, since $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, we can deduce that the proposition always holds for $\varphi_1 \wedge \varphi_2$.

If φ is in the format of $\varphi_1 \cup_I \varphi_2$. Since $\tilde{\rho}(\varphi_1 \cup_I \varphi_2, \pi)$ is a combination of the function $\widetilde{max}$ and $\widetilde{min}$, we can deduce that the proposition always holds for $\varphi_1 \cup_I \varphi_2$.

Therefore, we can conclude that the proposition holds. □

Intuitively, Proposition 3.5 clarifies that the gradient calculation function $D(\varphi, \pi, x^k)$ reflects the changing trend of the robustness

---

Algorithm 1: Trajectory repair algorithm

Input: variable/function x, time step k, magnitude δ

1 **case** x *is speed* **do**
2   Set $speed^k$ to be $speed^k + \delta$;
3 **case** x *is direction* **do**
4   Choose a value d_0 (0, 1, or 2) that is closest to $direction^k + \delta$;
5   Set the *steer* at time k to 0 if $d_0 = 0$;
6   Otherwise set the *steer* at time k to 0.1 if $d_0 = 1$;
7   Otherwise set the *steer* at time k to -0.1 if $d_0 = 2$;
8 **case** x *is of the form* $D(\_)$ *or* $Lane(\_)$ **do**
9   Search for a coordinate (a, b) (i.e., new position for the ego vehicle) such that $D(\_)$ becomes $D(\_) + \delta$ or $Lane(\_)$ becomes $Lane(\_) + \delta$;
10   Set the position of the ego vehicle at time k to (a, b);
11 **end**

---

function $\tilde{\rho}(\varphi, \pi)$ in terms of variable x^k. However, the changing trend is sensitive to the variable's current value. If we increase the variable by too much, it may lead to a decrease in robustness. For instance, consider the specification: $\varphi = 10 < speed < 100$. Suppose the current speed is 8, then the robustness is -2, and the gradient for speed $D(\varphi, \pi, speed)$ is 1, indicating that we should increase the value of speed. If we increase the speed within the interval $(0, 94)$, the robustness will always be larger than -2. However, if we increase the speed to 103, the robustness will become -3, resulting in a decrease. Therefore, to guarantee an increase in robustness, it is necessary to limit the modification within an interval of $(0, \Delta)$.

Repair the trajectory. The gradients calculated above allow us to determine how to effectively increase the robustness degree. We can proceed to repair the trace by modifying the variable with the maximal absolute gradient at time step k. The *magnitude* of the modification is calculated as follows:

$$\delta = \frac{\theta - \tilde{\rho}(\varphi, \pi^k)}{D(\varphi, \pi^k, x^k)}; \ While \ \tilde{\rho}(\varphi, \pi') < \tilde{\rho}(\varphi, \pi) \ Do: \ \{\delta \leftarrow \delta/2\}$$

where π' is the trace after the modification. This *magnitude* of the modification indicates that we try to increase the robustness value to θ. However, this adjustment might sometimes lead to overreactions, causing a decrease in the robustness value. In such cases, we reduce δ until we observe an increase in the robustness value, and the descent rate during this process follows a scale of 2^n, enabling us to efficiently determine the magnitude. For instance, according to Example 3.4, we should modify $D(stopline)$ or $D(junction)$ at time step 6 with a magnitude of $\frac{\theta - \rho(law38_3, \pi^6)}{0.5} = 7.7$. This modification results in $\rho(law38_3, \pi^6)$ increasing from 6.15 to 13.85.

PROPOSITION 3.6. *Let x^k be a variable, and δ be the* magnitude *of the modification on x^k. The robustness value $\tilde{\rho}(\varphi, \pi)$ always increases after the modification.*

PROOF. The modification is triggered only when $\theta - \tilde{\rho}(\varphi, \pi) > 0$, which implies that δ and $D(\varphi, \pi, x^k)$ share the same sign. As shown in Proposition 3.5, there exists an interval $(0, \Delta)$ in which the gradient value is effective. If the previous modification results in a decrease of $\tilde{\rho}(\varphi, \pi)$, we can ensure an increase in $\tilde{\rho}(\varphi, \pi)$ by decreasing $|\delta|$ to a value smaller than Δ. The proposition holds. □

