T4PC: Training Deep Neural Networks for Property Conformance

Felipe Toledo[1], Trey Woodlief[1], Sebastian Elbaum[1], and Matthew B. Dwyer[1]

***Abstract*—The increasing integration of Deep Neural Networks (DNNs) into safety critical systems, such as Autonomous Vehicles (AVs), where failures can lead to significant consequences, has fostered the development of many Verification and Validation (V&V) techniques. However, these techniques are applied mainly after the DNN training process is complete. This delayed application of V&V techniques means that property violations found require restarting the expensive training process, and that V&V techniques struggle in pursuit of checking increasingly large and sophisticated DNNs. To address this issue, we propose T4PC, a framework to increase property conformance *during* DNN training. Increasing property conformance is achieved by enriching: 1) the data preparation phase to account for properties' pre and postcondition satisfaction, and 2) the training phase to account for the property satisfaction by incorporating a new *property loss* term that is integrated with the main loss. Our family of controlled experiments targeting a navigation DNN show that T4PC can effectively train it for conformance to single and multiple properties, and can also fine-tune for conformance an existing navigation DNN originally trained for accuracy. Our case study in simulation applying T4PC to fine-tune two open source AV systems operating in the CARLA simulator shows that it can reduce targeted driving violations while retaining its original driving capabilities.**

***Index Terms*—Deep learning, Autonomous vehicles, Software and System Safety.**

I. INTRODUCTION

Advances in deep machine learning have demonstrated the ability to synthesize high-quality implementations of deep neural networks (DNNs). Contrary to traditional software development, building DNNs is mostly an automated optimization process. Conceptually, given a dataset of values x and corresponding labels y, constructing a deep neural network N entails an iterative process of computing $N(x)$, comparing it to y, and adjusting N to minimize their difference.

The increasing integration of DNNs into safety critical systems such as autonomous vehicles (AVs) [1]–[3] and their high visibility failures [4]–[7] have prompted the development of many specialized Verification and Validation (V&V) techniques. To position our contribution among those techniques, we classify them along two dimensions.

First, when classified based on the type of properties being analyzed, we find that most V&V techniques for DNNs operate on metamorphic properties over sensor inputs and output commands. For example, a property capturing the system's robustness to sensor noise may state that *"changing a small number of pixels in a camera image should not affect the steering angle by more than a given threshold"*. Verification techniques aim to provide guarantees that the DNNs satisfy such robustness properties, e.g., [8]–[11], while validation techniques aim to generate diverse inputs that may induce property violations, e.g., [12]–[15]. As V&V techniques have matured, researchers have begun to target richer system-level specifications such as *"when a vehicle approaches an intersection controlled by a stop sign, it shall stop at the marked line"*. This is challenging due to the semantic gap between the high-dimensional raw sensor inputs, e.g., image pixels, and the space over which such specifications are defined, e.g., vehicles and stop signs. Existing V&V techniques either assume that such higher-level data is available [16] or develop bespoke abstraction methods to move from the sensor to the property space [17]. Recent work has developed more general abstractions of sensor inputs that can be used to judge a dataset's completeness [18] or monitor a trained DNN for property violations [19], and we build on those in this work.

Second, when classified by the stage at which V&V is applied, we find that the techniques operate after the training process is completed, acting as quality assurance gatekeepers for DNNs instead of being an integral part of the development [20], [21]. Performing V&V after training means that detecting that N fails to conform to a desired property occurs late in the development. This can significantly increase costs as it requires repair, perhaps through re-training, or worse, the realization that more and particular data needs to be collected and labeled. Researchers have explored methods for integrating property specifications in the training process, e.g., [22]–[25], but these are either limited to DNNs with small input spaces or properties that only constrain the DNNs' output. Such techniques cannot support properties with preconditions specified over complex high-dimensional input spaces.

In summary, existing DNN V&V approaches are either too limited or come too late to meet the challenges of DNNs used in complex systems like those in the AV domain. We conjecture that adapting the training process to incorporate high-level specifications of desired DNN behavior can overcome these limitations. To achieve this, an approach must overcome two key challenges. First, it must be able to optimize not just based on label matching, but also based on reduction of property violations. Second, it must be able to evaluate property preconditions over high-dimensional raw sensor inputs.

In this work, we define an approach, T4PC, that overcomes those challenges. Building on recent work in AV property monitoring [19], T4PC enriches the data preparation process by abstracting raw sensor inputs captured by an AV using scene graphs (SGs), structured representations of a scene, to allow the evaluation of high-level specifications and the generation of property labels. Then, inspired by work in training DNNs with logical constraints, e.g., [24], T4PC incorporates a novel property loss term into training and utilizes a dual optimization process to balance property loss with traditional label loss. We implemented T4PC and performed a family of controlled experiments targeting a navigation system that

[1]University of Virginia, USA {ft8bn, adw8dm, selbaum, matthewbdwyer}@virginia.edu

predicts steering angle and acceleration, assessing its potential to increase property conformance. We find that T4PC can train navigation DNNs that improve property conformance over a base DNN by 67% for single and multiple properties. Our case study in simulation on two state-of-the-art AV systems shows that fine-tuning the system with T4PC reduces the targeted infractions by up to 38% and improves the overall driving score. These findings provide evidence that T4PC, and more generally mechanisms for training DNNs for property conformance, can offer a cost-effective complement to existing V&V techniques.

II. Background and Related Work

This section provides a summary of property specifications for DNNs, scene graphs, relational specifications, and training approaches for improving DNNs.

A. Properties of DNNs

A dataset, D, is comprised of input, output pairs, (x, y), where the output defines expected DNN behavior on the input, $y = N(x)$. The property specification, $\phi(x, y) = \phi_{\mathcal{X}}(x) \implies \phi_{\mathcal{Y}}(y)$, defines constraints over the inputs and outputs [26]. The constraint over inputs, $\phi_{\mathcal{X}}$, is termed the *precondition* and the constraints over outputs, $\phi_{\mathcal{Y}}$, is termed the *postcondition*. Properties can be evaluated over a dataset, $\forall (x, y) \in D : \phi(x, y)$, or at DNN evaluation time, given x evaluate $\phi(x, N(x))$.

In settings where primitive propositions can be defined, previous work has had success at applying a variety of formalisms, including First Order Logic (FOL) [27] with spatial and temporal requirements [28], [29], Linear Temporal Logic (LTL) [30]–[32], and Signal Temporal Logic (STL) [33], [34], to describe the behavior of critical systems, including autonomous systems, and their properties [35]. However, such primitive propositions are only viable over data generated by simpler sensor types (e.g., gyroscopes, proximity, pressure).

A key remaining challenge in specifying useful properties lies in the ability to express domain-specific *primitive propositions*. In the context of AV systems, as we target here, the challenge is how to encode and check for the presence of a vehicle, pedestrian, or stop sign over raw sensor inputs, e.g., camera or lidar data. While researchers have developed expressive languages to encode properties over such input spaces [9], [36], this does not address the core challenge of expressing such concepts due to the variability of their presentation in complex sensor inputs (i.e., the different ways a person may appear in an image, or a vehicle may relate to a person in a point cloud). In this work, we focus on FOL properties as they can capture relevant aspects of AV behavior and represent a first step toward end-to-end property conformance. We defer the integration of temporal properties to future work, as their reliance on input sequences poses further challenges for training as some AV systems handle temporal aspects internally rather than through input sequences.

B. Scene Graphs

Researchers have recently proposed the use of *scene graphs* (SGs) as an abstraction of multi-dimensional sensor data such as images and point clouds over which it is practical to express domain-specific primitive propositions. Scene graph generation builds a graph that encodes the semantic relationships between objects in a scene and the relationships between those objects with their surroundings as, e.g., Figure 1b [37], [38].

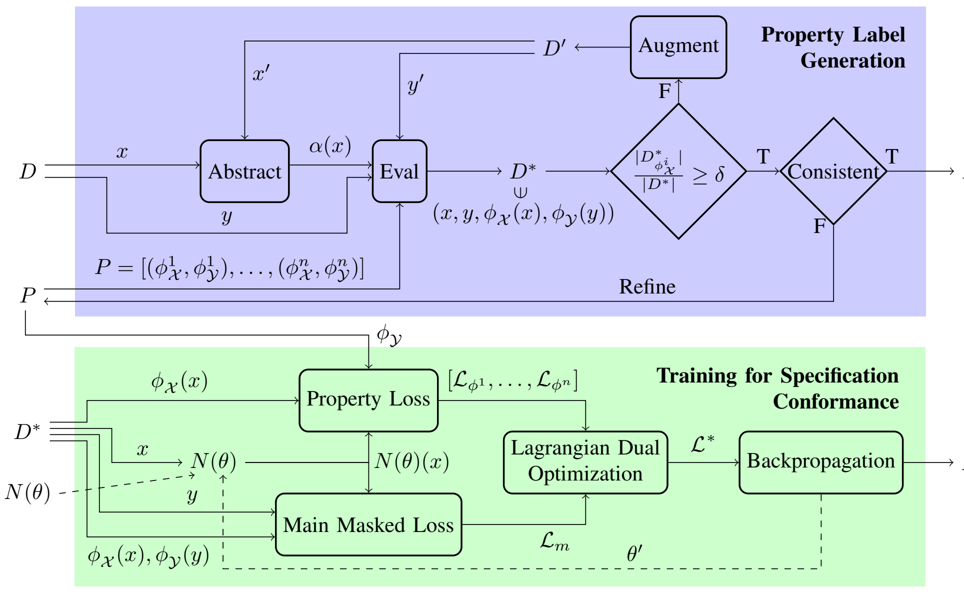
A *scene graph generator* (SGG) maps sensor inputs, I, to a graph representation, $sg : I \mapsto G$. SGs are directed graphs, with a vertex set, V, that represents the set of entities captured by the sensor, and a set of edges, $(u, v) \in E$, describing their relationships. Formally, $G = (V, E : V \mapsto V, rel : E \mapsto R, att : V \cup E \mapsto A)$ with functions to access the relation encoded by an edge, and attribute values of vertices and edges.

More specialized SGGs have emerged for particular domains, such as AVs [39]–[41], that capture semantics that are more relevant for driving scenes, such as the number of lanes, types of vehicles, traffic signage, and pedestrians. In the AV domain, SGs have been studied for dataset evaluation [18], runtime monitoring [19], and risk assessment [39]. Yet obtaining SGs of sufficient quality remains an ongoing challenge. In this work we utilize a simulation-based SGG [42] to evaluate our approach in a controlled setting independent from potential noise and faults in current research-prototype SGGs. However, SGGs are an area of research that is advancing rapidly in both the accuracy of SGs and the flexibility of SGGs in capturing entities and relationships [43]. As sensor-based SGGs become more capable, they can be easily integrated into T4PC.

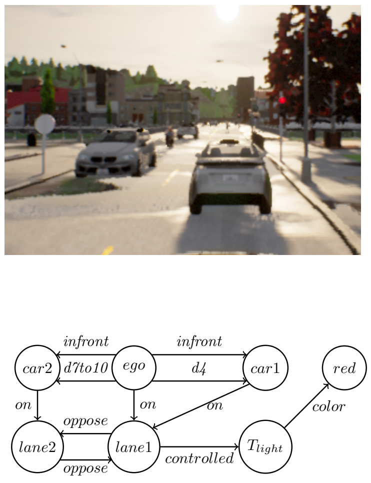
T4PC requires that the SGG utilized be sufficiently expressive and accurate to capture the primitives needed to encode the relevant properties against which the DNN will be evaluated. As studied in prior work on applying SGs to the AV domain, this includes entities to represent other road users, e.g., vehicles, pedestrians, and bicycles; entities to represent traffic semantics, e.g., traffic lights, stop signs, and junctions; and relations such as parameterized distances between entities, and which lane a vehicle occupies or a traffic light controls [18], [19]—such entities and relations are captured with perfect accuracy by the simulation-based SGG we utilize in our experiments [42]. An example SG is shown in Figure 1b. Recent work [44] showed the potential of visual language models to generate SGs from images as part of a property runtime monitor. Future work should investigate how to incorporate more information into the SG, explore the impacts of inaccuracies or imprecision in the SGs, and the generality of T4PC with SGs tailored for other domains.

