

Verification of Autonomous Neural Car Control with KeYmaera X

Enguerrand Prebet^[0009–0008–0160–5219], Samuel Teuber^[0000–0001–7945–9110],
and André Platzer^[0000–0001–7238–5710]

Karlsruhe Institute of Technology, Karlsruhe, Germany
 {enguerrand.prebet,teuber,platzer}@kit.edu

Abstract. This article presents a formal model and formal safety proofs for the ABZ’25 case study in differential dynamic logic (dL). The case study considers an autonomous car driving on a highway with a neural network controller avoiding collisions with neighbouring cars. Using KeYmaera X’s dL implementation we prove collision-freedom on an infinite time horizon which ensures that safety is preserved independently of trip length. The safety guarantees hold for time-varying reaction time and brake force. Our dL model considers the single lane scenario with cars ahead or behind. We demonstrate dL and its tools are a rigorous foundation for runtime monitoring, shielding, and neural network verification. Doing so sheds light on inconsistencies between the provided specification and simulation environment `highway-env` of the ABZ’25 study. We attempt to fix these inconsistencies and uncover numerous counterexamples indicative of issues in the provided reinforcement learning environment.

Keywords: Differential dynamic logic · Hybrid systems · Formal verification · Highway car control · Neural Network Control Systems.

1 Introduction

This paper contributes a comprehensive study of formal safety proofs for the ABZ’25 highway case study of straight-line driving on highways with a neural network (NN) control system for the ego car based on the rigorous foundations of differential dynamic logic [25, 26, 28, 29] (dL). Given the interest in highway driving, the contributions to the ABZ’25 case study challenge stand a more general appeal. While the specific outcomes focus on the ABZ’25 case study, the generality of the underlying tools could help make other applications safe.

Contributions. To tackle ABZ’s case study we provide: *i)* A formal, provably safe dL [25, 26, 28, 29] model of the hybrid systems dynamics of straight-line driving described by ABZ’25 [16]. We identify the control constraints required for safe driving. *ii)* A derivation of real arithmetic constraints that serve either as sandbox/shield for the black-box NN or for the *gapless* rigorous white-box verification of concrete NNs. *iii)* A verification-based, exhaustive characterization of all unsafe behaviours in two NNs trained using the `highway-env` environment provided by ABZ’25. *iv)* An empirical validation of the derived sandbox and shield.

Importantly, our safe controller and the derived monitoring/verification conditions are fully symbolic and proved safe for arbitrary parameter choices making the model, controller, sandbox and NN verification technique useful for future endeavours. Additionally, reaction time and braking power may vary (within bounds) during execution. The results underscore that safety guarantees in dL are practically applicable to (neural) real-world systems – either through monitoring/shielding or via verification of the NN w.r.t. dL derived constraints.

While we demonstrate that dL and implementation monitoring/verification can be *gaplessly* integrated, we observe the existence of a significant *model-to-simulation (model2sim)* gap between the specification and the simulator provided by ABZ [16]. The well-known *sim2real gap* leads to decreased performance when simulation-trained agents are deployed in the real world. Similarly, the *model2sim* gap induces unsafe behaviour of an agent if the simulation insufficiently matches the model’s assumptions about the real world. We identify this gap as an important roadblock on the highway to safe NN controllers.

Related Work. Prior work analysed safe car control in dL [18, 27, 32] (in one instance using refinement [17]). Unlike prior case studies applying dL guarantees to NN control [8, 34], this work has a more complex environment (e.g. variable speed for surrounding cars) which increases the complexity of safety criteria. Car control (with different dynamics [35]) has also been studied by numerous closed-loop NN verification tools (see e.g. the ARCH competition [19–21]). Unlike the closed-loop approaches, our work can provide guarantees on an infinite-time horizon, i.e. independent of the car’s trip length. Event-B [1] has also been used to model automotive applications [2] without application to NNs. Unlike a `highway-env` ProB model [37] we explicitly model the environment’s continuous dynamics and support NN verification. Unlike another shielding approach [33] we characterize safe behaviour *a priori* instead of learning from catastrophic behaviour.

2 Background

This section provides an overview of differential dynamic logic (dL). Before presenting results on highway car control, we first illustrate the concepts of this section using a cartoonishly simplified application: We consider a car that starts at a one-dimensional, positive position p and pretend the car’s controller can influence the car’s position by directly choosing the car’s velocity v with immediate effect. The safety requirement of the controller is to keep the car at a positive position, i.e. $p > 0$. We first present dL in general, then the ModelPlex technology for the derivation of runtime monitors and three applications of these formulas.

2.1 Differential Dynamic Logic for Hybrid Systems

dL is a program logic for reasoning about cyber-physical systems given as *hybrid programs*. On a high level, dL is a first-order multi-modal logic where modalities are parameterized with programs and the first-order formulas are interpreted w.r.t. real arithmetic. Formulas of dL have the following structure:

Definition 1 (Formulas). Formulas are defined by the grammar below where θ, η are terms, ϕ, ψ are formulas and α, β are hybrid programs (Definition 2):

$$\phi, \psi ::= \theta \leq \eta \mid \neg\phi \mid \phi \wedge \psi \mid \forall x \phi \mid [\alpha]\phi \mid \alpha \leq \beta$$

While the first four elements of the grammar correspond to logical structures known from first-order real arithmetic formulas, the latter two are specific to differential dynamic logic [25, 26, 28, 29] and differential refinement logic [17, 30].

Unlike first-order formulas which are usually evaluated in a fixed structure, dL evaluates formulas w.r.t. states that assign values to variables. The programs (which will be discussed in greater detail below) then induce a state transition relation which is integrated into the logic via the grammar's fifth formula: $[\alpha]\phi$ is true in a state ω iff *after every program run* of α the formula ϕ is satisfied, i.e. if for all state transitions of α from the current state ω the formula ϕ holds in the resulting state. If ϕ is a property that indicates safety of the system, then $[\alpha]\phi$ expresses that the system always remains safe (see Section 3.5). Finally, $\alpha \leq \beta$ expresses that the program α refines the program β in the current state, i.e. $\alpha \leq \beta$ holds in a state ω iff all states reachable from ω via the transitions of α are also reachable via β 's state transition relation. Refinements are used to transfer safety properties between hybrid programs (see Section 4.1). A formula is called *valid* if it is satisfied in all states. We now turn to dL 's *hybrid programs* which allow discrete and continuous actions and are formally defined as follows:

Definition 2 (Hybrid Programs). Hybrid programs α, β are defined by the grammar below where x is a variable, θ is a term and ψ is a formula:

$$\alpha, \beta ::= ?\psi \mid x := \theta \mid x := * \mid x' = \theta \& \psi \mid \alpha \cup \beta \mid \alpha; \beta \mid \alpha^*$$

The first program primitive $?ψ$ (check) only proceeds if formula ψ is satisfied in the current state. The second and third primitive are assignments, either w.r.t. a term ($x := \theta$) or nondeterministically to an arbitrary value ($x := *$). The fourth primitive ($x' = \theta \& \psi$) describes the continuous, nondeterministic evolution of variable x along the differential equation $x' = \theta$ within the domain constraint ψ . The next two primitives allow the composition of programs by either nondeterministically choosing one of two ($\alpha \cup \beta$) or by executing them sequentially ($\alpha; \beta$). The final primitive α^* nondeterministically runs the program α for 0 or more iterations. The support of hybrid programs for continuous evolution and discrete as well as continuous nondeterminism is e.g. crucial for the analysis of cyber-physical systems without a fixed clock cycle. Many classical program constructs can be translated into the primitives of hybrid programs. For example, if-then-else can be rewritten as follows: $\text{if } (\psi) \alpha \text{ else } \beta \stackrel{\text{def}}{=} (?ψ; α) ∪ (?¬ψ; β)$. Similarly, we can represent while loops: $\text{while } (\psi) \alpha \stackrel{\text{def}}{=} (?ψ; α)^*; ?¬ψ$.

Example. We now explain how our simple cyber-physical system (the velocity-controlled car) can be modelled in dL . All variables, p, v, \dots , that evolve along the execution are in lower-case, while constants like T are in upper-case. As

outlined above, the car’s position is described by a real-valued position p . The car’s dynamics are then described by the differential equation $p' = v$ where v is the velocity determined by the controller. To derive safety guarantees we assume that our controller is invoked at least every T seconds. Hence, we model the physical part of our example as $\alpha_{\text{plant}} \stackrel{\text{def}}{=} t := 0; p' = v, t' = 1 \& t \leq T$. Here, p evolves as outlined above and we additionally introduced a clock variable t which guarantees that the evolution runs for at most T seconds via the domain constraint $t \leq T$. We already formulated the car’s safety condition as $p > 0$ at the beginning of this section. The final ingredient for our dL model is a *control envelope* that provides a nondeterministic description of allowed behaviour which keeps the system safe. Using dL to verify control envelopes rather than one concrete controller is quite a common approach as it allows the verification of a whole family of possible controller implementations at once [11]. It is generally preferable to design very general control envelopes that encompass the largest possible range of behaviours that can be certified as safe. In our example, we can formulate the control envelope as the nondeterministic program $\alpha_{\text{ctrl}} \stackrel{\text{def}}{=} v := *; ?(p + Tv > 0)$. This control envelope ensures that we only choose velocities v that avoid negative positions. Indeed, we can use dL ’s proof calculus (and its implementation in KeYmaera X) to prove the validity of the following dL formula $T > 0 \wedge p > 0 \rightarrow [(\alpha_{\text{ctrl}}; \alpha_{\text{plant}})^*]p > 0$. This formula expresses that (assuming an initial state with $T > 0$ and $p > 0$) we can run this system for arbitrarily long (note the nondeterministic loop) and the safety condition $p > 0$ will always be satisfied afterwards. This can be proven inductively through the invariant $T > 0 \wedge p > 0$. The ability to perform inductive, infinite time horizon reasoning for dL models is one of the major advantages of dL over many reachability-based analyses.

The formula above also exhibits a very common pattern in dL models where we provide a safety guarantee over the execution of a nondeterministic loop which consists of the sequential execution of a control envelope (α_{ctrl}) and an environment model (α_{plant}). However, while we have verified an infinite class of potential controllers, we have not yet verified any concrete given controller implementation. In the remainder of this section, we will present dL -based technologies that allow us to bridge the gap between a verified control envelope and a concrete controller implementation.

2.2 ModelPlex for Verified Runtime Monitoring

In the previous section, we saw how dL can be used to model cyber-physical systems and to verify control envelopes. However, the verified control envelopes differ from the control systems we would like to use in practice: Concrete, real-world controllers will often be implemented in compilable programming languages or, as in the case of the highway case study, the controller’s behaviour might even be determined by an NN. This raises the question how this challenge can be overcome. On the one hand, it is possible to embed the behaviour of more complicated programming languages into dL [10, 12], however, such approaches are

always tailored to specific programming languages and require that we perform interactive proofs on the concrete controller’s behaviour. On the other hand, we can use a verified control envelope to derive runtime monitoring conditions that can subsequently be checked on a concrete system – possibly even in a black box fashion. This technique to derive correct-by-construction runtime monitoring conditions from a given control envelope α_{ctrl} is called *ModelPlex* [22].

Based on a given control envelope α_{ctrl} over variables $V(\alpha_{ctrl})$, ModelPlex uses dL’s calculus rules to derive a first-order real arithmetic formula ψ over variables $V(\alpha_{ctrl}) \dot{\cup} \{x^+ | x \in V(\alpha_{ctrl})\}$ where x^+ indicates the value of x in the next state. For instance assuming $V(\alpha_{ctrl}) = \{x\}$, if the formula ψ is satisfied by $x = v_1$ and $x^+ = v_2$ for some v_1, v_2 , then there exists a state transition for α_{ctrl} where the value of x changes from v_1 to v_2 . Since we have a safety proof for α_{ctrl} this implies the safety guarantees for our control envelope carry over to a system where x ’s value changes from v_1 to v_2 . Hence, the formula ψ can be used to monitor the safety of a (black-box) controller implementation by checking whether a concrete assignment of the implementation’s pre- and post-values satisfies ψ .

For the velocity-controller car, the variables $V(\alpha_{ctrl})$ are the position and the velocity: $\{p, v\}$, and the formula given by ModelPlex is $\psi \stackrel{\text{def}}{=} p^+ = p \wedge p + T v^+ > 0 \wedge t^+ = 0$. Thus, any concrete implementation of such a controller will be safe if this formula is satisfied during execution, i.e if the controller does not change the position ($p = p^+$) and sets some velocity v^+ that respects $p + T v^+ > 0$. Additionally, it requires that the clock variable t be reset to 0.