Algorithm 2: Runtime enforcement algorithm

Input: specification φ, trajectory Γ, the threshold θ

1   Generate trace π based on Γ;

2   **if** $\rho(\varphi, \pi) \leq \theta$ **then**

3   Identify the smallest k such that $\rho(\varphi, \pi^k) \leq \theta$;

4   Compute $D(\varphi, \pi^k, x^k)$ for every controllable variable x^k;

5   Identify variable x^k with the maximal absolute gradient;

6   Invoke Algorithm 1 to fix the trajectory;

7   **end**

Recall that our goal is to modify the planned trajectory so as to trigger different control commands. While we may modify the trace arbitrarily, we cannot do the same for the planned trajectory. First, some of the variables may not be controllable, e.g., the color of the traffic light is beyond the control of the ADS. Second, a variable may have a specific domain of discrete values in the ADS (e.g., *direction* has the value of 0, 1, or 2) and thus we can only choose one of those valid values. Finally, the value of a variable may be the result of a function which depends on the current and future scenes. For instance, $D(stopline)$ measures the distance from the ego vehicle (according to the planned trajectory) to the stop line ahead (according to the map). In these situations, it is very difficult to translate the modification to the planned trajectory. Thus, we focus on modifying those signals that the ADS has control over and modify the planned trajectory accordingly, which are *speed*, *acc*, *direction*, *Lane*(_) (i.e., which lane the ego vehicle should be in), and $D(\_)$ (i.e., how far the ego vehicle is from a certain artifact). These naturally correspond to what human drivers focus on. Algorithm 1 describes how the planned trajectory is repaired with respect to a specific variable/function x, time step k, and magnitude δ. Note that the fixes are specific to certain variables since they may have specific domains. In the case of *direction*, we are constrained to choose a value from 0, 1, 2 and set the value in the trajectory accordingly. In the case of functions based on the ego vehicle's position (e.g., $D(stopline)$), we search for nearby coordinates that are close to the desired value while still remaining on the road. Note that the ADS's planning module and the control module are entirely black boxes to us. Therefore, we do not take into account the correlations between variables when modifying the planned trajectory. To do so would require the construction of an exhaustive physical model, essentially equivalent to rebuilding the planning module of the ADS.

EXAMPLE 3.7. *Given the planned trajectory in Table 1, and the repair computed for $D(stopline)$ and $D(junction)$ in Example 3.4. We modify the planned car position at time step 6 from $(0, 35.85)$ to $(0, 28.15)$ (so the vehicle should be positioned further from the junction), leaving the remaining planned trajectory unchanged. Note that changing the value of speed can rectify the trajectory as well, however, the gradient values strongly suggest that optimizing the position of the waypoint is a more efficient approach.* □

3.3 Runtime Enforcement

We are now ready to present our runtime enforcement algorithm, as shown in Algorithm 2. First, we generate a trace π based on the planned trajectory Γ and check whether $\rho(\varphi, \pi) \leq \theta$. If so, we

proceed to identify the time step k when the robustness degree falls below the threshold. Then we compute gradients for the variables at time k, identify the controllable one with the maximal absolute gradient (w.r.t. the robustness degree) and repair the trajectory accordingly. The repaired trajectory is then sent to the control module, which generates the commands accordingly (e.g., turn on/off beam, and apply brake).

Recall that the specification φ may also constrain the generated commands, e.g., the need to signal before turning. To make sure the commands generated do not violate φ, we introduce a *control validation module* (refer to Figure 1) that intercepts and checks the generated commands, modifying them if necessary. Recall that commands related to motion (e.g., brake, accelerate, steer, and gear) are generated according to the (repaired) trajectory and thus do not require modification. We remark that these commands are mostly simple in nature (i.e., with Boolean values) and thus we can easily modify them according to the specification. For instance, consider the beam-related signals, namely highBeam and lowBeam, which have *on* and *off* states. We can easily modify these states by switching the values in the control commands sent to the AV's chassis control.

EXAMPLE 3.8. *Consider the trace shown in Table 4. Recall that the signals* fogLight *and* warningFlash *are not part of the planned trajectory: in fact, the ADS turns these off by default, i.e., we initially have $\alpha(x_{i,j}) = $ false ($x_{i,j}$ is the placeholder variable as discussed in Section 3.1). To satisfy the specification in Example 2.1,* REDriver *sets $\alpha(x_{i,j}) = $ true for each i, j. To realize this, we activate the* fogLight *and* warningFlash *in the control commands.* □

4 IMPLEMENTATION AND EVALUATION

We implemented REDriver for Apollo 6.0 and 7.0 [3, 4]. The code is on our website [1]. In particular, we built a bridge program that interprets Apollo's messages (in JSON format) and obtains the values of variables and functions used by the specification language. Some of these values are obtained directly (e.g., *speed* and *acc*), but some require complex processing. For example, to get the value of variable *NPCAhead.speed* at time t, we obtain the planned position of the ego vehicle at time t from the planning module, and check every NPC vehicle's predicted trajectory from the prediction module to identify the one that is ahead of the ego vehicle at time t. Our implementation relies on a third party component provided by LawBreaker [44]. In particular, we utilise the tool's specification language and the corresponding verification algorithm.

We conducted experiments to answer the following Research Questions (RQs):

RQ1: Can REDriver be used to enforce non-trivial specifications?

RQ2: How much overhead is there for runtime enforcement?

RQ3: Does REDriver minimise the enforcement?

RQ1 considers whether REDriver achieves its primary goal of being able to enforce complex specifications (i.e., beyond collision avoidance). RQ2 and RQ3 consider whether REDriver implements its enforcement in a way that is practically reasonable. The former focuses on the overhead of runtime enforcement, since AVs are expected to react quickly on the road. The latter focuses on the

Table 5: Violations of Chinese traffic laws

traffic laws		enforced?		improve	fail reason	content
		6.0	7.0			
Law38	sub1	√	√	+55.83%	-	green light
	sub2	√	√		-	yellow light
	sub3	√	√		-	red light
Law44		√	√	+30.00%	-	lane change
Law46	sub2	√	√	+44.00%	-	speed limit
	sub3	×	×	-	Lack support	speed limit
Law47		×	×	-	Lack support	overtake
Law51	sub3	×	×	-	Lack support	traffic light
	sub4	√	√	+35.00%	-	traffic light
	sub5	√	√		-	traffic light
Law57	sub1	×	×	-	Lack support	left turn signal
	sub2	×	×	-	Lack support	right turn signal
Law58		×	×	-	Lack support	warning signal
Law59		×	×	-	Lack support	signals

magnitude of the repair to the original trajectory, i.e., the enforcement should take place only if necessary and should minimally alter the behaviour of ADS.