C. Relational Specifications

SGs offer the potential to leverage specification methods appropriate to graph-structured data, such as *relational first order logic* (RFOL). RFOL includes set theoretic, relational, and logical sub-languages. The set theory support includes standard operators, e.g., intersection ($\cap$), containment ($\in$), and equality ($=$), and important constants, e.g., the empty set ($\emptyset$). The relational support includes operators like relational image (.) and inverse image (.^). The logical sub-language includes


Property Label Generation
Augment
D'
x'
y'
F
D
x
Abstract
$\alpha(x)$
Eval
D^*
$\Cup$
$(x, y, \phi_{\mathcal{X}}(x), \phi_{\mathcal{Y}}(y))$
$\frac{|D^*_{\phi^i_{\mathcal{X}}}|}{|D^*|} \geq \delta$
T
Consistent
T
D^*
F
y
$P = [(\phi^1_{\mathcal{X}}, \phi^1_{\mathcal{Y}}), \ldots, (\phi^n_{\mathcal{X}}, \phi^n_{\mathcal{Y}})]$
P
Refine
$\phi_{\mathcal{Y}}$
Training for Specification Conformance
$\phi_{\mathcal{X}}(x)$
Property Loss
$[\mathcal{L}_{\phi^1}, \ldots, \mathcal{L}_{\phi^n}]$
D^*
x
$N(\theta)$
$N(\theta)(x)$
Lagrangian Dual Optimization
$\mathcal{L}^*$
Backpropagation
N^*
$N(\theta)$
y
Main Masked Loss
$\mathcal{L}_m$
$\phi_{\mathcal{X}}(x), \phi_{\mathcal{Y}}(y)$
θ'

(a) T4PC: phase 1 (blue) generates property labels for dataset D, and phase 2 (green) uses these labels to train a model N to conform to properties P outputting N^*, a model with improved property conformance.



(b) An image (top) and a portion of its scene graph (bottom).

Fig. 1: Overview of T4PC (left) and sample camera input from *ego* vehicle and a portion of its scene graph abstraction (right).

the usual logical connectives. Recent work on monitoring safety driving properties [19] introduced a domain specific language called "Scene Graph Language (SGL)" that contains low-level SG querying functions to facilitate the specification and checking of properties. SGL is sufficient to express all of the RFOL operators described earlier, allowing RFOL to be compiled for efficient evaluation over SGs. For example, ϕ_2 in Table I, which states *"If the ego lane traffic light is red or yellow, then ego should stop,"* has its precondition evaluated as true in Figure 1b, since ego is on a lane with a red light.

D. Runtime Enforcement & Informed Machine Learning

Purely data-driven approaches to learning may exhibit poor performance when there is insufficient data to train a generalized DNN, or when the trained DNN must meet constraints dictated by natural laws, regulations, or guidelines [45]. *Runtime enforcement* techniques complement learning approaches by monitoring and correcting system behaviors at runtime to ensure compliance with specified properties. For example, REDriver [46] can enforce AV properties based on the perception output of the system. While runtime enforcement acts after a system has made a decision, *informed machine learning* seeks to integrate prior knowledge directly into the training process [47], enabling the system to internalize such constraints. Different sources of knowledge can be represented in different ways like: algebraic equations, logic rules, or knowledge graphs. Moreover, this field has developed multiple strategies to integrate these representations into the machine learning pipeline by, for example, adding training data, modifying the network architecture, adding regularization terms, shaping a reward function, or altering the optimization. Among these strategies are techniques for modifying reinforcement learning algorithms rewards to satisfy temporal properties [48]–[50], enriching the loss term used in training to integrate algebraic equations [51]–[53], and logical formulas [22]–[25]. A key difference between these approaches and ours — and the novelty of our approach — is that T4PC uses sensor input abstractions (SGs) for evaluating safe driving property preconditions, to selectively integrate loss functions capturing postcondition violations on DNN outputs, enabling the enforcement of system-level properties during training, as we explain in the next section.

III. APPROACH

To improve the property conformance of learned components, we address two key challenges. First, we develop a novel approach to evaluate property preconditions over high dimensional sensor inputs. Second, we develop a method for incorporating loss terms to minimize property violations while maximizing DNN accuracy. In addressing these challenges, we also propose solutions to address the issues of data sufficiency—that there is enough data for each specified property—and property consistency—that the set of properties do not conflict. These methods are integrated into the Training for Property Conformance (T4PC) approach.

A. Overview

Figure 1a depicts the two phases of T4PC. The property label generation phase (blue) takes a dataset, $(x, y) \in D$, and set of correctness properties, P, and enhances the dataset, D^*, prior to training. The training for specification conformance phase (green) uses the enhanced dataset and properties to train a given DNN architecture, N, and an optional set of pretrained weights, θ, to produce an enhanced DNN, N^*, that better conforms to the properties.

TABLE I: Properties with preconditions ($\phi_{\mathcal{X}}$) in RFOL over SGs and postconditions ($\phi_{\mathcal{Y}}$) as DNN output constraints.

Prop (Rule)	English description	$\phi_{\mathcal{X}}$	$\phi_{\mathcal{Y}}$
ϕ_1 (46.2-816)	If ego has an entity within 10m in front and in the same lane, then ego should stop.	$ego.dist(\leq, 10) \cap ego.on.\hat{}on \cap ego.infront \neq \emptyset$	$N(x).acc \leq -0.25$
ϕ_2 (46.2-833)	If ego lane traffic light is red/yellow, then ego should stop.	$ego.controlled.T_{light}.color \in \{red, yellow\}$	$N(x).acc \leq -0.25$
ϕ_3 (46.2-888)	If ego has no entity within 25m in front and in the same lane, there is no red or yellow traffic light, and there is no stop sign, then ego should accelerate.	$ego.controlled.stopsign = \emptyset \wedge$ $ego.controlled.T_{light}.color \notin \{red, yellow\} \wedge$ $ego.dist(\leq, 25) \cap ego.on.\hat{}on \cap ego.infront = \emptyset$	$N(x).acc \geq 0.25$
ϕ_4 (46.2-833)	If there is a green traffic light and nothing in front of ego in the same lane within 10m, then ego should accelerate.	$ego.controlled.T_{light}.color = green \wedge$ $ego.dist(\leq, 10) \cap ego.on.\hat{}on \cap ego.infront = \emptyset$	$N(x).acc \geq 0.25$
ϕ_5 (46.2-802)	If ego is in the rightmost lane and not in a junction, then ego should not steer to the right.	$ego.on.right = \emptyset \wedge ego.junction = \emptyset$	$N(x).steer < 0.07$
ϕ_6 (46.2-804)	If ego is in the leftmost lane and not in a junction, then ego should not steer to the left.	$ego.on.left = \emptyset \wedge ego.junction = \emptyset$	$N(x).steer > -0.07$
ϕ_7 (46.2-821)	If ego is moving and there is a stop sign affecting it within 10m, then ego should stop.	$ego.speed \geq 0.1 \wedge$ $ego.dist(\leq, 10) \cap ego.controlled.stopsign \neq \emptyset$	$N(x).acc \leq -1.00$
ϕ_8 (46.2-821)	If ego is stopped and there is a stop sign affecting it within 10m and there is nothing in front in the same lane within 7m, then ego should accelerate.	$ego.speed < 0.1 \wedge$ $ego.dist(\leq, 10) \cap ego.controlled.stopsign \neq \emptyset \wedge$ $ego.dist(\leq, 7) \cap ego.on.\hat{}on \cap ego.infront = \emptyset$	$N(x).acc \geq 0.75$

Phase one (blue) accepts a dataset of labeled training instances, (x, y), and a set of property specifications, $\phi^i_{\mathcal{X}} \implies \phi^i_{\mathcal{Y}} \in P$, where the precondition, $\phi_{\mathcal{X}}$, is specified in relational first-order logic (RFOL) defined over an abstraction of the raw sensor input, and the postcondition, $\phi_{\mathcal{Y}}$, is defined as a conjunction of interval constraints in the DNN's output space.

To evaluate properties over high dimensional sensor data, each input, x, is **abstract**ed, $\alpha(x)$, and fed to a property **eval**uation component along with the associated label, y, and the properties, P. For an input, x, the pre and postconditions for all properties are evaluated to produce Boolean vectors indexed by property index, $\phi_{\mathcal{X}}(x) = [\phi^i_{\mathcal{X}}(\alpha(x)) : i \in [1, n]]$ and $\phi_{\mathcal{Y}}(x) = [\phi^i_{\mathcal{Y}}(x) : i \in [1, n]]$. The ith property is *active* for an input if its precondition is true, $\phi_{\mathcal{X}}(x)[i]$, and it *holds* for an input if $\phi_{\mathcal{X}}(x)[i] \implies \phi_{\mathcal{Y}}(y)[i]$. The output of the evaluation component is an enhanced dataset, D^*, that includes the evaluation results for pre and postconditions along with the original training instance.

To address the data sufficiency challenge, T4PC checks whether the ratio between the number of inputs meeting the precondition for each property and the total number of samples is above a parameterized threshold. If not, then there is insufficient data for some precondition and the dataset is **augment**ed to generate additional training instances, $(x', y') \in D'$, which are then abstracted, evaluated, and added to D^*.

If sufficient data is available for all properties, then T4PC checks that the properties are **consistent** relative to the dataset. This means that if an input satisfies the preconditions of two properties, then it must be possible to satisfy both of their postconditions. Resolving inconsistency among properties requires that the developer **refine** the specifications. The enhanced, augmented, and sufficient dataset, D^*, for a set of consistent properties, P, is then used as input for the second phase.

Phase two (green) takes the dataset, D^*, the properties, P, and a deep neural network, N, as input. To accommodate the application of T4PC to pre-trained networks, the set of network parameters, θ, can be provided as input as well. When θ is understood from the context we write $N(\theta)$ as N.

T4PC splits training for a data item, $(x, y, \phi_{\mathcal{X}}(x), \phi_{\mathcal{Y}}(y)) \in D^*$, into the calculation of two loss terms. The **main masked loss** term which typically consists of computing $\mathcal{L}_m = \|N(x) - y\|$, but in T4PC also incorporates a check of $\phi_{\mathcal{X}}(x) \implies \phi_{\mathcal{Y}}(y)$ to *mask* the loss of any property violations present in the training data. The **property loss** term computes, for each property i, the distance from the DNN prediction to the nearest point satisfying the postcondition, $\mathcal{L}_{\phi^i} = \|N(x) - \phi^i_{\mathcal{Y}}\|$.

To appropriately blend the $n + 1$ loss terms, T4PC uses the framework of **Lagrangian dual optimization**. In this process, the property losses, $[\mathcal{L}_{\phi^1}, \ldots, \mathcal{L}_{\phi^n}]$, are combined with the main loss, $\mathcal{L}$, to produce a single loss, $\mathcal{L}^*$, that aims to balance the performance of the DNN—matching training labels—and conformance with specifications. This loss term is used to update the DNN parameters, θ', through backpropagation. Training iterations continue for a specified number of epochs at which point the final DNN, N^*, is produced. By considering the abstracted sensor inputs during training time, T4PC produces N^*, with the aim of achieving the original training goal and also higher levels of property conformance. A more detailed description of T4PC follows.

B. Property Label Generation

The objective of this phase is to make a given dataset amenable for training a DNN for property conformance.

1) Abstract Inputs: To enable the evaluation of preconditions over inputs with high-dimensional sensor inputs such as image data, T4PC uses scene graphs as an abstraction of sensor inputs, $\alpha = sg$. Prior work [18], [19] has shown one can define a scene graph abstraction based on a set of properties. This involves identifying the kinds of entities and relations mentioned across a set of properties and using those to define the vertex and edge sets of the SG.

For example, given the set of preconditions, $\phi_{\mathcal{X}}$, for the properties shown in Table I, extracted from the Virginia Driving Code [54], the set of relations in an SG must include: $dist(op, d)$, relations indexed by relational operator, op, and distance, d; and on, $infront$, $light$, $color$, $speed$, $stopsign$, $right$, $junction$, and $left$. We note that $color$ and $speed$ define attributes—a special class of relations whose image is a singleton set. Generating such SG permits the evaluation of preconditions over $\alpha(x)$ using methods from [19].