2.3 Applications of ModelPlex

The formula computed by ModelPlex [22] tells us which control actions come with a dL 0 safety guarantee. As explained below, this formula can be used in at least three manners to derive safety guarantees for controller implementations.

Monitoring (VeriPhy). First and foremost, we can use the derived formula to check the actions computed by the controller implementation at runtime via a runtime monitor. To this end, we assign the formula’s variables with the implementation’s input and output values and check whether the action is provably safe according to the ModelPlex runtime monitor. In case the implementation chooses an action violating the runtime monitor, we overwrite the action using a fallback controller. This approach comes with a formally verified code generation pipeline called VeriPhy [3] which serves as a sandbox for a given controller and comprises provably correct machine arithmetic.

Shielding (Justified Speculative Control). One drawback of VeriPhy in the context of NN Control is its conservatism: While traditionally programmed controllers usually return exactly one action that must be overwritten if unsafe, NNs often return a probability distribution over actions. However, it is not necessarily reasonable to entirely overwrite the NN’s action if its most likely action is unsafe. Instead Justified Speculative Control [8,9] (JSC) *shields* the NN using

runtime enforcement technique [14, 22] that constrain the action space to known-safe options. Thus, JSC can still treat the concrete controller as a black box but allow for more flexibility in the chosen actions. To this end, JSC checks for possible actions whether they satisfy the ModelPlex condition. JSC then chooses the allowed action with the highest probability according to the reinforcement learning agent. Additionally, JSC only performs a safety check in situations where the environment behaves as modelled in dL (this is achieved via ModelPlex’s environment monitoring technology which goes beyond the scope of this exposition). Importantly, this technique can be applied both during training and at runtime.

Verification (VerSAILLE & NCubeV). The previous approaches only provide *a posteriori* guarantees by restricting or overwriting the controller’s actions at runtime. Alternatively, we can also use the monitoring condition derived by ModelPlex for *a priori* verification of the NN. This is achieved via the VerSAILLE approach [34]: In essence, we verify whether there exists a state inside the dL model’s invariant state space where the NN’s action violates the ModelPlex controller monitor. For this section’s running example, we would verify that an NN (with input p and output v^+) satisfies the following specification [34, Thm. 2]:

$$\underbrace{p > 0}_{\text{Invariant}} \rightarrow \underbrace{p + T v^+ > 0}_{\text{Controller Monitor}} .$$

This is achieved by a compute-intensive numerical analysis of the NN that mathematically proves the absence of such counterexamples. As our running example has a simple, linear controller monitor and invariant, most modern NN verifiers (as reported in recent surveys and competitions [4, 5, 13]) can be used. However, for realistic dL models, the ModelPlex conditions usually have a significantly more complicated propositional structure with nonlinear real arithmetic. Neither of these features is supported by “classical” NN verifiers nor by their common specification language [6]. To this end, we recently proposed the NCubeV tool [34] supporting both arbitrary propositional structure and polynomial arithmetic.

The usage of NN verification has multiple advantages. First, it allows the deployment of autonomous, unmonitored NN Control Systems. Second, it allows the usage of NNs in applications without an obvious fallback strategy or for cases with continuous action spaces. Finally, it can also serve for diagnostics: Either to estimate how often a given NN performs (un)safe actions or to discover unsafe behaviour that is empirically invisible, e.g. due to simulator limitations.

3 A Verified dL Model for the ABZ Highway Case-Study

This section presents the verified dL model developed for this case study. We start by introducing the cyber-physical system of interest (Section 3.1). After giving the general structure and how it interleaves the discrete and continuous actions that can occur between each control cycle (Section 3.2), we focus on the plant (Section 3.3) and the controller (Section 3.4). Finally, we express safety conditions in dL for the model and verify them using the theorem prover KeYmaera X [7, 24, 28] (Section 3.5). Our proofs are reproducible via an artifact [31].

3.1 A Safe Autonomous Driving System

The model is about a safe autonomous driving system, referred to as the ego car, that should prevent collision with another car on a single straight lane. All constants, A_{\max}, V, T, \dots , must be positive except for braking deceleration, B_{\min}, B_{\max} , which are negative. Both cars have length L but are modelled as single points: with position x_o , speed v_o , and acceleration a_o for the ego car, and with position x_e , speed v_e , and acceleration a_e for the other. Thus, absence of collision is ensured by maintaining a distance of at least L between the two cars. No car moves backwards and their speed is at most V . The cars have a maximum acceleration of A_{\max} and a maximum braking deceleration of B_{\max} . Additionally, the ego car may not always draw the maximum power of the brake or the engine. It will however always be able to brake with deceleration at least $B_{\min} \geq B_{\max}$ and accelerate with acceleration at least $A_{\min} \leq A_{\max}$. These constraints are imposed on the cars themselves, so even if they are trying to brake or accelerate, they cannot go backward or exceed speed limit V . The ego car observes the environment at least every T seconds, whereas the other car may react more often without restriction. No regularity or periodicity is assumed in the reaction time of the ego car as long as it always remains below T seconds.

Overall, the constants are constrained by the formula: $\text{ctx}_C \stackrel{\text{def}}{=} T > 0 \wedge L > 0 \wedge V > 0 \wedge B_{\max} \leq B_{\min} < 0 < A_{\min} \leq A_{\max}$. It can be extended by bounds on speed and acceleration: $\text{ctx} \stackrel{\text{def}}{=} \text{ctx}_C \wedge B_{\max} \leq a_e, a_o \leq A_{\max} \wedge 0 \leq v_e, v_o \leq V$.

3.2 Overall Structure of the dL Model

The general structure of the model is as follows:

$$\text{model}(c) ::= (\underbrace{\text{ctrl}_o; (c \cup ?t < t_e + T);}_{\text{control}} \underbrace{\text{accelCorr}; \text{dyn}}_{\text{plant}})^*$$

The model is parametric in the controller of the ego car c to handle both the generic controller ctrl_e (see Section 3.4) and the NN controller ctrl_{NN} (see Section 4.1). In this section, we write model for $\text{model}(\text{ctrl}_e)$.

ctrl_o models the controller of the other car. It does not assume any minimal time between each execution of ctrl_o . Then c models the controller of the ego car and sets t_e to t . If it has been less than T seconds since t_e was last set, the nondeterministic choice allows c to be skipped. Thus, the controller is only assumed to run at least once every T seconds. Having the possibility of skipping the controller allows discrete events, e.g. the other controller, to still occur without the ego car reacting. accelCorr (defined in Section 3.3) models the acceleration correction when reaching the speed boundaries. It ensures that a braking car, with negative acceleration, does not go backwards by changing its acceleration to zero. This is a discrete change but happens independently of any controller. In particular, the ego car does not notice the change before its next control cycle. Finally, dyn models the continuous dynamics of the system, i.e. the actual motion of the car evolving with time. These execute in a loop so

that the system alternates between the control and the plant arbitrarily many times. We elaborate the details of each component, starting with the plant.