Our experiments were run in the high-fidelity LGSVL simulator [38]. Due to randomness in the simulator (mostly due to concurrency), each experiment was executed 100 times and we report the averages. The threshold was determined in a preliminary experiment in which we ran Apollo multiple times for each scenario to get the range of the possible robustness values. All experiments were obtained using two machines with 32GB of memory, an Intel i7-10700k CPU, and an RTX 2080Ti graphics card. The machines respectively use Linux (Ubuntu 20.04.5 LTS) and Windows (10 Pro).

RQ1: Can REDriver ***be used to enforce non-trivial specifications?*** To answer this question, we adopted the formalisation of traffic laws reported in [44] as our specification and evaluated whether REDriver can be applied so that the ADS follows them. We remark the traffic laws are rather complicated as they model 13 testable traffic laws with many sub-clauses. Furthermore, we use the benchmark of scenarios provided by [44] in which Apollo is known to violate the specification. We replay these violation-inducing scenarios with REDriver enabled, and report in Table 5 whether our approach is able to prevent the violations from occurring. Each 'subX' in Table 5 represents sub-rules of a traffic laws. For example, Law38 pertains to traffic light regulations and has three sub-rules, each covering the yellow, green, and red lights, respectively. The 'enforced?' column indicates whether the enforcement is successful. The enforcement was considered to be successful if the average passing rate of the specification after the enforcement is more than 50% across all the repetitions. Note that Apollo's passing rate is always below 50% for the selected scenarios. The *improve* column in Table 5 reports the average improvement of REDriver over Apollo, i.e., the maximum enhancement achieved by REDriver across a threshold value spectrum ranging from 0.0 to 1.2, with intervals of 0.1. The improvement is calculated by subtracting the pass rate of Apollo from the pass rate of REDriver. Note that since the sub laws of law38 and law51 are closely related, they are evaluated together for *avg improve*. The only reason that we cannot enforce some of the failed laws is that some simulator support is currently lacking. For example, we generate a command to turn on fogLight to satisfy the law58 as shown in Example 2.1 and LGSVL's car model ignores the command since it currently does not support fog lights. We mark *Lack support* in the table to illustrate this. The detailed improvement for all thresholds is shown in Figure 3. In this figure,
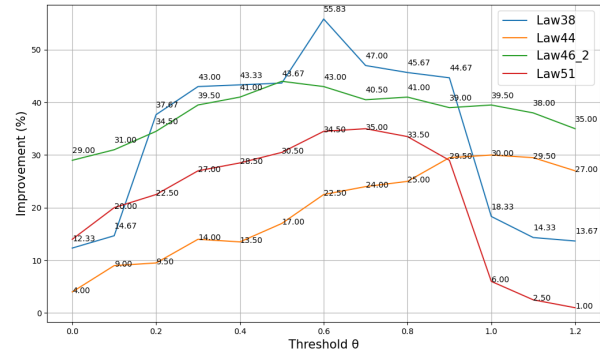
Figure 3: Improvement of performance across thresholds

the x-axis denotes the threshold value (θ), while the y-axis signifies the average percentage improvement of REDriver over Apollo. As shown in Figure 3, REDriver successfully enforced all cases where an enforcement is feasible.

To explore RQ1 in more detail, we designed a second experiment that focused on law38—one of the most complicated formulae in [44]—which specifies how a vehicle should behave at a traffic light junction (i.e., the constraints on movements due to green/yellow/red lights). We then selected three scenarios highly relevant to this traffic law: *Double Lined Junction*, *Single Direction Junction*, and *T Junction*. The detailed specification *law*38 is given in our website [1]. Note that these scenarios were generated by Law-Breaker [44] to reliably induce traffic law violations in Apollo. The seeds for the fuzzing algorithm are given on the website [1].

We tested Apollo with and without REDriver on each violation-inducing scenario 100 times and recorded the pass rate and average robustness with respect to law38. Table 6 presents the result of our evaluation using Apollo 7.0 (the results for version 6.0 are on our website [1]), where θ indicates threshold values. Recall that the smaller the robustness value, the 'closer' is a violation.

As can be seen from Table 6, REDriver significantly outperforms the original Apollo in terms of respecting the specification. In scenarios "Double-Lined Junction" and "Single-Direction Junction", the original Apollo failed to pass at the green light because it is too conservative at the junction. For instance, Apollo sometimes decides to stop before an intersection when there is enough space for the vehicle to pass safely. REDriver avoided the violations by enforcing the vehicle to drive within the junction first or pass the junction directly. As a consequence, the average improvement to the pass rate is more than 50% for scenario "Double-Lined Junction" and "Single-Direction Junction". For scenario "T-Junction", the improvement of REDriver is relatively small since there is heavy traffic in this scenario and Apollo sometimes produces the stop command. By design, the stop command has higher priority than the planned trajectory since it prevents crashing in urgent situations. Therefore, the enforcement did not take effect in some cases.