Figure 1b depicts an image and its associated scene graph which satisfies the preconditions of ϕ_1 and ϕ_2. In this SG, the family of distance relations includes: $d4$ meaning that the distance is less than or equal to 4m, and $d7to10$ meaning the distance is between 7m and 10m, both of these are included in the relations defined by $dist(\leq, 10)$.

2) Evaluate Properties: Evaluating $\phi_{\mathcal{X}}$ involves compiling the precondition to a traversal of the SG; e.g., for $\phi^1_{\mathcal{X}}$ on the SG from Figure 1b, this determines that the set of vehicles that occupy the lane of the ego vehicle, specified as $ego.on.\hat{}on$, includes a vehicle, $car1$, that lies in the image of relation $dist(\leq, 10)$ relative to ego. For $\phi^2_{\mathcal{X}}$, the SG traversal determines that the color of the light governing the ego vehicle lane, $ego.controlled.T_{light}.color$, is red, so this precondition is also satisfied. The Boolean evaluations of all preconditions on $\alpha(x)$ are stored, $\phi_{\mathcal{X}}(x)$, for use in subsequent processing.

Postconditions, $\phi_{\mathcal{Y}}$, are formulated over DNN outputs, $N(x)$, which for a training instance is the label, y. In this work, postconditions are conjunctions of axis-aligned linear constraints which define *hyper-rectangular* constraints in the DNN output space. Given a training instance, (x, y), we can evaluate the postcondition $\phi_{\mathcal{Y}}(y)$ by directly evaluating the linear constraints. For example, both ϕ_1 and ϕ_2 have the same postcondition, $N(x).acc \leq -0.25$, which requires the acceleration output of the DNN to indicate deceleration. The Boolean outcomes of evaluating all postconditions on labels, y, are stored, $\phi_{\mathcal{Y}}(y)$, for use in subsequent processing.

We store pre and postconditions separately, since this allows us to efficiently determine whether a training instance satisfies a precondition, $\phi_{\mathcal{X}}(x)$, and property, $\phi_{\mathcal{X}}(x) \implies \phi_{\mathcal{Y}}(y)$.

3) Data augmentation: This module addresses the need for data sufficiency relative to property preconditions. A dataset must have enough training samples that satisfy each property precondition, $\phi^i_{\mathcal{X}}$, in order for DNN training to be able to learn to conform to the property. Let

$$D^*_{\phi^i_{\mathcal{X}}} = \{x : x \in D^* \wedge \phi_{\mathcal{X}}(x)[i]\}$$

be the input samples that satisfy ϕ^i's precondition. A dataset is *insufficient* for a property, ϕ^i, if the ratio of training inputs that satisfy its precondition is below a threshold, $\frac{|D^*_{\phi^i_{\mathcal{X}}}|}{|D^*|} < \delta$. If this inequality holds, then we must generate at least $t_i = \left\lceil \frac{\delta|D^*| - |D^*_{\phi^i_{\mathcal{X}}}|}{1-\delta} \right\rceil$ new inputs for D^* through augmentation that satisfy the precondition; this is the least integer value added to the numerator and denominator of the left side of the threshold inequality that guarantees it will be false.

To minimize the impact of augmentation on the data distribution, we wish to transform elements of the dataset that are the closest to satisfying $\phi^i_{\mathcal{X}}$. Since preconditions are formulated over α, we define "distance to satisfaction" as:

$$\|x - \phi^i_{\mathcal{X}}\| = \min_{x' \in D^*_{\phi^i_{\mathcal{X}}}} \|\alpha(x) - \alpha(x')\|$$

as the minimum distance from x to a training input x' that satisfies the precondition. This provides a pool of transformation candidates, $C = \{(x, d) : x \in D^* \wedge d = \|x - \phi^i_{\mathcal{X}}\|\}$. We then select t_i values to augment, $D' \subseteq C$, where $|D'| = t_i$, and $\forall (x', d') \in D' : \forall (x, d) \in C - D' : d' \leq d$. As we discuss next, it may be possible to augment the same data point multiple ways; if so, $|D'| \leq t_i$.

We augment the dataset using metamorphic transformations. A metamorphic transformation consists of a pair of functions $f : X \rightarrow X$ and $g : Y \rightarrow Y$ that modify the input and output, respectively, while preserving a consistency relationship such that for any correct implementation, $h : X \rightarrow Y$, the transformation ensures that $\forall x, y : g(y) = h(f(x))$. Such f and g must be carefully crafted by the developer with respect to the requirements to ensure this validity holds.

To illustrate, consider a driving dataset where inputs, $x = (i, s)$, consist of an image, i, and the speed, s, of ego vehicle, with outputs, y, defining acceleration. A property might specify that: *"if there is a car stopping in front of the ego vehicle and ego vehicle speed is greater than 5 km/h, then ego vehicle should stop"*. An input x containing an image that shows a car with its brake lights illuminated in front of the ego vehicle and a speed of 2 km/h, does not satisfy the precondition of the property because the speed is less than 5 km/h. This input could be augmented using function $f(x) = (i, rand(minSpeed, maxSpeed))$ that (potentially repeatedly) chooses a speed within the speed range, and $g(y) = y$ to preserve the output. If a data point had a speed of 10km/h, but the image lacked the car in front of ego, a function $f(x) = (addCarFrontEgo(i), s)$ could be used [55].

4) Check Consistency: T4PC is designed to train for conformance with a set of properties. It is possible, however, that two properties are inconsistent. For example, if we modified the precondition of ϕ_1 to use an equality rather than the disequality, then this precondition would subsume the precondition of ϕ_4, which additionally constrains the color of the light governing the ego vehicle lane. Having such properties would be problematic because $\neg\exists y : \phi^1_{\mathcal{Y}}(y) \wedge \phi^4_{\mathcal{Y}}(y)$, which means that it would be impossible to satisfy both of these properties. T4PC requires that the set of properties, P, be defined so as to avoid such situations.

Given a universe of scene graphs, U, one can determine whether the preconditions of two properties, ϕ and ϕ', overlap by evaluating $\exists u \in U : \phi_{\mathcal{X}}(u) \wedge \phi'_{\mathcal{X}}(u)$. Given a distribution, $\mathcal{D}$, of sensor inputs, if U were guaranteed to subsume all scene graphs that could occur, $\forall x \sim \mathcal{D} : \alpha(x) \in U$, then it would be sound to use such an approach. Determining such a U is challenging, moreover existing approaches to reasoning about RFOL do not scale to graphs of the size that we have observed in realistic driving scenes [56].

We approximate the above check by computing the co-satisfaction of precondition pairs evaluated over the dataset:

$$O = \{(i, j) : 1 \leq i < j \leq n \wedge x \in D^* \wedge \phi_{\mathcal{X}}(x)[i] \wedge \phi_{\mathcal{X}}(x)[j]\}$$

A set of properties is *consistent* if $\forall (i, j) \in O : \phi^i_{\mathcal{Y}} \wedge \phi^j_{\mathcal{Y}}$. For a general class of postconditions, including the hyper-rectangular constraints used in this work but generalizing well-beyond that, we can encode the check for consistency as an SMT problem in the theory of linear real arithmetic. Since there are at most $\frac{n^2}{2}$ pairs of matching preconditions and postconditions are not overly complex, this is a very efficient check; for the properties in Table I the total time for all checks is less than 0.01 seconds. Inconsistent pairs of properties are reported to the user who must refine them and restart T4PC.

We note that the lack of soundness of dataset consistency is not a problem for T4PC. This is because the training process, described below, only assumes dataset consistency in computing property loss.

C. Training for Specification Conformance

The purpose of this step is to use our enhanced dataset D^* to train a DNN for specification comformance by incorporating the defined properties into the optimization.

1) Main Masked Loss: When a training instance, (x, y), violates a specification, $\exists i : \neg(\phi_{\mathcal{X}}(x)[i] \implies \phi_{\mathcal{Y}}(y)[i])$, there are several possible solution strategies. For example, one could provide this feedback to the DNN developer and ask them to refine the training instance, similar to our approach for property inconsistency, but this could be expensive for developers. We adopt a more permissive approach that allows for violating training data, but masks the main loss term—since it may bias training away from property conformance—and effectively uses the input, x, to perform a kind of weak-supervision based on property loss.

Given a deep neural network prediction $N(x)$, a data label y, and property labels $\phi_{\mathcal{X}}(x), \phi_{\mathcal{Y}}(y)$, the main masked loss $\mathcal{L}$ is defined:

$$\mathcal{L}_m(N(x), y) = \begin{cases} 0 & \text{if } \exists i : \phi_{\mathcal{X}}(x)[i] \wedge \neg\phi_{\mathcal{Y}}(y)[i] \\ l(N(x), y) & \text{otherwise} \end{cases}$$

where l is a traditional loss function, such as, MSE, MAE, or cross-entropy. If any property is violated by the training pair, (x, y), then we say that the main loss is *masked*.

For example, consider a property *"if there is a red light governing the ego lane, the ego vehicle should stop"*, and a data point for which the input image contains a red light, but its label says that ego should accelerate instead of stop. In this case, the framework will mask the main loss for that data point since the ground truth label for that input is wrong with respect to the property. This step is important because we do not want the DNN to learn behavior that does not conform to the specified properties.

2) Property Loss: When properties are violated, T4PC biases training towards their satisfaction. It takes in the DNN predictions, $N(x)$, the precondition labels $\phi^i_{\mathcal{X}}(x)$ and the postcondition labels $\phi^i_{\mathcal{Y}}(y)$ for each property, and computes a loss for each property, $\mathcal{L}_{\phi^i}$. In this section, we define property loss for a general class of postconditions containing multiple output variables expressed as hyper-rectangular (HR) constraints in the output space of a DNN. This class is sufficient for expressing a range of properties from the recent literature [18], [19], but we note that further generalization of postconditions, e.g., to disjunctions of HR postconditions, and to polyhedral postconditions, could be valuable. For example, handling disjunctions would allow for collision avoidance specifications like: *"if ego's speed is more than 10mph and a vehicle is 4m in front, and no vehicles are to the left or right, then the ego steering should be aggressively left or right"*. Such specifications could be supported by an extension of this framework, but we leave that to future work.

Let N's output space be $\mathbb{R}^m$ and for $y \in \mathbb{R}^m$ let $y[i]$ denote the coordinate value of its ith dimension. A set of m intervals, $\mathcal{I} = \{[l_i, u_i] : i \in [1, m]\}$, forms the basis for defining an HR postcondition, $\phi_y = \prod_{[l,u] \in \mathcal{I}} [l, u]$. Let $\phi_y[i]$ denote the interval for the ith dimension, $[l_i, u_i]$. The distance from a scalar value, v, to a closed interval is:

$$\|v - [l, u])\| = \begin{cases} 0 & \text{if } l \leq v \wedge v \leq u \\ \min(\|l - v\|, \|v - u\|) & \text{otherwise} \end{cases}$$

Lifting this distance to the vector of DNN outputs and associated postcondition intervals gives the property loss:

$$\mathcal{L}_{\phi^i}(N(x)) = \begin{cases} \|N(x) - \phi^i_{\mathcal{Y}}\| & \text{if } \phi_{\mathcal{X}}(x)[i] \wedge \neg\phi_{\mathcal{Y}}(N(x))[i] \\ 0 & \text{otherwise} \end{cases}$$

D. Lagrangian Optimization

As depicted in Figure 2, for each property that is violated a separate loss term is computed, e.g., $\mathcal{L}_{\phi^1}$ and $\mathcal{L}_{\phi^2}$. These losses are combined in order to bias training towards conformance of all properties. This combined loss, $\mathcal{L}_\cap$ in the figure, approximates the distance to the intersection of the active postconditions as depicted in the grey region of the figure. To blend the main masked loss and property loss terms without requiring parameter tuning, T4PC leverages Lagrangian dual optimization [57] to learn the blending hyperparameters, λ_i. Our problem's loss in this framework is:

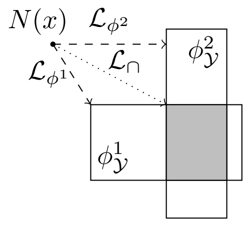


Fig. 2: Property loss (- -), combined loss (. .)

$$\mathcal{L}^*(N(x), y) = \mathcal{L}_m(N(x), y) + \sum_{i=1}^{|P|} \lambda_i \mathcal{L}_{\phi^i}(N(x))$$

The λ_i are initialized to 0 and updated after every training epoch by incorporating the property-specific loss:

$$\lambda_i^{k+1} = \lambda_i^k + \rho_i \sum_{(x,\ldots) \in D^*} \mathcal{L}_{\phi^i}(N(x))$$

The hyperparameter ρ_i is the Lagrangian update step which can be set for each property to control the rate of change of λ_i. Property loss weights are initially zero, $\lambda_i^0 = 0$. If a property, ϕ_i, is violated then its weight, λ_i, is updated in proportion to the cumulative loss within current training epoch moderated by ρ_i. This allows the optimization process to adapt during training to determine how to weight property loss—to achieve property conformance—while avoiding overweighting it—which might reduce DNN accuracy.