3.3 Modelling the Physical Plant

$$\text{accelCorr} \quad \begin{array}{l} \text{if } (v_o = 0 \wedge a_o < 0) \vee (v_o = V \wedge a_o > 0) a_o := 0 \\ \text{if } (v_e = 0 \wedge a_e < 0) \vee (v_e = V \wedge a_e > 0) a_e := 0 \end{array}$$

$$\text{dyn} \quad \begin{array}{l} x'_e = v_e, v'_e = a_e, x'_o = v_o, v'_o = a_o, t' = 1 \\ \quad \& t \leq t_e + T \wedge 0 \leq v_e \leq V \wedge 0 \leq v_o \leq V \end{array}$$

The plant is composed of a discrete part, `accelCorr`, and a continuous part, `dyn`. First, if any car has come to a stop or reached their speed limit, then their acceleration is set to 0 for saturation. Then the continuous dynamics follows the ODEs specifying for both cars, that speed is the derivative of the position, $x'_i = v_i$, and that acceleration is the derivative of speed, $v'_i = a_i$. Time is explicit with constant derivative. The domain constraints ensure that the dynamics always stop before a discrete event must be executed, whether it is a controller event – if $t = t_e + T$ – or a plant event – if a car stops, or reaches their speed limit.

3.4 Modelling the Car Controllers

The control consists of the controllers for the two cars. The controller `ctrlo` for the other car isn't concerned about safety so it just selects any acceleration within the limitation of the vehicle. As the assignment is nondeterministic, all choices

$$\text{ctrl}_o \quad a_o := *; ?(B_{\max} \leq a_o \leq A_{\max});$$

$$\text{ctrl}_e \quad \begin{array}{l} a_e := *; ?(B_{\max} \leq a_e \leq A_{\max}); t_e := t; \\ \text{if } (\neg(\text{safeBack} \vee \text{safeFront})) \\ \quad \text{if } (x_e \leq x_o) \\ \quad \quad a_e := *; ?(B_{\max} \leq a_e \leq B_{\min}); \\ \quad \text{else} \\ \quad \quad a_e := *; ?(A_{\min} \leq a_e \leq A_{\max}); \end{array}$$

of acceleration are taken into account for the safety proof. Then the controller for the ego car also selects an arbitrary acceleration. It however performs an additional check. If the chosen acceleration does not satisfy one of the safety conditions, `safeBack` or `safeFront` discussed below, then a fallback procedure overrides the acceleration. The fallback simply tries to increase the distance with the other car. If the ego is behind, it brakes with $a_e \leq B_{\min}$, and accelerates, $a_e \geq A_{\min}$, if ahead. Finally, t_e is set to t to record the last time the controller ran.

Safety condition when behind. We focus on `safeBack` shown in Formula (1). It expresses when an acceleration guarantees safety when the ego car is behind the other car. First, the two cars should be at distance at least L from each other, as that would correspond to a collision otherwise. Additionally, if both cars were to brake, there should still be a distance at least L when they stop. For a braking ego car with acceleration $a_e < 0$, it stops at position $\text{pos}_e(a_e) \stackrel{\text{def}}{=} x_e - \frac{v_e^2}{2a_e}$ meters. For the other car, we assume the worst case. This happens when the other car's acceleration is directed towards the ego car, that is when it is braking at maximum force, $a_o = B_{\max}$, in which case it stops at position $\text{pos}_o \stackrel{\text{def}}{=} x_o - \frac{v_o^2}{2B_{\max}}$.

With constant acceleration, if the current position of the cars and their stopping position are both at safe distance, then these properties are invariants of the dynamics and thus ensure collision-freedom. Changing acceleration for the other car can only increase its distance to the ego car and so does not risk collision.

$$\begin{aligned} & x_e + L \leq x_o \wedge (a_e \leq B_{\min} \wedge \text{pos}_e(B_{\min}) + L < \text{pos}_o) \\ & \vee B_{\min} \leq a_e \wedge v_e + a_e T < 0 \wedge \text{pos}_e(a_e) + L < \text{pos}_o \\ & \vee B_{\min} \leq a_e \wedge v_e + a_e T \geq 0 \wedge \text{pos}_e(B_{\min}) + \text{corrDist} + L < \text{pos}_o \end{aligned} \quad (1)$$

To handle the ego car's change of acceleration, this idea is refined further and split in three scenarios:

1. Since the ego car is only assumed to be able to brake with $a_e = B_{\min}$ for sure, even if it is currently braking more, we still must rely on the minimum braking deceleration for checking the distance, so we use $\text{pos}_e(B_{\min})$.
2. If $a_e \geq B_{\min}$ but the car will stop before T seconds, then the acceleration a_e can be used directly. Once stopped, the car remains safe, so we use $\text{pos}_e(a_e)$.
3. Otherwise, we must check that the car can start braking at the next control cycle, after at most T seconds, and stop before crashing. This reuses the first case, with a correction term to account for the distance travelled and the speed change before the next cycle: $\text{corrDist} \stackrel{\text{def}}{=} (\frac{-a_e}{B_{\min}} + 1)(\frac{a_e}{2}T^2 + Tv_e)$.

Safety condition when ahead. If the ego car is ahead, the setting is similar when changing the frame of reference. From the perspective of an observer moving at constant speed V , the two cars are moving at speed $\tilde{v}_i \stackrel{\text{def}}{=} v_i - V$ in the opposite direction. Their positions are now $\tilde{x}_i \stackrel{\text{def}}{=} x_i - V \times t$, and the worst case occurs when the other car approaches the ego car with maximal acceleration (i.e. $a_o = A_{\max}$). Reusing the insight for the previous case, we consider their stopping position in that new frame of reference ($\tilde{v}_i = 0$), which amounts to reaching maximum speed ($v_i = V$). This gives the following distances updated with the new variables: $\tilde{\text{pos}}_e(a_e) \stackrel{\text{def}}{=} \tilde{x}_e - \frac{\tilde{v}_e^2}{2a_e}$ for the ego car, and $\tilde{\text{pos}}_o \stackrel{\text{def}}{=} \tilde{x}_o - \frac{\tilde{v}_o^2}{2A_{\max}}$ for the other. The resulting formula `safeFront` is given in Appendix A.