Furthermore, the performance of REDriver varies with threshold θ, i.e., θ being too small or too large both lead to degraded performance. If θ is too small, for example 0 (which is equivalent to the driver being ignorant of what is going to happen), sometimes the ADS cannot enforce in time. But if θ is too large (e.g., 1.0-1.2 which is equivalent to the driver being too scared of what might happen), REDriver is overcompensating and fixing things it should

Table 6: Performance comparison of REDriver and Apollo

Scenario	Driver	θ	pass/total	robustness	avg time
Double Lined Junction	Apollo7.0	-	**40/100**	0.27	71.11s
	REDriver	0.0	**73/100**	0.67	55.18s
	REDriver	0.1	**78/100**	0.75	51.11s
	REDriver	0.2	**85/100**	0.96	50.56s
	REDriver	0.3	**93/100**	1.05	45.67s
	REDriver	0.4	**95/100**	1.07	45.71s
	REDriver	0.5	**93/100**	1.10	45.32s
	REDriver	0.6	**98/100**	1.19	44.92s
	REDriver	0.7	**99/100**	1.21	44.90s
	REDriver	0.8	**96/100**	1.15	45.13s
	REDriver	0.9	**99/100**	1.20	45.07s
	REDriver	1.0	**80/100**	0.78	51.90s
	REDriver	1.1	**75/100**	0.71	52.71s
	REDriver	1.2	**81/100**	0.80	53.10s
Single Direction Junction	Apollo7.0	-	**18/100**	0.18	57.60s
	REDriver	0.0	**26/100**	0.38	56.38s
	REDriver	0.1	**24/100**	0.37	55.93s
	REDriver	0.2	**85/100**	0.84	55.95s
	REDriver	0.3	**89/100**	0.86	55.75s
	REDriver	0.4	**89/100**	0.89	55.69s
	REDriver	0.5	**88/100**	0.87	56.71s
	REDriver	0.6	**91/100**	0.92	56.18s
	REDriver	0.7	**95/100**	0.99	56.13s
	REDriver	0.8	**93/100**	0.96	57.09s
	REDriver	0.9	**96/100**	1.02	57.55s
	REDriver	1.0	**35/100**	0.56	57.10s
	REDriver	1.1	**31/100**	0.45	56.79s
	REDriver	1.2	**24/100**	0.33	56.61s
T-Junction	Apollo7.0	-	**45/100**	0.29	64.66s
	REDriver	0.0	**41/100**	0.32	64.10s
	REDriver	0.1	**45/100**	0.43	63.93s
	REDriver	0.2	**46/100**	0.59	60.95s
	REDriver	0.3	**50/100**	0.77	61.82s
	REDriver	0.4	**49/100**	0.79	60.76s
	REDriver	0.5	**53/100**	0.45	62.30s
	REDriver	0.6	**55/100**	0.39	61.92s
	REDriver	0.7	**50/100**	0.41	61.70s
	REDriver	0.8	**51/100**	0.35	62.89s
	REDriver	0.9	**42/100**	0.33	63.23s
	REDriver	1.0	**43/100**	0.37	63.45s
	REDriver	1.1	**40/100**	0.40	63.70s
	REDriver	1.2	**39/100**	0.41	64.07s

Table 7: Overhead of REDriver

	θ	avg fix	max fix	fix (%)	avg(ms)	max(ms)	time (%)
	0.0	26.08	35	5.09%	1.92	6.82	4.88%
	0.1	24.19	35	4.76%	1.88	9.11	4.81%
	0.2	19.20	35	4.15%	1.90	9.10	4.72%
	0.3	18.33	32	3.57%	1.87	9.23	5.05%
	0.4	22.33	35	3.89%	1.85	9.86	4.98%
	0.5	33.12	45	6.19%	1.91	8.81	4.90%
S1	0.6	32.06	45	6.07%	1.72	9.08	4.89%
	0.7	33.20	45	6.19%	1.88	9.12	5.04%
	0.8	39.75	93	7.31%	1.95	9.10	4.92%
	0.9	40.18	102	7.45%	1.96	10.53	4.85%
	1.0	87.20	230	17.29%	2.13	11.15	6.35%
	1.1	86.02	231	17.14%	2.05	10.55	6.01%
	1.2	86.05	230	16.79%	2.11	11.13	6.26%
	0.0	10.22	17	2.51%	1.67	7.01	4.52%
	0.1	10.71	17	2.55%	1.53	6.97	4.49%
	0.2	10.56	20	2.78%	1.55	6.74	4.51%
	0.3	12.05	22	2.93%	1.56	6.72	4.22%
	0.4	12.10	22	2.90%	1.55	7.15	4.21%
	0.5	12.21	20	2.95%	1.66	6.97	4.38%
S2	0.6	12.23	20	2.98%	1.54	7.53	4.31%
	0.7	12.48	22	2.73%	1.52	7.19	4.47%
	0.8	14.35	22	3.11%	1.54	6.80	4.19%
	0.9	14.75	20	3.60%	1.49	6.78	4.28%
	1.0	170.15	177	42.57%	1.88	9.15	5.89%
	1.1	169.12	185	40.82%	1.75	9.23	5.30%
	1.2	169.20	177	41.17%	2.09	9.09	5.61%
	0.0	38.92	51	9.02%	1.63	9.14	4.18%
	0.1	37.03	49	8.87%	1.67	9.21	4.50%
	0.2	37.71	49	8.89%	1.70	9.45	4.39%
	0.3	37.28	49	8.96%	1.84	9.21	4.54%
	0.4	35.22	52	8.75%	1.82	9.50	4.43%
	0.5	37.33	49	9.01%	1.71	9.34	4.70%
S3	0.6	34.70	52	8.05%	1.69	10.13	4.90%
	0.7	33.13	52	7.80%	1.75	9.12	4.82%
	0.8	35.56	40	8.35%	1.70	9.29	4.88%
	0.9	32.46	40	7.76%	1.77	9.17	4.91%
	1.0	233.73	298	52.05%	2.19	11.21	5.83%
	1.1	232.76	298	52.18%	2.27	11.20	5.79%
	1.2	234.46	298	51.99%	2.22	11.34	5.85%