E. Limitations

T4PC enables training a DNN to improve property conformance. However, the approach for T4PC is limited in the expressiveness of the properties that can be leveraged for training. In order to apply the property loss regime specified in Section III-C2, the property must be specified over a single input-output pair describing the expected behavior in response to each input. For example, a property could say *"if there is a green light, acceleration must be positive"* as this is evaluated over a single input (whether there is a green light in the abstract input representation) and a single output (whether the resultant acceleration is positive). These properties are described formally in RFOL. This allows T4PC to assign the attendant property loss to each input to guide the training through backpropagation. However, autonomous vehicles are bound by richer temporal properties that span multiple inputs and outputs; for example, *"the vehicle must stop at the stop sign before continuing"*. This property does not describe the expected output of the DNN at any particular time step—whether the vehicle should accelerate or decelerate depends on whether the vehicle has already stopped or not. This is not directly expressible in RFOL and requires a form of temporal logic [19]; this paradigm is not amenable to ascribing a particular loss based on a single input-output pair. Although Section V-B demonstrates how T4PC can approximate such properties during training to improve conformance at runtime, future work should investigate methods to expand T4PC to directly leverage these richer temporal properties.

F. Implementation

We instantiated and used T4PC for the following controlled experiments. We briefly describe 3 implementation details. For the abstraction we used the same scene graphs (SG) and scene graph generators used in previous work [18], [19] whose structure is explained in Section III-B2. To evaluate the properties over the SGs we utilize the scene graph language (SGL) [19] to formally specify the properties and check them over scene graphs. Finally, the main masked loss, property loss and Lagrangian dual optimization were implemented in Python using PyTorch [58]. For the family of controlled experiments we designed an end-to-end training pipeline that includes these losses and optimization, while for the case study in Section V, we incorporate the losses in two AV systems training pipeline. We make our implementation available on https://github.com/less-lab-uva/T4PC [Archive].

IV. Controlled Experiment

To study T4PC's ability to increase property conformance of a DNN we set the following research questions:

RQ1. How effective is T4PC at training a DNN from scratch to conform to varying numbers of properties? We explore T4PC's performance when applied to train a series of models towards the conformance of property sets of different sizes.

RQ2. How efficient is T4PC in training a DNN to conform to multiple properties? We explore how the number of properties targeted affects the training time, as well as the trade-offs between training time and performance gains when training for different periods.

RQ3. How effective is T4PC at fine-tuning a DNN to conform to single properties?

We defer studying the treatment combination of fine-tuning for multiple properties to the case study in Section V.

To answer these research questions, we first introduce the *dataset* used for training and evaluation, followed by the *navigation DNN* used in our experiment. Second, we present a detailed description of the *properties* studied and describe our *experimental design* for both training from scratch and fine-tuning. Finally, we outline the *metrics* employed to assess T4PC's effectiveness in achieving property conformance.

A. Dataset

To ensure high-quality data collection, we utilized TCP [59], an end-to-end AV system (that we later study as a whole in Section V), currently ranked number 3 in the CARLA leaderboard challenge [60], which evaluates the driving proficiency of autonomous agents in realistic traffic scenarios. We collected a dataset of **400,487** frames each containing an image, steering angle, acceleration, and scene graph using the CARLA [61] simulator. Following the procedure defined by the TCP system, we collect data from 6 CARLA environments, also known as "Towns", that cover a range of driving landscapes, from urban to rural, including single and multi-lane roads, and various weather and lighting conditions. For each town, the TCP procedure executes many routes, where each route contains a series of waypoints that the ego vehicle must follow until reaching the destination. Depending on the town, between 321 and 480 routes are collected. The entire dataset, over 137 GB, is available to download in our repository.

During the data collection phase, we use a plugin [42] that interfaces with the CARLA Python API to generate ground-truth scene graphs (our abstraction α)[1], adopting the SG parameterization used in prior work [18], [19]. We use ground-truth SGs to enable analysis of T4PC independent of noise or faults in scene graph generators over sensor data; however, the current pace of research in scene graph generation [62] is encouraging for the future application of T4PC in practice.

B. AV Navigation DNN

We focus on a DNN that can be used by an AV to navigate in an environment. To facilitate experimentation, we trained our own DNN so we could manipulate the properties that we optimize for and use different data. Note that the next section complements this controlled setup by applying T4PC to two real systems in a simulated environment. The AV will capture images through its front camera and feed them to the DNN; the DNN will output a steering angle and acceleration signal that the AV will use to move around. Both DNN outputs are normalized to the range $[-1, 1]$. For acceleration, a value of -1 is the strongest signal for braking, while a value of 1 is the strongest signal for accelerating. For steering angle, these values represent a range between -70 and 70 degrees. Similar to prior work using SG for AVs [18], we leverage a pre-trained ResNet34 backbone since it is used by the top 3 ADS [59], [63], [64] in CARLA leaderboard, though with a modified output layer to predict acceleration and steering angle.

[1]Data from 2 out of 8 towns were not collected due to incompatibilities between the CARLA versions used by TCP and by our instrumentation plugin.

C. Properties

We define six safe driving properties for this experiment. Their English description and logic formula for their pre and postconditions are shown in the first part of Table I (ϕ_1–ϕ_6). We do not include ϕ_7 and ϕ_8 as their preconditions require the ego vehicle speed, and the AV navigation DNN used in this experiment does not take that as input (we consider them in Section V). The first 4 properties state a postcondition in terms of the model's acceleration output, while the following 2 properties state a postcondition in terms of the model's steering angle output. Among the 4 acceleration properties, ϕ_1 and ϕ_2 expect a negative acceleration as they enforce a stop action, while ϕ_3 and ϕ_4 expect a positive acceleration.

D. Experimental Design

We perform three distinct experiments to answer each research question. For each question we train two types of DNNs: base DNNs ($\mathcal{B}$) that only include the main loss function computed from the data labels, and treatment DNNs ($\mathcal{M}$) using T4PC that also account for property conformance. Since we do not explore augmentation in this experiment, we have no metamorphic functions to be defined. All the DNNs, $\mathcal{B}$ and $\mathcal{M}$, throughout the three research questions are based on the architecture described in Section IV-B, trained for 15 epochs[2], using batches of 256 images, with the same dataset. We use a Stochastic Gradient Descent (SGD) optimizer, a constant learning rate (lr) of 10^{-4}, and a ρ for the property losses of 0.1. For training every DNN, we used 8 CPU cores, 128G of RAM, and 1 A100 GPU.

To account for the variation across towns and routes, we implemented leave-one-out cross-validation, leading to 6 splits with 5 towns used for training and validation, and 1 town used for testing[3]. We train and evaluate a DNN for each split, leading to 6 DNNs per type. The metrics are reported on the aggregate of the test results obtained from each split.

E. Metrics

We define 2 types of metrics: *violation improvement*, and *loss improvement*, using percentages[4]. These metrics represent the difference between the property violations (V_{imp}), the steering angle (L^s_{imp}), and acceleration (L^a_{imp}) loss of the baseline, $\mathcal{B}$, and the DNN that uses property loss, $\mathcal{M}$. For each of the metrics, a higher score indicates an improvement of performance, with a value of 100.00% meaning that $\mathcal{M}$ has removed all violations present in $\mathcal{B}$. Higher values of V_{imp} indicate that T4PC is effective at improving property conformance. Note that while higher values of L^{type}_{imp} are better, a 0.00% would indicate that T4PC did not decrease the performance of the baseline DNN with respect to the original loss. Formally:

$$V_{imp} = \frac{\sum_{i=0}^{|Splits|} v(B^i) - \sum_{i=0}^{|Splits|} v(M^i)}{\sum_{i=0}^{|Splits|} v(B^i)} \times 100$$

$$L^{type}_{imp} = \frac{\sum_{i=0}^{|Splits|} \mathcal{L}_{type}(B^i) - \sum_{i=0}^{|Splits|} \mathcal{L}_{type}(M^i)}{\sum_{i=0}^{|Splits|} \mathcal{L}_{type}(B^i)} \times 100$$

Where v is a function to obtain the number of violations for a given DNN. To aggregate the results from the 6 split cross-validation, we sum the number of violations/loss across the splits. These sums are used for computing V_{imp}, L^s_{imp}, and L^a_{imp} respectively.

F. RQ1-Training towards property conformance

1) Setup specific to experiment: We train 6 baseline $\mathcal{B}$ DNNs (one per split). We also train with T4PC for each of the 6 splits using varying combinations of the first 6 properties in Table I. This results in varying numbers of $\mathcal{M}$ DNNs, based on combinations of the 6 properties. Specifically, choosing 1, 2, 4, or all 6 properties yields $\binom{6}{1} = 6$, $\binom{6}{2} = 15$, $\binom{6}{4} = 15$, and $\binom{6}{6} = 1$ combinations, respectively. Each combination is evaluated across 6 data splits, resulting in a total of 36, 90, 90, and 6 $\mathcal{M}$ DNNs, respectively.

2) Results: The first four subfigures of Figure 3 show the results for the treatments (instances of 1 prop, 2 props[5], 4 props[5], and 6 props[6]). The x-axis measures the main loss improvement for steering (●) and acceleration (×). The y-axis measures the violation improvement where 0.00% means that $\mathcal{B}$ and $\mathcal{M}$ cause the same number of violations, and a value of 100.00% means that, different from $\mathcal{B}$, $\mathcal{M}$ does not violate the property. Negative values in either axis means that DNN $\mathcal{M}$ has either a higher loss or more violations than its counterpart $\mathcal{B}$. Ideally, $\mathcal{M}$ will result in observations at the top (violation improvement > 0) and center-right (loss improvement ≥ 0) of each figure—reducing violations but retaining the original performance of $\mathcal{B}$. To ease the reading of Figure 3, we used dashed lines to represent acceleration properties and solid lines to represent steering angle properties, and we included the total number of violations from $\mathcal{B}$ and $\mathcal{M}$ respectively.

We first focus on training for one property. Starting from the top of the graph in Figure 3 (1 prop), when T4PC is applied to conform to ϕ_5, $\mathcal{M}$ achieves a 100.00% violation reduction over $\mathcal{B}$ (from 5,912 violations to 0), meaning that it was able to learn not to violate the property unlike the base DNN. Applying T4PC to the first four properties (acceleration properties), shows a range of gains from 79.66% to 34.10%, depending in part on the DNN's capability to identify the property precondition features. For example, features such as detecting traffic light by color (ϕ_2, ϕ_3, ϕ_4) are readily identified by models like Detectron2 [65]. In contrast, distance features like entities within X meters (ϕ_1 and ϕ_3), are hard to estimate by using a monocular camera [66]. The low improvement for ϕ_3 may be due to the DNN's difficulty in recognizing whether a car is within 25m in the same lane as ego. More than 83% of the violations occur in the 6th split where Town 10 is used for testing. This town contains parked cars that can confuse the DNN, causing it to fail to satisfy the

[2]We show why 15 epochs is adequate as part of Section IV-G1.

[3]The number of samples in each split/town can be found in the appendix.

[4]Violations per property and split are available in the appendix.

[5] Since the 2 props and 4 props sets contain 5 and 10 more property combinations than 1 prop, we scale up the number of $\mathcal{B}$ violations (number to the left of the colored arrows) to serve as a baseline.

[6]Since we train a single DNN to satisfy the 6 properties simultaneously, the main loss differences (dots and crosses) are the same for all properties.