3.5 Safety Proofs

Now that the model is defined comes the actual verification. Since the goal is to prevent collisions, the safety condition is simply that the two cars have at least a distance L between them. Being on a single lane, they cannot cross each other, so the order of the cars remains the same, so the two cases when the ego car is behind or ahead can be proved independently. Due to their similarity, we again focus on the case where the ego car is behind. The general assumptions include the constraints from the specifications from Section 3.1, i.e. `ctx`, and assume the controller of the ego car has last been run T seconds ago so that it must run initially, i.e. $t_e = t - T$. The only other requirement is that initial states where a crash is unavoidable are prohibited, in which case, no controller can guarantee

safety. This the initial condition correspond to the first case of Formula (1): the fallback action should give enough distance before stopping.

Theorem 1. *Formulas (2) and (3) are valid and guarantee absence of collision.*

$$\text{ctx} \wedge x_e + L \leq x_o \wedge \text{pos}_e(B_{\min}) + L < \text{pos}_o \wedge t_e = t - T \rightarrow [\text{model}] x_e + L \leq x_o \quad (2)$$

$$\text{ctx} \wedge x_o + L \leq x_e \wedge \widetilde{\text{pos}}_o + L < \widetilde{\text{pos}}_e(A_{\min}) \wedge t_e = t - T \rightarrow [\text{model}] x_o + L \leq x_e \quad (3)$$

The theorem is proved using KeYmaera X. The proof relies on invariants that generalise of `safeBack` and `safeFront` where T is replaced by $T+t_e-t$ to account for the time elapsed since the last run of ctrl_e , extended with the specification constraints `ctx`. The evaluation of the two verifications is given in Appendix A.

4 Safeguarding Neural Control

The previous section derived a `dL` model for the highway environment as specified in ABZ’s case study document [16] and proved its safety. As a next step, we connect these (abstract) safety guarantees to the concrete control system implementation running inside the `highway-env` simulation [15]. To this end, we use the techniques described in Section 2.3. In contrast to the `dL` controller ctrl_e that chooses a (continuous) acceleration value $B_{\max} \leq a_e \leq A_{\max}$, the trained reinforcement learning agent for the single-lane case of `highway-env` consists of an NN outputting one of three discrete actions (brake, idle, accelerate). The NN outputs three values and determines its action via an argmax operation (e.g. brake is chosen whenever the NN’s first output is maximal), prioritising lowest speed in case of ties. Hence, we must first extend our `dL` controller model to account for the NN’s three outputs (Section 4.1). Subsequently, we can use ModelPlex and the refined controller to derive a formula that can be used for verification, shielding and monitoring (Section 4.2). While our methodology is general, this section focuses on the case where the ego car drives behind another car and must ensure safety.

4.1 Refining the `dL` Controller

To account for the concrete NN, we transform the controller’s action space from choosing an acceleration a_e to choosing an action via three outputs y_1, y_2, y_3 :

$$\text{ctrl}_{\text{NN}} \left| \begin{array}{l} y_1 := *; y_2 := *; y_3 := *; \\ \text{if}(y_1 \geq y_2 \wedge y_1 \geq y_3) \{a_e := *; ?(B_{\max} \leq a_e \leq B_{\min})\}; \\ \text{if}(y_2 > y_1 \wedge y_2 \geq y_3) \{a_e := 0\}; \\ \text{if}(y_3 > y_1 \wedge y_3 > y_2) \{a_e := *; ?(A_{\min} \leq a_e \leq A_{\max})\}; \\ \{ \quad ?(x_e \leq x_o \wedge B_{\max} \leq a_e \leq B_{\min}) \\ \quad \cup ?(x_e \geq x_o \wedge A_{\min} \leq a_e \leq A_{\max}) \\ \quad \cup ?(\text{safeBack} \vee \text{safeFront}) \}; t_e := t \end{array} \right.$$

Based on the NN’s outputs y_1, y_2, y_3 the program determines the corresponding

acceleration value a_e and then ensures safety via the checks we already know from the dL model for ctrl_e . To recover the formal guarantee from Formula (2), we show that $\text{model}(\text{ctrl}_{\text{NN}})$ refines $\text{model}(\text{ctrl}_e)$, i.e. $\text{model}(\text{ctrl}_{\text{NN}})$'s transitions are included in $\text{model}(\text{ctrl}_e)$'s. In fact, we prove a slightly relaxed refinement to ignore the variables y_1, y_2, y_3 that are modified by ctrl_{NN} and not ctrl_e .

Lemma 1. *The following refinement is valid:*

$$\text{ctx}_C \rightarrow (\text{model}(\text{ctrl}_{\text{NN}}) \leq (y_1 := *; y_2 := *; y_3 := *; \text{model}(\text{ctrl}_e))) \quad (4)$$

Using the refinement, it is then trivial to extend the proof of Formula (2) to $\text{model}(\text{ctrl}_{\text{NN}})$. The proof of refinement is done using KeYmaera X's differential refinement logic implementation¹ and is based on a proof of refinement between ctrl_{NN} and $y_1 := *; y_2 := *; y_3 := *; \text{ctrl}_e$.