not, which can lead to unexpected ADS behaviour. Note that a significant number of valid trajectories have a robustness of 1, and such an overreaction is more likely to occur for $\theta \geq 1$.

RQ2: How much overhead does the runtime enforcement impose? To answer this question, we collect information on the running time of the plan validation module of REDriver for different scenarios. The overhead for control validation is very small (less than 0.01% of the the overhead of the plan validation module), and we ignore it in the later experiment. The detailed data for REDriver based on Apollo 7.0 is shown in Table 7 (the results for version 6.0 are shown on our website [1]). Here, *S1-S3* corresponds to the *Double Lined Junction*, *Single-Direction Junction*, and *T-Junction* as in Table 6, *avg fix* represents the average number of fixes during a test that successfully enables the ADS to follow the specification, *max fix* represents the maximum number of fixes detected across the test cases, *avg(ms)* means the average time consumption of the

plan validation module in one run, *max(ms)* indicates the maximum time consumption detected, *fix (%)* is calculated by dividing the average fixes by the average updates of the planned trajectory during a run, and *time (%)* is calculated by dividing the average time consumption of the plan validation module by the average time consumption of the production of a planned trajectory. The time units in Table 7 are all milliseconds.

As can be seen from Table 7, the time consumption of the plan validation module is practical, i.e., the average time consumption is always smaller than 2.5 milliseconds, the max time consumption is always smaller than 12 milliseconds, and the *time percent* is always within 6%. Furthermore, the number of fixes is related to the value of θ as expected. There is a large increase in the number of fixes and *fix percent* for a large *threshold* ≥ 1.0 across all three scenarios. This is consistent with the performance degradation at *threshold* ≥ 1.0 shown in Table 6 since unnecessary fixes cause problems.

RQ3: Does REDriver ***minimise the enforcement?*** To answer this question, first, recall our approach as described in Section 3. We identify the smallest k with $\rho(\varphi, \pi^k) < \theta$. Hence, our fix applies
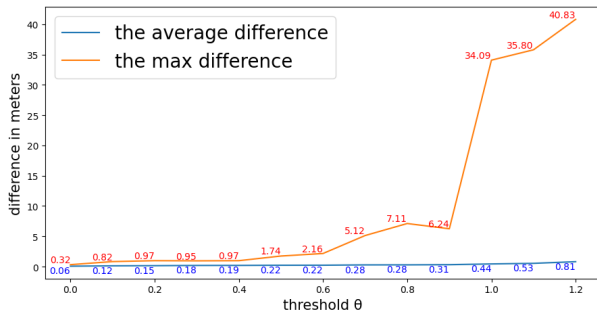
Figure 4: Magnitude of modifications to planned trajectories

only to the earliest part of the planned trajectory that leads to the near-violation of the specification. In most cases, we only modify one variable at one time step, such as the "speed" at some time step. For some rare cases, we may modify multiple variables at one time step, such as the "speed" and "position", if modifying one single variable is not sufficient. Note that the ADS updates the planned trajectory based on current perceptions and predictions and the impact of our change does not accumulate.

Here, we require a method to assess the variance between the modified planned trajectory and the original trajectory. This quantification is calculated by assessing the positional variance between these trajectories. When alterations are made to the speed or acceleration, we translate these changes into positional differences. To be precise, the conversion for speed discrepancies is determined as follows: $(speed'_t - speed_t) \cdot I$, and for acceleration discrepancies: $(acc'_t - acc_t) \cdot I^2$, where I signifies the time interval between the current planned *waypoint* and the subsequent *waypoint*. For instance, if a speed adjustment of magnitude $2m/s$ is applied to a planned waypoint, and the time interval I is $0.2s$, then the positional difference is calculated as $2m/s \cdot 0.1s = 0.2m$. The magnitude of modifications is shown in Figure 4. In this figure, the x-axis represents the threshold value θ, and the y-axis represents the magnitude of the modification of REDriver in meters. The graph presented in the figure denotes the average/max modification value of REDriver across thousands of fixes. Notably, for thresholds ranging from 0.0 to 1.2, the average difference consistently remains below 1 meter. This observation suggests that REDriver's modification on the planned trajectory is small. Note that there is a significant increase in max difference for threshold 1.0. This phenomenon is attributed to an excessive number of unnecessary fixes, as explained in RQ1.