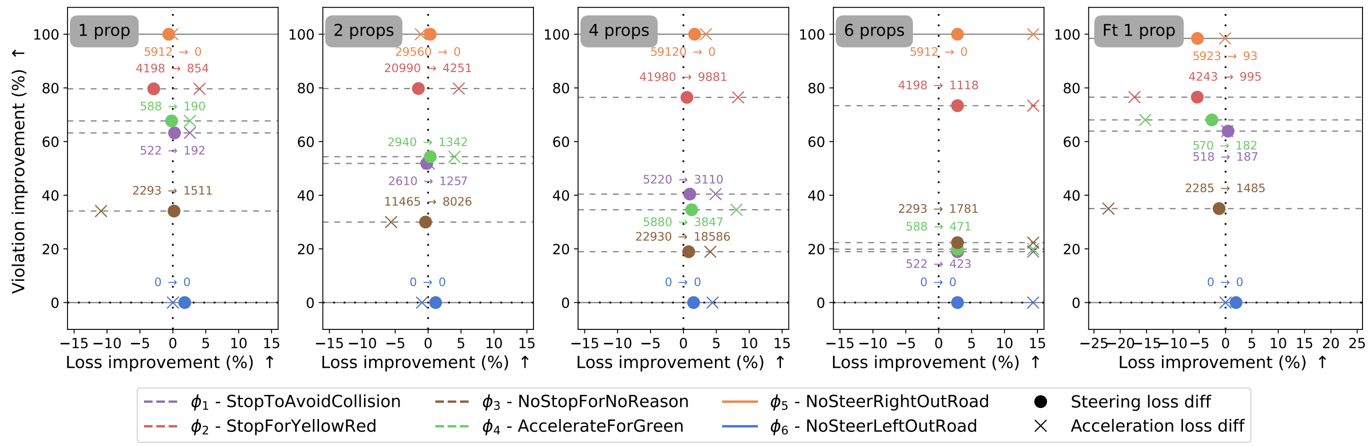


Fig. 3: Experiment results when training from scratch with 1, 2, 4, and 6 properties, and fine-tuning with 1 property.

property thus decreasing conformance. At the bottom of the graph we note that T4PC was not able to provide gains for ϕ_6 (0.00%) as $\mathcal{B}$ already does not violate the property.

Beyond violation improvements, the mostly central position of the dots and crosses indicate that the main losses for most properties do not vary much from the baseline. The average loss improvement percentage for both losses is -0.27. However, the reduction in violations for ϕ_3 led to more noticeable negative loss improvements, likely due to the DNN's inability to identify preconditions, thus biasing predictions towards positive accelerations. The experiment in Section V explores this further by deploying such DNNs in simulation.

The trends observed when improving conformance of single properties extend mostly to multiple properties. As we increase the number of properties included in the optimization (2, 4, and 6 props in Figure 3), we observe that violation reductions achieved by T4PC for individual properties decrease, while the main losses improve. This trend can be attributed to the increased competition among properties: when multiple constraints are optimized together, their effects on the network's outputs can counterbalance one another. For example, some properties may encourage acceleration while others promote deceleration, or may favor steering in opposite directions. This interplay among property objectives balances the network's behavior, leading to improved performance on the main losses.

We also note that the properties that perform the best have the largest amount of data satisfying their preconditions, have preconditions with easier-to-detect features, or both. For example, ϕ_5 and ϕ_2—which have around 210k and 120k datapoints per split satisfying the preconditions, respectively—exhibit higher violation reductions than other properties. This is not only due to their broader coverage in the dataset, but also because their preconditions are simple to identify: ϕ_5 involves detecting a curve to the right, which implies being the rightmost lane, and ϕ_2 corresponds to the presence of a yellow or red traffic light. Although ϕ_4 has the fewest datapoints satisfying its preconditions (23k), it still performs relatively well, likely because its precondition—detecting a green traffic light—is easy for the network to recognize. In contrast, ϕ_3 has 57k datapoints satisfying its precondition but performs worse, likely due to the greater difficulty of detecting entities within 25 meters, as discussed earlier in this section. This imbalance in both coverage and precondition complexity helps explain which properties benefit most from the optimization.

Despite the competition, we highlight that none of the properties experiences a negative violation improvement, demonstrating that even with multiple properties pulling the model in different directions, the optimization process yields consistent safety benefits without introducing regressions.

T4PC is able to **train** DNNs optimizing for **multiple properties** at the same time, reducing property violations and in some cases improving the main losses.

G. *RQ2-Training-time efficiency with multiple properties*

To evaluate the efficiency of T4PC, we consider two dimensions: (1) the training-time overhead incurred when targeting an increasing number of properties simultaneously, and (2) the trade-offs between training time and performance gains as models are trained for longer periods.

1) Efficiency in Terms of Training Time: We assess how well T4PC scales by measuring the per-batch training cost as the number of properties increases. To do so, during the evaluation of RQ1, we instrumented the training phase to record the time taken for each batch, including the forward pass, the calculation of the original loss and, for T4PC, the property loss, as well as the backpropagation step to update the DNN weights.

The box plots in blue (left) in Figure 4 show the distribution of time taken per batch (measured in milliseconds in the left-axes), while the black dots show the median time taken to train 15 epochs (measured in minutes in the right-axes), all across the number of properties targeted during training. For the baseline models, targeting zero properties, the median batch time is 232ms. When targeting one property, the median batch time is 612ms due to the overhead incurred by T4PC; although substantial in relative terms, the successful application of T4PC through the other RQs demonstrate its feasibility. Increasing to 2, 4, and 6 properties incurs only a

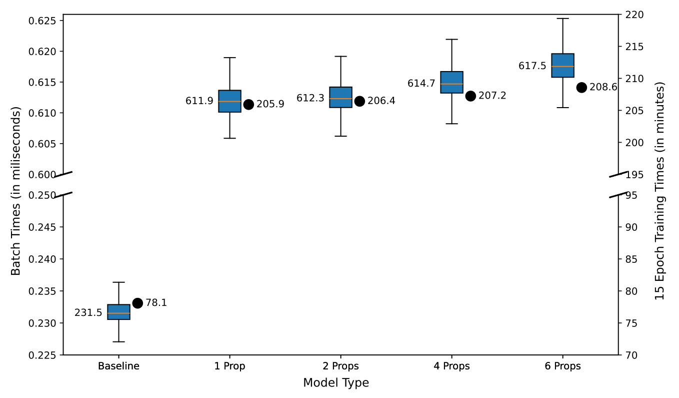


Fig. 4: Time to train a batch (blue, left-axes) and median time to train for 15 epochs (dot, right-axes), across property sets.

minor increase in the median time on the order of a millisecond per additional property, or less than 1%. Through the black dots we can see how these values scale when training the different models for 15 epochs of 1,351 batches each. Neither the box plots nor the black dots include the I/O time of loading images and labels which adds approximately 100 minutes with high variability to both baseline and T4PC. In this context, the overhead of T4PC is practical and similar to the overhead of other parts of the training process.

2) Cost-effectiveness of Training Longer: To examine the cost-effectiveness of training over extended durations, we trained a model with property loss for 72 hours—the maximum time allowed per GPU partition on our SLURM server—using the same hyperparameter and hardware settings described in Section IV-D. We focused on the most challenging setting: optimizing the first six properties from Table I, completing 156 epochs within the time limit. To account for variance, we repeated the experiment three times using the same data split, resulting in three trained models.

Figure 5 shows the average total number of property violations computed over the 3 trained models during validation, with a 2 standard deviation interval shaded in gray. At epoch 0, there are 10,099 violations. By epoch 15, the number of violations is 742 (a 92.7% reduction), and by epoch 63, it drops to 364 (a 96.4% reduction)—the minimum average of total violations across all epochs. This additional 3.7 percentage point improvement comes at a significant computational cost: training for 15 epochs takes approximately 5 hours, whereas 63 epochs—required to achieve the additional gain—take 21 hours, representing over 300% more training time.

T4PC **scales efficiently** to multiple properties, adding only minimal overhead per batch, and a significant percentage of the gains are obtained early in training.

H. RQ3-Fine-tuning towards property conformance

1) Setup specific to experiment: To answer this question, we need to fine-tune existing DNNs. We use the 6 baseline DNNs ($\mathcal{B}$) from Section IV-F as a starting point, fine-tuning each for 15 additional epochs on the same data split and dataset. The fine-tuning process using only the loss function computed from the data labels results in 6 fine-tuned baselines, $\mathcal{B}_{ft}$, while the fine-tuning process using T4PC towards conformance for 6 properties renders 36 new DNNs, $\mathcal{M}_{ft}$. For fine-tuning the DNNs we use a smaller lr of 10^{-5}.

2) Results: Figure 3 with "Ft 1 prop" label summarizes the results. Overall, the violation improvements show a similar trend as in Section IV-F. All but one property shows clear violation reductions, with the only property not seeing gains (ϕ_6) having the baseline DNN already achieving 0 violations. ϕ_5 achieves a 98.43% violation improvement, and the other 4 properties show between 35.01% and 76.55% violation improvement. For the properties involving acceleration, except for ϕ_1, violation improvement comes at the cost of main loss decreases that are more noticeable than when training the DNN towards property conformance from scratch (shown in Section IV-F). For example, the DNN acceleration loss for ϕ_4 went from 2.59% when training from scratch to -15.27% when fine-tuning towards conformance. We conjecture that the reduction in main loss arises from the base DNNs trained with data labels having achieved a local minimum with respect to the main loss. Thus, fine-tuning those DNNs for conformance to those properties is likely to have moved them away from that minimum. In such cases, the engineer will have to analyze the effect of increases in main loss in light of the decrease in violations. In Section V we deploy a navigation DNN in simulation to assess the true impact of such losses.

T4PC is able to **fine-tune** existing DNNs, reducing number of violations, albeit with more noticeable differences in main loss, showing its effectiveness at reducing violations of existing highly trained DNNs.

I. Threats to Validity of Findings

The validity of our findings across the family of experiments suffers from 3 key threats.

First, the starting navigation DNN, the collected dataset, and the properties selected allow us to conduct a family of studies where we can manipulate the conditions under which T4PC is applied in order to answer multiple research questions. However, the external validity of this selection limits the reach of our findings. The DNN architecture was the same as used in prior work [18], but more complex architectures and parameterization may offer different tradeoffs when training for conformance. The collected dataset also came from CARLA, following the same procedures and environments employed by similar systems [59]. Yet, differences in sensors, their fidelity, and their rates, for example, may impact T4PC. The selection of properties was inspired by work encoding driving rules [19] that were relevant to the chosen navigation DNN. Although the 6 properties we use are representative of many others, there are still many properties that would require a more expressive abstraction and language to be incorporated into T4PC.

Second, the implementation of T4PC involves many components, from the scene graph generator to the Lagrangian optimization, of considerable complexity and with many con-

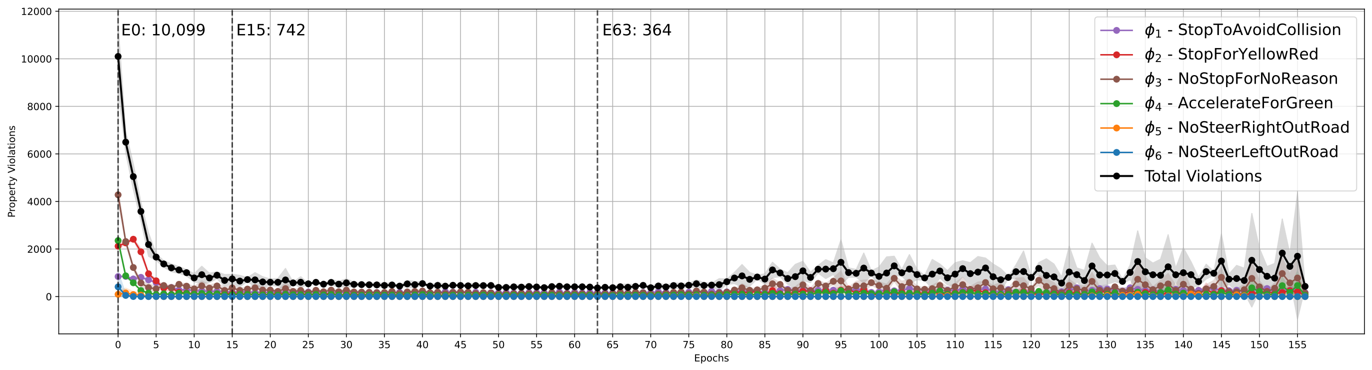


Fig. 5: Average validation property violations with 2 std interval when training for 72 hs.

figurable parameters. Although we have tested those components in different configurations, they may still contain faults. We make them available, as well as raw and intermediate data, for the community to reuse and assist in improving them at https://github.com/less-lab-uva/T4PC [Archive].