4.2 ModelPlex for Safe Neural Network Control

We have now shown that any action taken by ctrl_{NN} keeps the system safe on an infinite-time horizon. Using ModelPlex we derive a controller monitor for ctrl_{NN} that we can use w.r.t. a concrete NN. To this end, we note that according to the specification [16] the NN has (among other inputs) a vector of inputs $= (x_e, v_e, x_o, v_o)$ and the NN's only output is a vector $\overline{\text{out}} = (y_1^+, y_2^+, y_3^+)$. Besides the variables in $\overline{\text{in}}$, $\overline{\text{out}}$ the controller monitor derived via ModelPlex also constrains the acceleration variables a_e and a_e^+ (as ctrl_{NN} modifies a_e) as well as the clock variables t, t_e^+ (required for book-keeping on control cycles). We denote this ModelPlex condition for ctrl_{NN} as $\text{mon}(\overline{\text{in}}, \overline{\text{out}}, a_e, a_e^+, t, t_0^+)$. As described in Section 2.3, VerSAILLE allows us to use the monitor mon to verify the safety of an NN by additionally exploiting the dL model's loop invariant which tells us what states are reachable (and thus for which states the NN must exhibit safe actions). We denote this invariant as $\text{inv}(\overline{\text{in}}, a_e, a_o, t, t_0)$. In addition to the two cars' positions and velocities, the invariant also mentions the cars' accelerations and the clock variables t, t_e . As explained in Section 2.3 we can prove the infinite-time horizon safety of an NN by showing that all inputs inside the invariant inv lead to outputs satisfying the controller monitor mon . Formally, this can be expressed as the following Theorem which follows from [34, Thm. 2]:

Theorem 2 (NNCS Safety Criterion). *Let g be an NN for highway car control as modeled in Section 3. If Formula (5) is satisfied for all $\overline{\text{in}}$ and $\overline{\text{out}} = g(\overline{\text{in}})$ then the safety guarantees derived in Theorem 1 apply to $\text{model}(g)$.*

$$\forall a_e, a_e^+, a_o, t, t_e, t_e^+ \underbrace{\text{inv}(\overline{\text{in}}, a_e, a_o, t, t_0)}_{\text{system invariant}} \rightarrow \underbrace{\text{mon}(\overline{\text{in}}, \overline{\text{out}}, a_e, a_e^+, t, t_0^+)}_{\text{monitoring formula}} \quad (5)$$

While this work omits the precise formulation, it is worth noting that the safety guarantees for g are rigorously founded in dL via a reconstruction of g inside dL through the notion of *nondeterministic mirrors* [34, Def. 16].

¹ <https://github.com/LS-Lab/KeYmaeraX-release/tree/dRL-ABZ'25>

Unfortunately, Formula (5) cannot effectively be used for the NN verification directly as the NNs do not set the ego-cars acceleration (a_e^+) but rely on surrounding software which computes a_e^+ based of y_1, y_2, y_3 (and resets the clock variable t_e). Moreover,

a_e, a_o, t and t_e are no inputs to the NN and would thus need to be quantified over. To make our verification condition practical, we derive a simplified version that we prove equivalent to Formula (5). To this end, we begin by axiomatizing our assumptions on the NN's surroundings. We assume the software correctly assigns a_e based on y_1, y_2, y_3 , correctly manages clock variables and that we drive behind the other car (as mentioned above, we focus on this case). We also set $A_{\min} = A_{\max}$ (as done in the official ABZ specifacaton [16]) and assume the known ranges of constants as formalized in Figure 1. Assuming nnCtx , the system's invariant can then be simplified as follows:

$$\text{inv}_{\text{simp}} \stackrel{\text{def}}{=} 0 \leq v_o \leq V \wedge 0 \leq v_e \leq V \wedge x_e + L \leq x_o \wedge \text{pos}_e(B_{\min}) + L < \text{pos}_o$$

The simplified invariant makes sense intuitively as it matches the initial condition constraints in Formula (2) on the variables in $\overline{\text{in}}$. Similarly, we simplify mon by removing cases irrelevant to the ego-car driving behind, the management of clock variables and explicit mentions of a_e^+ . This yields a simplified formula $\text{mon}_{\text{simp}}(\overline{\text{in}}, \overline{\text{out}})$ (see Appendix A). For these simplifications, we prove equivalence to Formula (5) in KeYmaera X under the assumption of nnCtx :

Lemma 2 (Simplified NN Verification). *The following formula is valid:*

$$\begin{aligned} \text{nnCtx}(\overline{\text{in}}, \overline{\text{out}}, a_e^+, t_0, t_0^+, t) \rightarrow \\ \left(\underbrace{\left(\begin{array}{l} \text{inv}_{\text{simp}}(\overline{\text{in}}) \rightarrow \\ \text{mon}_{\text{simp}}(\overline{\text{in}}, \overline{\text{out}}) \end{array} \right)}_{\text{simplified}} \right) \leftrightarrow \underbrace{\left(\begin{array}{l} \forall a_e \forall a_o \left((\text{inv}(\overline{\text{in}}, a_e, a_o, t, t_0)) \right. \\ \left. \rightarrow \text{mon}(\overline{\text{in}}, \overline{\text{out}}, a_e, a_e^+, t, t_0^+) \right) \end{array} \right)}_{\text{Formula (5)}} \end{aligned}$$

This serves as justification for verifying the simplified condition $\text{nnSpec}_{\text{simp}} \stackrel{\text{def}}{=} \text{inv}_{\text{simp}}(\overline{\text{in}}) \rightarrow \text{mon}_{\text{simp}}(\overline{\text{in}}, \overline{\text{out}})$ on our NNs as we can assume nnCtx . While $\text{nnSpec}_{\text{simp}}$ is free of quantifiers, it still contains polynomial arithmetic (e.g. in $\text{pos}_e(B_{\min})$). In addition to the two cars modelled in dL , the NN controller gets as input the states of up to three more cars (we will call these cars car 1 to car 5 with car 1 being the ego car). For the single-lane case, the ego car's influence on crashes with cars 3-5 is very limited. However, we know that car 2 can avoid a crash with car 3 if the velocity of car 3 is larger than the velocity of car 2 (e.g. by performing an emergency brake). For now, we thus assume that for the extra cars $3 \leq i \leq 5$ it is guaranteed that car $i-1$ is slower than car i . We thus encode these additional

$$\begin{aligned} \text{nnCtx}(\overline{\text{in}}, \overline{\text{out}}, a_e^+, t_0, t_0^+, t) &\stackrel{\text{def}}{=} \\ y_1^+ \geq y_2^+ \wedge y_1^+ \geq y_3^+ \rightarrow B_{\max} \leq a_e^+ \leq B_{\min} \wedge \\ y_2^+ > y_1^+ \wedge y_2^+ \geq y_3^+ \rightarrow a_e^+ = 0 \wedge \\ y_3^+ > y_1^+ \wedge y_3^+ > y_2^+ \rightarrow a_e^+ = A_{\max} \wedge \\ x_e + L \leq x_o \wedge t_e^+ = t \wedge t_e \leq t \leq t_e + T \wedge \text{ctx}_C \end{aligned}$$

Fig. 1. Context assumptions for simplification

constraints on the state of cars 3–5 in a predicate $\text{nnSpec}_{\text{add}}$ (see Appendix A) and then verify the NN w.r.t. to the specification $\text{nnSpec}_{\text{add}} \rightarrow \text{nnSpec}_{\text{simp}}$. $\text{nnSpec}_{\text{add}}$ also contains constraints on the encoding of (non-)presence of cars and the NN’s input space normalisation described in ABZ’s specification [16]. In Section 5 we will see concrete examples for verifying NNs with respect to the full specification $\text{nnSpec}_{\text{add}} \rightarrow \text{nnSpec}_{\text{simp}}$, but we will first demonstrate that similar formulas can also be used for monitoring and shielding.