In addition, the average running time for the test cases is listed in the last column of Table 6. Here, *avg time* represents the average time spent by the ADS to travel from the start point to the destination. As can be seen from the last column of Table 6, the running time did not increase across all these test cases. This indicates that REDriver did not, in practice, force the ADS to produce a substantially different trajectory to follow (e.g., halting the car). Note in addition that the time consumption of REDriver has dropped substantially compared to the original Apollo in the scenario "Double-Lined Junction". This is because Apollo hesitated at the green light, while REDriver successfully passed through.

5 RELATED WORK

Runtime verification approaches monitor messages obtained from ADSs and evaluate them against a specification using a number of different techniques. For instance, Kane et al. [26] generate a system trace from the observed network state, and Heffernan et al. [24] use system-on-a-chip based monitors as sources of information. Watanabe et al. [45] focus on runtime monitoring of the controller safety properties of advanced driver-assistance systems (ADASs). Mauritz et al. [30] generate monitors for ADAS features from safety requirements and by training on simulators. D'Angelo et al. [10] present Lola, a simple and expressive specification language to describe both correctness/failure assertions, which has been successfully deployed on autonomous vehicles in addition to many successful flight deployments. Note that there is no enforcement of specifications in the works mentioned above.

Runtime enforcement goes beyond monitoring and attempts to enforce certain safety properties. Existing works [8, 21, 25, 43] already propose a few methods for runtime enforcement of ADSs. AVGuardian [25] performs static analysis of the communication messages between the ADS modules to generate control policies and enforce them during runtime. Guardauto et al. [8] divide the ADS into a few partitions for the detection of rogue behaviours and restart the partition in order to clear them. Shankaro et al. [43] define a policy using an automaton and enforce the car to stop when the policy is violated. Grieser et al. [21] build an end-to-end neural network (from LIDAR to torques/steering) that implicitly picks up safety rules. Simultaneously, the distance to obstacles on the current trajectory is monitored and emergency brakes are applied if a collision is likely. Generally, when enforcement for an ADS takes place in these works, it tends to be quite 'weak', (e.g. emergency brake). REDriver, on the other hand, provides runtime enforcement for a rich specification in ways that are less intrusive.

In addition, there are existing works for cyber-physical systems [37, 48, 49]. Pinisetty et al. [37] formalise the runtime enforcement problem for CPSs, where policies depend not only on a controller but also an environment. Another approach, Safety Guard [49], adds automata-based reactive components to the original system, which react to ensure a predefined set of safety properties, while also keeping the deviation from the original system to a minimum. ModelPlex [32] checks for model compliance of cyber-physical systems and includes a fail-safe action to avoid violations of safety properties. CBSA [36] proposes the idea of integrating assume-guarantee reasoning to allow runtime assurance of cyber-physical systems. These works are relevant to the runtime enforcement of ADSs since ADSs are cyber-physical systems as well. However, we can not directly apply these methods to ADSs and customization of the enforcement techniques is necessary due to the unique requirements and challenges posed by ADSs. For instance, the enforcement of ADSs requires consideration of not only the current control commands but also the planned trajectory.

Runtime enforcement is not limited to ADSs, i.e., there are works providing runtime enforcement/verification for general systems (e.g., [5, 7, 11, 12, 16, 17, 19, 28, 33, 39–42, 47]). The Simplex architecture [5, 42] introduces the idea of "runtime enforcement" to enhance the reliability of complex software, and has been widely adopted in both academia and industry. Shield synthesis [7] proposes a method

of runtime enforcement for reactive systems while also minimising interference to the original behaviour. Schneider [40] looks at runtime enforcement of security policies and stops the program when they are violated. Falcone et al. [16] propose enforcement by buffering actions and dumping them only when deemed safe. Ligatti et al. [28] use 'edit automata' to respond to dangerous actions by suppressing them or inserting other actions. Desai et al. [11] enforce the plan trajectory of mobile robots so as to follow STL specifications. Expanding upon this idea, Soter [12] allows for safety properties to be specified and enforced in robotic systems. Tools such as TuLip [47] and LTLMoP [19] synthesize trajectories to assist evaluation of the control system under linear temporal logic (LTL) specifications. Barron Associates provide a comprehensive study of runtime enforcement architecture for highly adaptive flight ontrol systems [39]. The Copilot tool [33] offers a comprehensive runtime enforcement environment that incorporates numerous operating-system-like functionalities. The R2U2 [41] monitors the security properties of on-board Unmanned Aerial Systems (UAS) and is implemented in FPGA hardware. Unfortunately, many existing general runtime enforcement/verification methods are not suitable for ADSs due to their safety-critical and highly interactive nature. The survey paper by Falcone et al. [17] on existing runtime enforcement/verification tools clarifies that many 'reactions' provided by general runtime verification tools are weak, which is not acceptable for our situation. In this paper, we propose a runtime enforcement method applicable to any given specification with acceptable overhead for ADSs, and our method concerns not only the current driving conditions but also the ADS's future plans.