Third, from a construct validity perspective, in these experiments the DNNs trained for conformance are judged against a validation set in terms of property violations and main loss. This assessment is limited in that it computes those metrics against an evaluation dataset and the impact of changes in the main loss metric in the absence of an enclosing system context. We address this concern in the next section by applying T4PC to a real autonomous system, running it in a simulated environment that provides a range of driving metrics.

V. Case study in Simulation

This study aims to broaden the T4PC effectiveness assessment from Section IV along 4 dimensions: 1) target two AV systems developed by third parties with different architectures and consuming multiple inputs; 2) fine-tune each system for multiple property conformance; 3) deploy the systems in a series of CARLA [61] simulated environments; and 4) judge the systems using CARLA's driving assessment measures.

A. Target Systems and Execution Environment

We target the TCP [59] and Interfuser [63] AV systems, two of the three best performing in the CARLA Leaderboard competition [60]. These systems were selected due to their strong performance and architectural diversity. TCP is a fully end-to-end DNN, while Interfuser combines a DNN with a rule-based module written in Python. Since T4PC requires the entire system to be differentiable, we approximate the relevant portions of Interfuser's coded module with a DNN during training to enable the use of our property-based loss function.[7] TCP takes in a single image from an RGB camera, the ego's speed, and the target waypoint. Its architecture includes a ResNet34 [67] image encoder, and two GRU [68] branches for steering angle and acceleration predictions. In contrast, Interfuser takes in images from 3 RGB cameras, a LiDAR point cloud, the ego's speed, and the target waypoint. It features a ResNet50 image encoder and a ResNet18-based LiDAR encoder. These features are fused using a transformer encoder-decoder, which outputs ten future waypoints, a semantic feature map, and traffic-related information, including traffic light status, stop sign presence, and intersection detection. These outputs are then passed to a coded controller, which produces the final steering angle and acceleration commands. As we only target acceleration-based properties for Interfuser (explained in Section V-B), we only need to approximate the acceleration portion of the controller. To do so, we collected the inputs it consumes and the acceleration outputs it produces over the same dataset, and trained a DNN that achieved 0.04 Mean Absolute Error (more details available in the appendix). Both systems produce a steering angle in the range $[-1, 1]$, representing -70 and 70 degrees, and acceleration in the range $[-1, 0.75]$. We use publicly available pretrained weights from the respective repositories. For evaluation, both AV systems are deployed in simulation on ten routes in Town 05, which are excluded from training and validation. This town includes a variety of roads (e.g., 2-lane roads, 3 and 4-lane highways, and T intersections) and a variety of entities (e.g., traffic lights, stop signs, pedestrians, cyclists, cars, and trucks).

B. Properties

Despite its high ranking, we observed that TCP struggled to comply with stop signs. To address this, we defined two properties (ϕ_7 and ϕ_8 in Table I) targeting stop sign behavior. ϕ_7 enforces stopping at a stop sign, while ϕ_8 promotes resuming motion once a full stop has occurred. The preconditions of these two properties differ slightly; ϕ_8 includes a constraint requiring no entity to be within 7 meters ahead, ensuring safe acceleration, while ϕ_7 does not. We adopt a 0.1 m/s speed threshold—used by CARLA's low-level controller—to define when the vehicle is considered "stopped." If a stop sign controls the ego lane and speed exceeds this threshold, the vehicle should decelerate; otherwise, it should accelerate.

Interfuser exhibited violations such as running red lights or colliding with pedestrians and other vehicles. These failures align closely with the behaviors targeted by several of the properties used in the controlled experiment. As a result, we fine-tune Interfuser with properties ϕ_1–ϕ_4 from Table I.

[7]In principle, any coded module could be approximated using a data-driven method such as a DNN. However, such approximations become more difficult and require more data and sophisticated architectures as the module complexity increases or incorporates internal states.

TABLE II: TCP and Interfuser results. Values in bold are statistically significantly better.

(a) **TCP** scores and infractions

Treatment	Driving Score $\uparrow$	Collision Vehicles $\downarrow$	Stop Sign Infraction $\downarrow$	Vehicle Blocked $\downarrow$
T4PC 10%	81.09±6.58	0.13±0.17	**0.54±0.20**	0.09±0.09
Base 10%	79.20±4.34	0.12±0.14	0.98±0.22	**0.00±0.00**
T4PC 15%	**81.98±5.16**	0.16±0.13	**0.59±0.18**	0.03±0.05
Base 15%	78.37±3.48	0.11±0.09	1.02±0.17	0.01±0.03
T4PC 20%	**81.81±4.77**	**0.10±0.10**	**0.76±0.23**	0.02±0.04
Base 20%	75.99±3.31	0.16±0.10	1.06±0.19	0.01±0.03

(b) **Interfuser** scores and infractions

Treatment	Driving Score $\uparrow$	Collision Pedestrians $\downarrow$	Red Light Infraction $\downarrow$	Route Timeout $\downarrow$
T4PC 10%	58.40±9.22	0.21±0.13	0.42±0.11	0.20±0.15
Base 10%	59.44±8.33	0.19±0.12	0.44±0.12	0.22±0.11
T4PC 15%	63.90±8.33	0.16±0.10	**0.24±0.10**	0.17±0.14
Base 15%	59.45±8.35	0.15±0.11	0.43±0.10	0.21±0.15
T4PC 20%	62.48±9.16	**0.17±0.10**	**0.19±0.08**	0.23±0.17
Base 20%	60.15±6.92	0.28±0.15	0.42±0.11	**0.14±0.10**

While these properties are relatively simple, they reflect the single-frame nature of TCP's and Interfuser's perception models. We leave exploring richer, temporal properties that better capture the full space of traffic rules for future work.

C. Dataset

For training TCP with T4PC, we reused the dataset from the controlled experiments described in Section IV, which includes single RGB images, ego speed, and target waypoints, along with the labels required by TCP [69]. Unlike the original TCP that used 4 towns for training and 4 for validation, we used 3 towns for training (1, 4, and 10), and 3 for validation (2, 5, and 7) because we collected 6/8 towns as explained in Section IV-A. For training Interfuser with T4PC, we collected a new dataset using its official data collection scripts in CARLA. The dataset contains **138,585** frames, each with 3 RGB images, a LiDAR point cloud, steering angle, acceleration, and a scene graph. Data was collected across 6 CARLA towns, and we followed Interfuser's standard procedure of executing multiple routes per town, where each route guides the ego vehicle through a sequence of waypoints. Following the original Interfuser, we used 5 towns (1, 2, 4, 7, and 10) for training[8], excluding two due to collecting data from only 6 of the 8 towns, as discussed in Section IV-A, and 1 town for validation (5). Details on the data augmentation used during training are provided in the appendix.

D. TCP Application

We fine-tune the TCP and Interfuser DNNs released by the authors with and without T4PC[9]. To fine-tune the DNNs, we used the same hyperparameters defined by the authors of TCP and Interfuser. We fine-tuned the DNNs for 5, 10, and 15 epochs for TCP and 3, 5, and 7 epochs for Interfuser, corresponding to approximately 10%, 15%, and 20% of their original training schedules of 60 and 35 epochs, respectively. The decision of 10%, 15%, and 20% matches what is shown in Figure 5, where we picked 15 epochs to strike a balance between performance and cost which represents 10% of the 155 epochs. The two greater percentages are meant to sample from the region where better results appeared without incurring significant training costs. More details about the hyperparameters and hardware are provided in the appendix.

[8]We limit data collection to only the default weather configuration as changing weather is not part of any property.

[9]Fine-tuning the DNNs without T4PC allows us to refine it with the same data we collected from CARLA.

After fine-tuning, we evaluate the DNNs using the 10 evaluation routes described in Section V-A. To account for variation in training, we train 5 DNNs with T4PC and 5 DNNs without T4PC, using the 3 different number of epochs, for a total 30 DNNs per AV system. Likewise, to account for variation during the evaluation of the DNNs in simulation, we ran each DNN in CARLA 5 times. We then group the results by epochs trained and whether it was trained with T4PC or not, resulting in 25 data points (5 DNNs by 5 runs), and aggregated them by taking the mean and standard deviation. Of the several metrics CARLA calculates, we report the overall driving score and the average infraction scores that include a statistically significant difference for each system[10].

E. Results

Table II provides a summary of the results for TCP (left) and Interfuser (right), with the rows alternating between optimization with T4PC and the baseline, for the different percentages of training epochs. The bold values are significantly better, determined by a t-test with p-value of 0.05.

Table IIa shows that TCP's stop sign infractions, the one that we aim to reduce, decreased significantly (bold) with T4PC, by 38%. Meanwhile, the driving score column shows that T4PC also increased the driving score in all cases, on average by 5%, with the improvements by the DNNs trained with 15% and 20% of the epochs deemed significant. The other two columns provide insight into the side-effects of optimizing for ϕ_7 and ϕ_8 on other types of infractions (collisions with vehicles and vehicles blocked). For models trained with 10% of epochs, we find that prioritizing these properties leads to a significant increase in vehicle blocked infractions. However, when training for 20% of epochs, T4PC renders similar results for the vehicle blocked infractions, while significantly reducing infractions from collisions. This suggests that training TCP for more epochs allows it to perform significantly better, while not making other infractions worse. Overall, these results show that the TCP models trained with T4PC using the 2 properties defined above can significantly reduce the number of stop sign infractions and slightly increase the CARLA driving score, a tall order given that TCP is one of the leaders in the CARLA leaderboard.

Table IIb shows that Interfuser's red light infractions are significantly reduced with T4PC when training for 15% and

[10]Other metrics did not lead to statistically significant differences. Tables with all metrics available in appendix.

20% of epochs, and pedestrian collisions are significantly reduced when training for 20% of epochs, both properties targeted by T4PC. The driving score improves with models trained with 15% and 20% of epochs using T4PC, though the changes are not significant, and the route timeout infractions are significantly worse for models trained with 20% of epochs. This could be explained by two main factors. First, the properties may not be specific enough, e.g., ϕ_3 is designed to make the ego vehicle move if there is nothing in front and in the same lane but does not consider cases where multiple lanes overlap (intersections), not helping Interfuser move in those situations. Second, the DNN approximation we trained for mimicking the Interfuser's Python component, although 95% accurate, is not perfect and can introduce noise when training the Interfuser DNN. Further, there may exist latent faults within Interfuser's Python component that are beyond the reach of T4PC's ability to improve the system's performance through training. Although T4PC was less effective overall at significantly decreasing Interfuser's violations as compared to TCP's, these results indicate T4PC's ability to provide benefit for heterogeneous architecture systems as well.

T4PC is able to **fine-tune two existing open-source AV systems** optimized for the CARLA competition, **decreasing the number of infractions**, and improving their overall driving score.

F. Threats to Validity of Case Study Findings

This case study has similar threats to the previous experiments. Although it reduces the external validity and construct threats by utilizing two third-party DNNs, deploying them in the widely used CARLA simulation environment, and using the environment's builtin metrics to judge performance. Still, the simulation environment has limitations as the findings may not translate to real deployed systems equipped with DNNs, limiting the generalization to real-world applications.

VI. CONCLUSION

We have introduced T4PC, a complementary approach to existing V&V techniques that instead of checking for property conformance after a DNN is trained, directly trains a DNN towards high-level property conformance. The approach is novel in its use of SGs as sensor input abstractions for evaluating safe driving property preconditions, and its integration of property loss into the optimization process. Our experiments and case study provide evidence about T4PC's potential to reduce property violations for AV systems, while maintaining main loss. In future work, we aim to broaden the type of properties to consider, extend T4PC to handle more complex multi-module systems, compare and investigate the interaction with runtime enforcement approaches, and study its applicability in other domains like factory or surgical robots.

VII. ACKNOWLEDGMENTS

This research is supported in part by NSF awards 2312487 and 2403060; by ARO grant W911NF-24-1-0089; by Lockheed Martin Advanced Technology Labs; and by Copenhaver Charitable Trust. The authors acknowledge Research Computing at The University of Virginia for providing access to computational resources.