Justified Speculative Control and VeriPhy. Assuming nnCtx and inv_{simp} , it also holds that $\text{mon}_{\text{simp}}(\overline{\text{in}}, \overline{\text{out}}) \leftrightarrow \text{mon}(\overline{\text{in}}, \overline{\text{out}}, a_e, a_e^+, t, t_0^+)$. Consequently, we can use the simplified monitoring condition not only for verification, but also for the construction of shields (JSC) and runtime monitors (VeriPhy). JSC is meant to only check the runtime monitor when the observed behaviour matches the model. To this end, JSC usually has a model monitor that checks whether a given state transition is explainable by the dL environment model. However, early experiments showed divergence in the simulation’s environment and the dL environment model which would effectively deactivate JSC in most of the state space (these observations will be discussed in more detail in the latter sections). Consequently, we relaxed the model monitor to its “most” safety-critical parts and only check whether a given state is inside the invariant state space.

In this section, we have seen how ModelPlex conditions and invariants can be applied even if they contain unobservable variables [23]. This allows us to apply dL-based monitoring, shielding and verification techniques independently of whether some variables (e.g. time or effective acceleration) are measurable or not.

5 Verification Results

A manual analysis of the agents in ABZ’s case study [16] uncovered that the agents’ action space is different from its formal specification [16]: The `highway-env` simulator configuration admits different action spaces. The provided agents used `DiscreteMetaAction` configuring the agent’s action space as decreasing/increasing a *reference velocity* $v_r \in \{0, 5, \dots, 35, 40\}$ achieved via a low-level proportional controller. This can lead to very different action outcomes compared to the specified action space. For example, the *brake* action as described in the formal specification *always* leads to a deceleration (unless v_e is zero already). In contrast, the “brake” action with the `DiscreteMetaAction` configuration can even lead to an *acceleration* (e.g. if $v_e = 10$ and $v_r = 20$, “braking” sets v_r to 15 and the proportional controller accelerates so that $v_e = 15$ is reached). The safety guarantees and verification conditions derived in Sections 3 and 4 thus only apply to the written specification, but not to the simulator’s default configuration violating its own description. We trained a new NN using `highway-env`’s `DiscreteAction` configuration option² (otherwise using the default configuration) that can brake, idle or accelerate *directly*

Table 1. Results of NN verification.

| NN | Time | # Crashes | |
|-------------|-------|-----------|---------|
| | | default | braking |
| Section 5.1 | 3.6 h | 538 | 3,593 |
| Section 5.2 | 1.9 h | 4,852 | 8,713 |

$(a_e \in \{B_{\max}, 0, A_{\max}\})$. We discuss verification of this NN (Section 5.1) and an improved version (Section 5.2). Results are reproducible via our artifact [31].

5.1 A First Attempt at Verification

As a first step we attempted to verify the NN for two cars w.r.t. the specification derived in Section 4 using NCubeV [34] which supports polynomial arithmetic specifications. In case a specification cannot be proven, NCubeV is also capable of enumerating *all* counterexample regions (represented as polytopes) to a given specification. Notably, successful verification would, by construction, guarantee that the two cars on the highway will never crash – independently of trip time. The trained NN instead turned out to be unsafe: NCubeV returned **14,917 counterexample regions**. Computing these counterexamples took 3.6 hours (see Table 1). However, verifying safety is often quicker than enumerating all counterexamples for NCubeV [34]. Each counterexample region has a representative input violating the specification. These inputs can be used to sample trajectories from the simulator to find concrete crashes. Figure 2 shows one of 538 concrete crashes we observed when the front car is controlled by the *Intelligent Driver Model* [36] (IDM). Importantly, these crashes could have all been avoided by braking. When the front car is configured to perform an emergency brake, our sampling strategy yielded 3,593 crash trajectories.

These observations raise two questions: 1. Why did the NN not learn to brake in time? 2. Is there nonetheless a way to safely deploy the NN at hand? One answer to the former question can be found in the IDM. While originally derived as a means to understand traffic congestion, `highway-env` uses the model to control the environment’s cars. Due to the way IDM is set up, the ego-car rarely experiences emergency brakes of front cars and thus does not learn to account for them (as indicated by over 3,000 crash trajectories for emergency braking front cars). The `highway-env` simulation environment is thus another example of a previously observed phenomenon that worst-case scenarios which occur with low probabilities during training are typically not learned by reinforcement learning agents and that these errors can be uncovered by formal verification [34]. In the present environment, this issue is exacerbated by the fact that the agent learns that it can brake with acceleration $a_e = B_{\max}$ although (according to the specification) the acceleration can be as little as B_{\min} .

We now turn to the question of how the NN can be safeguarded under the given conditions. To this end, we evaluated the NN’s empirical performance (reward) and crash behaviour w.r.t. the IDM front-car (`default`) as well as w.r.t. an

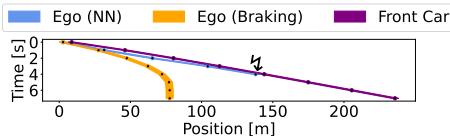


Fig. 2. One of 538 examples of unsafe NN behaviour in `default` environment (x-axis shows position, y-axis shows time). Braking could have avoided a crash (↓).

² Python’s weak type system makes the configuration especially error-prone: The `acceleration_range` is configured via a *2-tuple* (`min, max`). Accidentally providing a *list* of actions interpolates discrete accelerations between the list’s first two elements.