6 CONCLUSION

We proposed, REDriver, a solution to the runtime enforcement problem for ADSs. REDriver supports the enforcement of complex user-provided specifications such as national traffic laws in a way which is similar to experienced human drivers, i.e., based on near-future predictions and proactively correcting the vehicle's trajectory accordingly with minimal adjustment.

ACKNOWLEDGMENT

REFERENCES

[1] 2023. REDriver Source Codes. https://redriver2023.github.io/. Online; accessed Jan 2024.
[2] Autoware.AI. 2022. Autoware.AI. www.autoware.ai/. Online; accessed Jan 2024.
[3] Baidu. 2019. APOLLO 6.0. https://github.com/ApolloAuto/apollo/releases/tag/v6.0.0. Online; accessed Jan 2024.
[4] Baidu. 2022. APOLLO 7.0. https://github.com/ApolloAuto/apollo/releases/tag/v7.0.0. Online; accessed Jan 2024.
[5] Stanley Bak, Deepti K Chivukula, Olugbemiga Adekunle, Mu Sun, Marco Caccamo, and Lui Sha. 2009. The system-level simplex architecture for improved real-time embedded system safety. In 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium. IEEE, 99–107.
[6] Sai Krishna Bashetty, Heni Ben Amor, and Georgios Fainekos. 2020. DeepCrashTest: Turning Dashcam Videos into Virtual Crash Tests for Automated Driving Systems. In 2020 IEEE International Conference on Robotics and Automation, ICRA. Paris, France, 11353–11360.
[7] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. 2015. Shield Synthesis: - Runtime Enforcement for Reactive Systems. In TACAS'15 (Lecture Notes in Computer Science, Vol. 9035). Springer, 533–548.
[8] Kun Cheng, Yuan Zhou, Bihuan Chen, Rui Wang, Yuebin Bai, and Yang Liu. 2021. Guardauto: A Decentralized Runtime Protection System for Autonomous Driving. IEEE Trans. Computers 70, 10 (2021), 1569–1581.
[9] Chinese Government. 2021. Regulations for the Implementation of the Road Traffic Safety Law of the People's Republic of China. http://www.gov.cn/gongbao/content/2004/content_62772.htm. Online; accessed Jan 2024.
[10] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: runtime monitoring of synchronous systems. In 12th International Symposium on Temporal Representation and Reasoning (TIME'05). IEEE, 166–174.
[11] Ankush Desai, Tommaso Dreossi, and Sanjit A Seshia. 2017. Combining model checking and runtime verification for safe robotics. In Runtime Verification: 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings. Springer, 172–189.
[12] Ankush Desai, Shromona Ghosh, Sanjit A Seshia, Natarajan Shankar, and Ashish Tiwari. 2019. SOTER: a runtime assurance framework for programming safe robotics systems. In 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 138–150.
[13] Jyotirmoy V Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A Seshia. 2017. Robust online monitoring of signal temporal logic. Formal Methods in System Design 51, 1 (2017), 5–30.
[14] Vinayak V Dixit, Sai Chand, and Divya J Nair. 2016. Autonomous vehicles: disengagements, accidents and reaction times. PLoS one 11, 12 (2016), e0168054.
[15] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An open urban driving simulator. In Conference on Robot Learning. 1–16.
[16] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. 2012. What can you verify and enforce at runtime? Int. J. Softw. Tools Technol. Transf. 14, 3 (2012), 349–382.
[17] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. 2021. A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. 23, 2 (2021), 255–284.
[18] Francesca M Favarò, Nazanin Nader, Sky O Eurich, Michelle Tripp, and Naresh Varadaraju. 2017. Examining accident reports involving autonomous vehicles in California. PLoS one 12, 9 (2017), e0184952.
[19] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. 2010. LTLMoP: Experimenting with language, temporal logic and robot control. In 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 1988–1993.
[20] Yann Gilpin, Vince Kurtz, and Hai Lin. 2020. A smooth robustness measure of signal temporal logic for symbolic control. IEEE Control Systems Letters 5, 1 (2020), 241–246.
[21] Jörg Grieser, Meng Zhang, Tim Warnecke, and Andreas Rausch. 2020. Assuring the Safety of End-to-End Learning-Based Autonomous Driving through Runtime Monitoring. In DSD. IEEE, 476–483.
[22] Andreas Griewank et al. 1989. On automatic differentiation. Mathematical Programming: recent developments and applications 6, 6 (1989), 83–107.
[23] Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. 2019. Towards a Two-Layer Framework for Verifying Autonomous Vehicles. In NFM (Lecture Notes in Computer Science, Vol. 11460). Springer, 186–203.
[24] Donal Heffernan, Ciaran MacNamee, and Padraig Fogarty. 2014. Runtime verification monitoring for automotive embedded systems using the ISO 26262 functional safety standard as a guide for the definition of the monitored properties. IET Softw. 8, 5 (2014), 193–203.
[25] David Ke Hong, John Kloosterman, Yuqi Jin, Yulong Cao, Qi Alfred Chen, Scott Mahlke, and Z Morley Mao. 2020. AVGuardian: Detecting and mitigating publish-subscribe overprivilege for autonomous vehicle systems. In 2020 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 445–459.
[26] Aaron Kane, Omar Chowdhury, Anupam Datta, and Philip Koopman. 2015. A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System. In RV'15 (Lecture Notes in Computer Science, Vol. 9333). Springer, 102–117.
[27] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael Sullivan, Siva Kumar Sastry Hari, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2020. AV-FUZZER: Finding safety violations in autonomous driving systems. In 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). IEEE, 25–36.
[28] Jay Ligatti, Lujo Bauer, and David Walker. 2009. Run-Time Enforcement of Nonsafety Policies. ACM Trans. Inf. Syst. Secur. 12, 3 (2009), 19:1–19:41.
[29] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems. 152–166.
[30] Malte Mauritz, Falk Howar, and Andreas Rausch. 2016. Assuring the Safety of Advanced Driver Assistance Systems Through a Combination of Simulation and Runtime Monitoring. In ISoLA (2) (Lecture Notes in Computer Science, Vol. 9953). 672–687.