REFERENCES

[1] F. Lambert, "Tesla finally releases fsd v12, its last hope for self-driving," Jan 2024, accessed on 08.20.2024. Link.

[2] A. J. Hawkins, "Inside waymo's strategy to grow the best brains for self-driving cars," May 2018, accessed on 08.20.2024. Link.

[3] comma.ai, "How openpilot works in 2021," Oct 2021, accessed on 08.20.2024. Link.

[4] A. Marshall, "Uber video shows the kind of crash self-driving cars are made to avoid," Mar 2018, accessed on 07.25.2025. Link.

[5] A. Roy and H. Jin, "California regulator probes crashes involving gm's cruise robotaxis," Aug 2023, accessed on 07.25.2025. Link.

[6] R. Bellan, "A waymo self-driving car killed a dog in 'unavoidable' accident," Jun 2023, accessed on 07.25.2025. Link.

[7] T. Thadani, "Cruise recalls all its driverless cars after pedestrian hit and dragged," Nov 2023, accessed on 07.25.2025. Link.

[8] G. Katz et al., "The marabou framework for verification and analysis of deep neural networks," in *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*. Springer, 2019, pp. 443–452.

[9] D. Shriver, S. Elbaum, and M. B. Dwyer, "Dnnv: A framework for deep neural network verification," in *International Conference on Computer Aided Verification*. Springer, 2021, pp. 137–150.

[10] H. Zhang, S. Wang, K. Xu, L. Li, B. Li, S. Jana, C.-J. Hsieh, and J. Z. Kolter, "General cutting planes for bound-propagation-based neural network verification," *Advances in neural information processing systems*, vol. 35, pp. 1656–1670, 2022.

[11] H. Duong, D. Xu, T. Nguyen, and M. B. Dwyer, "Harnessing neuron stability to improve dnn verification," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 859–881, 2024.

[12] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*, 2018.

[13] T. Zohdinasab et al., V. Riccio, and P. Tonella, "Deepatash: Focused test generation for deep learning systems," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 954–966.

[14] L. Wang, X. Xie, X. Du, M. Tian, Q. Guo, Z. Yang, and C. Shen, "Distxplore: Distribution-guided testing for evaluating and enhancing deep learning systems," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 68–80.

[15] S. Dola, R. McDaniel, M. B. Dwyer, and M. L. Soffa, "Cit4dnn: Generating diverse and rare inputs for neural networks using latent space combinatorial testing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[16] K. Viswanadha, E. Kim, F. Indaheng, D. J. Fremont, and S. A. Seshia, "Parallel and multi-objective falsification with scenic and verifai," in *Runtime Verification: 21st International Conference, RV 2021, Virtual Event, October 11–14, 2021, Proceedings 21*. Springer, 2021.

[17] F. Toledo, D. Shriver, S. Elbaum, and M. B. Dwyer, "Deeper notions of correctness in image-based dnns: Lifting properties from pixel to entities," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2122–2126.

[18] T. Woodlief, F. Toledo, S. Elbaum, and M. B. Dwyer, "S3c: Spatial semantic scene coverage for autonomous vehicles," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

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

[20] J. Zhang and J. Li, "Testing and verification of neural-network-based safety-critical control software: A systematic literature review," *Information and Software Technology*, vol. 123, p. 106296, 2020.

[21] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, and X. Yi, "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability," *Computer Science Review*, vol. 37, p. 100270, 2020.

[22] M. Diligenti, S. Roychowdhury, and M. Gori, "Integrating prior knowledge into deep learning," in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2017, pp. 920–923.

[23] J. Xu et al., "A semantic loss function for deep learning with symbolic knowledge," 2018.

[24] M. Fischer et al., "DL2: Training and querying neural networks with logic," in *Proceedings of the 36th International Conference on Machine Learning*, vol. 97. PMLR, 09–15 Jun 2019, pp. 1931–1941.

[25] K. Ahmed, K.-W. Chang, and G. Van den Broeck, "Semantic strengthening of neuro-symbolic learning," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2023, pp. 10 252–10 261.

[26] D. Shriver, S. Elbaum, and M. B. Dwyer, "Reducing dnn properties to enable falsification with adversarial attacks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021.

[27] B. Aminof, A. Murano, S. Rubin, and F. Zuleger, "Verification of asynchronous mobile-robots in partially-known environments," in *PRIMA 2015: Principles and Practice of Multi-Agent Systems*. Cham: Springer International Publishing, 2015, pp. 185–200.

[28] C. Morse, L. Feng, M. Dwyer, and S. Elbaum, "A framework for the unsupervised inference of relations between sensed object spatial distributions and robot behaviors," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 901–908.

[29] H. Khosrowjerdi and K. Meinke, "Learning-based testing for autonomous systems using spatial and temporal requirements," in *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*. New York, NY, USA: Association for Computing Machinery, 2018, p. 6–15.

[30] T. Reinbacher, M. Függer, and J. Brauer, "Runtime verification of embedded real-time systems," *Formal methods in system design*, 2014.

[31] S. Pinisetty, P. S. Roop, S. Smyth, N. Allen, S. Tripakis, and R. V. Hanxleden, "Runtime enforcement of cyber-physical systems," *ACM Transactions on Embedded Computing Systems (TECS)*, 2017.

[32] H. Jiang, S. Elbaum, and C. Detweiler, "Reducing failure rates of robotic systems though inferred invariants monitoring," in *International Conference on Intelligent Robots and Systems*. IEEE, 2013.

[33] A. Desai, T. Dreossi, and S. A. Seshia, "Combining model checking and runtime verification for safe robotics," in *Runtime Verification*, S. Lahiri and G. Reger, Eds. Cham: Springer International Publishing, 2017.

[34] C. Gladisch et al., "Experience paper: search-based testing in automated driving control applications," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '19. IEEE Press, 2020, p. 26–37.

[35] H. Araujo, M. R. Mousavi, and M. Varshosaz, "Testing, validation, and verification of robotic and autonomous systems: A systematic review," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, mar 2023.

[36] S. Demarchi, D. Guidotti, L. Pulina, A. Tacchella, N. Narodytska, G. Amir, G. Katz, and O. Isac, "Supporting standardization of neural networks verification with vnnlib and coconet." in *FoMLAS@CAV*, 2023.

[37] X. Chang, P. Ren, P. Xu, Z. Li, X. Chen, and A. Hauptmann, "A comprehensive survey of scene graphs: Generation and application," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[38] H. Li, G. Zhu, L. Zhang, Y. Jiang, Y. Dang, H. Hou, P. Shen, X. Zhao, S. A. A. Shah, and M. Bennamoun, "Scene graph generation: A comprehensive survey," *Neurocomput.*, vol. 566, no. C, mar 2024.

[39] A. V. Malawade, S.-Y. Yu, B. Hsu, H. Kaeley, A. Karra, and M. A. Al Faruque, "Roadscene2vec: A tool for extracting and embedding road scene-graphs," *Know.-Based Syst.*, vol. 242, no. C, apr 2022.

[40] J. Li et al., "Important object identification with semi-supervised learning for autonomous driving," in *2022 International Conference on Robotics and Automation (ICRA)*, 2022, p. 2913–2919.

[41] A. Prakash, S. Debnath, J.-F. Lafleche, E. Cameracci, G. State, S. Birchfield, and M. T. Law, "Self-supervised real-to-sim scene generation," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021, pp. 16 024–16 034.

[42] T. Woodlief, F. Toledo, S. Elbaum, and M. B. Dwyer, "Closing the gap between sensor inputs and driving properties: A scene graph generator for carla," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2025, pp. 29–32.

[43] R. Li, S. Zhang, D. Lin, K. Chen, and X. He, "From pixels to graphs: Open-vocabulary scene graph generation with vision-language models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 28 076–28 086.

[44] F. Toledo, S. Elbaum, D. Gopinath, R. Kaur, R. Mangal, C. S. Pasareanu, A. Roy, and S. Jha, "Monitoring safety properties for autonomous driving systems with vision-language models," 2025, accessed on 08.11.2025. Link.

[45] B. Li, P. Qi, B. Liu, S. Di, J. Liu, J. Pei, J. Yi, and B. Zhou, "Trustworthy ai: From principles to practices," *ACM Comput. Surv.*, 2023.

[46] Y. Sun, C. M. Poskitt, X. Zhang, and J. Sun, "Redriver: Runtime enforcement for autonomous vehicles," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024.

[47] L. von Rueden, S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach, R. Heese, B. Kirsch, J. Pfrommer, A. Pick, R. Ramamurthy, M. Walczak, J. Garcke, C. Bauckhage, and J. Schuecker, "Informed machine learning – a taxonomy and survey of integrating prior knowledge into learning systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 1, pp. 614–633, 2023.

[48] X. Li, C.-I. Vasile, and C. Belta, "Reinforcement learning with temporal logic rewards," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 3834–3839.

[49] A. Balakrishnan and J. V. Deshmukh, "Structured reward shaping using signal temporal logic specifications," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[50] H. Hasanbeig, D. Kroening, and A. Abate, "Certified reinforcement learning with logic guidance," *Artificial Intelligence*, vol. 322, p. 103949, 2023.

[51] A. Karpatne, W. Watkins, J. S. Read, and V. Kumar, "Physics-guided neural networks (pgnn): An application in lake temperature modeling," *ArXiv*, vol. abs/1710.11431, 2017.

[52] R. Stewart and S. Ermon, "Label-free supervision of neural networks with physics and domain knowledge," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, p. 2576–2582.

[53] N. Muralidhar, M. R. Islam, M. Marwah, A. Karpatne, and N. Ramakrishnan, "Incorporating prior domain knowledge into deep neural networks," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 36–45.

[54] V. D. of Motor Vehicles, "Virginia driver's manual," accessed on 04.23.2025. Link.

[55] T. Woodlief, S. Elbaum, and K. Sullivan, "Semantic image fuzzing of ai perception systems," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1958–1969.

[56] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[57] F. Fioretto, P. Van Hentenryck, T. W. K. Mak, C. Tran, F. Baldo, and M. Lombardi, "Lagrangian duality for constrained deep learning," in *Machine Learning and Knowledge Discovery in Databases. Applied Data Science and Demo Track*. Cham: Springer, 2021, pp. 118–135.

[58] A. Paszke et al., *PyTorch: an imperative style, high-performance deep learning library*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[59] P. Wu, X. Jia, L. Chen, J. Yan, H. Li, and Y. Qiao, "Trajectory-guided control prediction for end-to-end autonomous driving: a simple yet strong baseline," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2024.

[60] CarlaSimulator, "Carla leaderboard," accessed on 2024-07-19. Link.

[61] A. D. et al., "CARLA: an open urban driving simulator," *CoRR*, vol. abs/1711.03938, 2017.

[62] PapersWithCode, "Scene graph generation on visual genome," 2023, accessed on 08.20.2024. Link.

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

[64] D. Chen and P. Krähenbühl, "Learning from all vehicles," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022, pp. 17 201–17 210.

[65] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, "Detectron2," 2019, accessed on 08.11.2025. Link.

[66] V. Arampatzakis et al., "Monocular depth estimation: A thorough review," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 4, pp. 2396–2414, 2024.

[67] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[68] K. Cho et al., "Learning phrase representations using rnn encoder–decoder for statistical machine translation," in *Conference on Empirical Methods in Natural Language Processing*, 2014.

[69] OpenDriveLab, "Tcp training configuration," https://github.com/OpenDriveLab/TCP/blob/main/TCP/config.py, accessed: 2024-07-19.

APPENDIX

A. *Controlled Experiment Dataset*

To provide more information about the dataset used in the controlled experiment, we include the following additional data. Table III shows the number of data points in each of our 6 splits. The train and validation columns represent the number of samples from the remaining towns, while the test column represents the number of samples from the town left out. Table IV shows the number of training and validation data points in each of the towns. Table V shows the number of property preconditions satisfied per split. The town mentioned in the split column is the one left out.

TABLE III: Data points for each split.

Split	Train	Val	Test
Town 01 left out	345,757	45,674	9,056
Town 02 left out	338,732	48,040	6,690
Town 04 left out	361,018	47,564	7,166
Town 05 left out	339,656	43,480	11,250
Town 07 left out	366,186	45,251	9,479
Town 10 left out	344,416	43,641	11,089

TABLE IV: Number of train and val data points per town.

Town	Train	Val	Total
Town 01	73,396	9,056	82,452
Town 02	80,421	6,690	87,111
Town 04	58,135	7,166	65,301
Town 05	79,497	11,250	90,747
Town 07	52,967	9,479	62,446
Town 10	74,737	11,089	85,826

TABLE V: Properties' preconditions for each split.

Split	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6
Town 01	71,590	114,279	58,813	21,538	190,678	173,238
Town 02	64,462	114,947	62,016	20,310	185,042	167,602
Town 04	82,899	124,266	51,394	24,732	220,419	205,241
Town 05	80,600	108,778	55,791	22,618	217,678	212,275
Town 07	75,898	136,267	60,898	26,541	223,841	206,229
Town 10	75,546	127,588	55,513	23,771	224,367	210,240
Average	75,166	121,021	57,404	23,252	210,338	195,804

Furthermore, we present the number of violations for each property and split given the number of properties in the optimization in Table VI. $\mathcal{B}$ represent the base models and $\mathcal{M}$ represent the models trained with T4PC. These violations appear on the towns left out during training.

B. *Case Study Details*

1) Interfuser controller approximation: Interfuser employs a controller implemented in Python, consisting of over 400 lines of code. We cannot directly apply T4PC to Interfuser because this module is not differentiable. To address this, we approximate the controller by collecting its inputs and outputs and training a DNN. The controller takes as input ten future waypoints, a semantic feature map, and traffic-related information such as traffic light status, stop sign presence, and intersection detection. Its outputs a steering angle ranging from -1 to 1, and acceleration ranging from -1 to 0.75. Since the properties we aim to improve depend solely on acceleration, we focus our approximation efforts on that output. The DNN architecture consists of six fully connected layers with 2,824, 2,500, 2,000, 1,500, 1,000, and 500 neurons respectively, each using ReLU activation functions. The input layer has 2,824 neurons to match the dimensionality of the flattened input features. We train the model using Mean Absolute Error (MAE) as the loss function, the Adam optimizer, and a learning rate of 10^{-4}. Training runs for up to 1,000 epochs with a ReduceLROnPlateau learning rate scheduler using default parameters, reducing the learning rate by a factor of 10 if validation performance does not improve for 10 consecutive epochs. Early stopping is triggered if the learning rate falls below 10^{-7}. Training concluded at epoch 62, achieving a final MAE of 0.04.

2) Data augmentation: The application of T4PC to TCP required property-based augmentation because the preconditions were only met for ϕ_7 and ϕ_8 by 913/206,268 (0.44%) and 481/206,268 (0.23%) dataset inputs. To bring these numbers closer to the proportions used in the experiment, we arbitrarily selected a δ of 10% for ϕ_x^7 and 5% for ϕ_x^8. We wanted to have more data points that meet ϕ_x^7 compared to ϕ_x^8 because we found that TCP struggled to stop when there was a stop sign. We then defined a pool of transformation candidates by retrieving the data points with the same properties' preconditions met except for the ego speed constraint (less than 0.70% of dataset). Then we used two metamorphic functions, one for ϕ_7 and one for ϕ_8, to adjust the speeds of the data points to satisfy the preconditions. For example, given an input x that meets the ϕ_7 precondition with $x.speed \geq 0.1$, $f_{0.5}$ changes x's speed to 0.5 (still meeting the precondition), while the corresponding $g_{0.5}(y) = y$ is the identity as changing the speed should not change the output, but helps increase the DNN robustness. The functions generate 19 and 10 new data points for each element in the transformation candidate pool, producing a total of 26,505/246,633 (10.75%) and 13,860/246,633 (5.62%), satisfying the target δ. This property-based augmentation dataset is then used by T4PC.

In contrast, we did not apply data augmentation when training Interfuser with T4PC, as the collected dataset already contained sufficient coverage of property preconditions. Specifically, 18.44% of the data met the ϕ_1 precondition, 34.67% ϕ_2, 16.49% ϕ_3, and 9.01% ϕ_4.

3) Training Hyperparameters and Hardware: For TCP, we followed its original training setup. We used the Adam optimizer and a starting lr of 5×10^{-5}, consistent with the final stage of its original training procedure. The lr was halved midway through training, as in the original setup. When applying T4PC, we used a property loss weight ρ of 0.1. We used a batch size of 256 and performed training on 8 CPU cores, 200GB of RAM, and 2 A100 GPUs.

For Interfuser, we also followed its original training setup by using the Adam optimizer with an initial learning rate (lr) of 5×10^{-5}—the final rate used in their official training—and applied a cosine lr scheduler. When training with T4PC, we set the property loss weight ρ to 0.001, as higher values were found to disrupt the main losses. We used a batch size of 64 and ran experiments on a machine with 16 CPU cores, 250GB of RAM, and 4 A40 GPUs.

TABLE VI: Property violations per split and # of properties in optimization

		1 prop		2 props		4 props		6 props		Ft 1 prop	
		$\mathcal{B}$	$\mathcal{M}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{B}$	$\mathcal{M}$
Property	Split										
ϕ_1 - StopToAvoidCollision	Town 01	132.0	27.0	132.0	33.8	132.0	44.6	132.0	25.0	136.0	32.0
	Town 02	98.0	28.0	98.0	37.8	98.0	40.0	98.0	46.0	97.0	14.0
	Town 04	38.0	14.0	38.0	28.8	38.0	35.5	38.0	70.0	36.0	15.0
	Town 05	97.0	53.0	97.0	44.2	97.0	41.4	97.0	32.0	95.0	56.0
	Town 07	133.0	61.0	133.0	99.4	133.0	140.5	133.0	227.0	131.0	62.0
	Town 10	24.0	9.0	24.0	7.4	24.0	9.0	24.0	23.0	23.0	8.0
ϕ_2 - StopForYellowRed	Town 01	978.0	303.0	978.0	259.6	978.0	234.7	978.0	241.0	1000.0	327.0
	Town 02	582.0	74.0	582.0	64.2	582.0	47.5	582.0	59.0	582.0	60.0
	Town 04	743.0	63.0	743.0	57.4	743.0	48.0	743.0	59.0	754.0	77.0
	Town 05	602.0	56.0	602.0	75.2	602.0	84.3	602.0	90.0	590.0	79.0
	Town 07	461.0	294.0	461.0	250.2	461.0	257.2	461.0	258.0	454.0	314.0
	Town 10	832.0	64.0	832.0	143.6	832.0	316.4	832.0	411.0	863.0	138.0
ϕ_3 - NoStopForNoReason	Town 01	24.0	3.0	24.0	12.2	24.0	25.8	24.0	48.0	24.0	4.0
	Town 02	94.0	89.0	94.0	93.6	94.0	129.8	94.0	156.0	93.0	77.0
	Town 04	68.0	1.0	68.0	12.2	68.0	33.8	68.0	43.0	72.0	4.0
	Town 05	155.0	55.0	155.0	81.6	155.0	108.1	155.0	118.0	152.0	72.0
	Town 07	389.0	106.0	389.0	169.6	389.0	297.7	389.0	389.0	385.0	130.0
	Town 10	1563.0	1257.0	1563.0	1236.0	1563.0	1263.4	1563.0	1027.0	1559.0	1198.0
ϕ_4 - AccelerateForGreen	Town 01	63.0	21.0	63.0	30.4	63.0	46.2	63.0	94.0	60.0	17.0
	Town 02	78.0	33.0	78.0	37.4	78.0	58.1	78.0	47.0	78.0	21.0
	Town 04	22.0	2.0	22.0	5.6	22.0	7.3	22.0	7.0	22.0	0.0
	Town 05	229.0	58.0	229.0	68.4	229.0	86.7	229.0	152.0	226.0	91.0
	Town 07	39.0	3.0	39.0	8.6	39.0	12.0	39.0	14.0	34.0	4.0
	Town 10	157.0	73.0	157.0	118.0	157.0	174.4	157.0	157.0	150.0	49.0
ϕ_5 - NoSteerRightOutRoad	Town 01	1531.0	0.0	1531.0	0.0	1531.0	0.0	1531.0	0.0	1552.0	0.0
	Town 02	1114.0	0.0	1114.0	0.0	1114.0	0.0	1114.0	0.0	1156.0	0.0
	Town 04	564.0	0.0	564.0	0.0	564.0	0.0	564.0	0.0	503.0	0.0
	Town 05	2440.0	0.0	2440.0	0.0	2440.0	0.0	2440.0	0.0	2444.0	93.0
	Town 07	263.0	0.0	263.0	0.0	263.0	0.0	263.0	0.0	268.0	0.0
	Town 10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ϕ_6 - NoSteerLeftOutRoad	Town 01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Town 02	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Town 04	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Town 05	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Town 07	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Town 10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

4) Extended tables: Full CARLA leaderboard infractions table for TCP (Table VII, extension of Table IIa) and Interfuser (Table VIII, extension of Table IIb).

TABLE VII: TCP infractions. Bold means statistically significantly better.

Treatment	Driving Score ↑	Collision Layout ↓	Collision Pedestrians ↓	Collision Vehicles ↓	Outside Lanes ↓	Red Light Infraction ↓	Route Deviation ↓	Route Timeout ↓	Stop Sign Infraction ↓	Vehicle Blocked ↓
T4PC 5	81.09±6.58	0.00±0.00	0.00±0.00	0.13±0.17	0.00±0.00	0.04±0.05	0.00±0.00	0.01±0.03	**0.54±0.20**	0.09±0.09
Base 5	79.20±4.34	0.00±0.00	0.00±0.00	0.12±0.14	0.00±0.00	0.04±0.06	0.00±0.00	0.00±0.00	0.98±0.22	**0.00±0.00**
T4PC 10	**81.98±5.16**	0.00±0.00	0.00±0.00	0.16±0.13	0.00±0.00	0.04±0.06	0.00±0.00	0.01±0.03	**0.59±0.18**	0.03±0.05
Base 10	78.37±3.48	0.00±0.00	0.00±0.00	0.11±0.09	0.00±0.02	0.02±0.04	0.00±0.00	0.00±0.00	1.02±0.17	0.01±0.03
T4PC 15	**81.81±4.77**	0.00±0.00	0.00±0.00	**0.10±0.10**	0.00±0.00	0.03±0.05	0.00±0.00	0.00±0.00	**0.76±0.23**	0.02±0.04
Base 15	75.99±3.31	0.00±0.00	0.00±0.00	0.16±0.10	0.00±0.00	0.03±0.06	0.00±0.00	0.00±0.02	1.06±0.19	0.01±0.03

TABLE VIII: Interfuser infractions. Bold means statistically significantly better.

Treatment	Driving Score ↑	Collision Layout ↓	Collision Pedestrians ↓	Collision Vehicles ↓	Outside Lanes ↓	Red Light Infraction ↓	Route Deviation ↓	Route Timeout ↓	Stop Sign Infraction ↓	Vehicle Blocked ↓
T4PC 3	58.40±9.22	0.00±0.00	0.21±0.13	0.61±0.31	0.00±0.00	0.42±0.11	0.00±0.00	0.20±0.15	0.00±0.00	0.00±0.00
Base 3	59.44±8.33	0.00±0.02	0.19±0.12	0.51±0.26	0.01±0.03	0.44±0.12	0.00±0.00	0.22±0.11	0.00±0.00	0.00±0.02
T4PC 5	63.90±8.33	0.00±0.00	0.16±0.10	0.60±0.27	0.01±0.03	**0.24±0.10**	0.00±0.00	0.17±0.14	0.00±0.00	0.00±0.00
Base 5	59.45±8.35	0.00±0.00	0.15±0.11	0.53±0.23	0.00±0.00	0.43±0.10	0.00±0.00	0.21±0.15	0.00±0.00	0.00±0.00
T4PC 7	62.48±9.16	0.00±0.00	**0.17±0.10**	0.62±0.18	0.00±0.00	**0.19±0.08**	0.00±0.00	0.23±0.17	0.00±0.00	0.00±0.02
Base 7	60.15±6.92	0.00±0.00	0.28±0.15	0.52±0.22	0.01±0.03	0.42±0.11	0.00±0.00	**0.14±0.10**	0.00±0.00	0.00±0.00