[31] Roger L McCarthy. 2022. Autonomous vehicle accident data analysis: California OL 316 reports: 2015–2020. *ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part B: Mechanical Engineering* 8, 3 (2022), 034502.

[32] Stefan Mitsch and André Platzer. 2016. ModelPlex: Verified runtime validation of verified cyber-physical system models. *Formal Methods in System Design* 49 (2016), 33–74.

[33] NASA. 2023. Copilot. https://nari.arc.nasa.gov/sws-tc3-diagram/capability/copilot/. Online; accessed Jan 2024.

[34] Dejan Ničković and Tomoya Yamaguchi. 2020. RTAMT: Online robustness monitors from STL. In *International Symposium on Automated Technology for Verification and Analysis*. 564–571.

[35] Yash Vardhan Pant, Houssam Abbas, and Rahul Mangharam. 2017. Smooth operator: Control using the smooth robustness of temporal logic. In *2017 IEEE Conference on Control Technology and Applications (CCTA)*. IEEE, 1235–1240.

[36] Dung Phan, Junxing Yang, Matthew Clark, Radu Grosu, John Schierman, Scott Smolka, and Scott Stoller. 2017. A component-based simplex architecture for high-assurance cyber-physical systems. In *2017 17th International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, 49–58.

[37] Srinivas Pinisetty, Partha S. Roop, Steven Smyth, Nathan Allen, Stavros Tripakis, and Reinhard von Hanxleden. 2017. Runtime Enforcement of Cyber-Physical Systems. *ACM Trans. Embed. Comput. Syst.* 16, 5s (2017), 178:1–178:25.

[38] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. 2020. LGSVL simulator: A high fidelity simulator for autonomous driving. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. 1–6.

[39] John D Schierman, Michael D DeVore, Nathan D Richards, Neha Gandhi, Jared K Cooper, Kenneth R Horneman, Scott Stoller, and Scott Smolka. 2015. *Runtime assurance framework development for highly adaptive flight control systems*. Technical Report. Barron Associates, Inc. Charlottesville.

[40] Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50.

[41] Johann Schumann, Patrick Moosbrugger, and Kristin Y Rozier. 2015. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. In *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*. Springer, 233–249.

[42] Lui Sha et al. 2001. Using simplicity to control complexity. *IEEE Software* 18, 4 (2001), 20–28.

[43] Saumya Shankar, Ujwal V. R, Srinivas Pinisetty, and Partha S. Roop. 2020. Formal Runtime Monitoring Approaches for Autonomous Vehicles. In *OVERLAY'20 (CEUR Workshop Proceedings, Vol. 2785)*. CEUR-WS.org, 89–94.

[44] 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 *ASE*. ACM, 62:1–62:12.

[45] Kosuke Watanabe, Eunsuk Kang, Chung-Wei Lin, and Shinichi Shiraishi. 2018. Runtime monitoring for safety of intelligent vehicles. In *DAC*. ACM, 31:1–31:6.

[46] Ching-Feng Wen and Hsien-Chung Wu. 2012. Using the parametric approach to solve the continuous-time linear fractional max–min problems. *Journal of Global Optimization* 54, 1 (2012), 129–153.

[47] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M Murray. 2011. TuLiP: a software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*. 313–314.

[48] Meng Wu, Jingbo Wang, Jyotirmoy Deshmukh, and Chao Wang. 2019. Shield Synthesis for Real: Enforcing Safety in Cyber-Physical Systems. In *FMCAD*. IEEE, 129–137.

[49] Meng Wu, Haibo Zeng, Chao Wang, and Huafeng Yu. 2017. Safety Guard: Runtime Enforcement for Safety-Critical Cyber-Physical Systems: Invited. In *DAC*. ACM, 84:1–84:6.

[50] Yuan Zhou, Yang Sun, Yun Tang, Yuqi Chen, Jun Sun, Christopher M. Poskitt, Yang Liu, and Zijiang Yang. 2023. Specification-Based Autonomous Driving System Testing. *IEEE Trans. Software Eng.* 49, 6 (2023), 3391–3410.