Table 2. Empirical results for the original, monitored (VeriPhy) and shielded (JSC) controller given initial conditions *inside* the safely controllable (i.e. invariant) state space. The velocity bounds of JSC’s invariant check had to be modified as the simulator occasionally produces velocities outside $[0, V]$ which would otherwise deactivate JSC.

| Env | Original NN | | VeriPhy | | JSC* | |
|---------------|-------------------------|------------|-------------------------|------------|-------------------------|------------|
| | Reward | Crash | Reward | Crash | Reward | Crash |
| default (IDM) | 17.63 \pm 0.21 | 0 % | 16.72 \pm 0.32 | 0 % | 17.63 \pm 0.21 | 0 % |
| braking | 5.44 \pm 1.27 | 99.6% | 16.47 \pm 0.05 | 0 % | 16.47 \pm 0.05 | 0 % |

environment where the front car performs emergency brakes and the ego car can only decelerate with B_{\min} (**braking**). We evaluate the stand-alone NN, a monitored version using a Python implementation of the VeriPhy approach (this implementation comes without the rigorous compilation guarantees of VeriPhy [3]) and a shield for the NN using JSC [8]. We evaluate w.r.t. initial conditions satisfying the invariant over 1000 sampled trajectories³. The results are in Table 2 and indicate relatively consistent behaviour w.r.t. to the reward standard deviation. Empirically, we observe that the agent trained w.r.t. to the IDM model (**default** environment) crashes in 996 out of 1000 cases when evaluated w.r.t. a braking front car (**braking** environment). Our investigation indicates that the dynamics in **default** lack diversity in at least three dimensions: First, the environment assumes maximal braking power (contradicting the formal specification [16]); Secondly, the environment very rarely simulates braking front cars. Finally, we posit that **default** only samples initial conditions from a small subset of admissible initial conditions as our verifier found many concrete initial conditions that lead to crashes in **default**. Importantly, VeriPhy and JSC allow us to (probably!) avoid these crashes by intervening when the model chooses unsafe actions. We observe that, based on the reward function, JSC matches the best results across both environments while leading to 0 crashes. VeriPhy’s and JSC’s behaviour differs in their statistics on taken actions: For example, JSC chooses the idle action in 6.3% of time steps while VeriPhy never chooses this action.

5.2 An Improved NN Controller

Based on the results from Section 5.1 we attempted to train a second agent. To this end, we also modified the training. First, we enforce that 80% of initial states satisfy the invariant (like above, we achieve this by sampling with reduced car density). Second, we modified the behaviour of environment variables: In each control round a car initiates (and then continues in subsequent steps) an emergency brake with 15% probability. Our objective is to increase the likelihood of the agent experiencing worst-case behaviour of the environment as a learning opportunity – especially in situations where crashes can be avoided. Finally, as NN verification and counterexample region enumeration scales exponentially

³ Initial conditions are generated via rejection sampling. For a sufficiently high success rate, we had to reduce the simulator’s car density parameter.

with the NN’s size, we reduce the NN to two layers with 16 neurons each. Unlike the provided environment (20k steps) we train for up to 40k steps and choose the best-performing model (achieved after 22k steps). To simplify the task, we furthermore assume $B_{\min} = B_{\max} = -5.0$. An evaluation across 1,000 initial conditions for `braking` (with $B_{\min} = -5$) yielded a reward of 16.08 ± 0.07 with 0 crashes. Compared to the first NN’s performance for the `braking` environment in Table 2 this is a notable performance improvement. Given these promising results, we attempted verification w.r.t. the full NN specification (2 to 5 cars).

Verification took 1.9 hours and still returned **11,059 counterexample regions**. Simulations with the representative inputs for the returned regions uncovered 4852 crashes in the `default` simulation (using IDM) and 8713 crashes in the `braking` simulation (with $B_{\max} = B_{\min} = -5$; see Table 1). Surprisingly, for the two simulations we resp. found 181 and 40 cases that even produced a crash when the ego-car performed an emergency brake! This is surprising as our `dL` proof states that braking should keep our system safe. A closer examination uncovered that these are *Euler Crashes*, i.e. the occurrence of a crash depends on the resolution of the Euler approximation. For a finer step size of the Euler approximation, the spurious crash disappears. Importantly, in almost all cases the crash produced by the NN remained. An example for an Euler Crash (evaluated with 10 and 100 Euler steps per second of evolution) can be found in Figure 3.

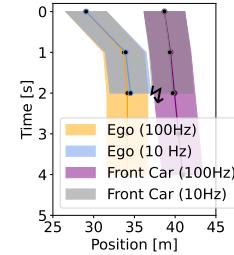


Fig. 3. An *Euler Crash* (\notin): Occurrence depends on Euler approximation resolution.

all cases the crash produced by the NN remained.

6 The Model2Simulation Gap

Overall, this work has not only derived an abstract `dL` model, but also demonstrated in practice that verification can serve as a powerful tool to detect flaws in reinforcement learning systems. Across two NNs our analysis uncovered numerous concrete counterexamples for NNs even though they performed *flawlessly* in their respective simulations. Throughout, we attempted to trace these faults to design choices in the simulator such as the intelligent driver model or the sampling method for choosing initial conditions. Overall, our results provide strong evidence that *as is* the `highway-env` simulator provides no reliable basis for the training of safe car control NNs. However, we believe the detected issues point to a larger issue concerning inconsistencies between models and simulators in general. While we consistently took the stance that our model is correct and the simulation is to blame, in reality, this is not always the case: For example, was it justified that we changed the NN’s action space or should we have built an entirely different KeYmaera X model? Here, we believe our choice was justified by ABZ’s specification document [16], but such documentation may not always be available. While this work demonstrates how far `dL`-based safety certification for NN Control Systems has come, it also underscores the intricate issues of interlinking simulation-based evaluation with a symbolic, `dL`-based analysis.

Acknowledgements. This work was supported by funding from the pilot program Core-Informatics of the Helmholtz Association (HGF) and by an Alexander von Humboldt Professorship.

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010). <https://doi.org/10.1017/CBO9781139195881>
2. Banach, R., Butler, M.J.: Cruise control in hybrid Event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theoretical Aspects of Computing - ICTAC 2013 - 10th International Colloquium, Shanghai, China, September 4-6, 2013. Proceedings. LNCS, vol. 8049, pp. 76–93. Springer (2013). https://doi.org/10.1007/978-3-642-39718-9_5
3. Bohrer, R., Tan, Y.K., Mitsch, S., Myreen, M.O., Platzer, A.: VeriPhy: verified controller executables from verified cyber-physical system models. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 617–630. ACM (2018). <https://doi.org/10.1145/3192366.3192406>
4. Brix, C., Bak, S., Johnson, T.T., Wu, H.: The fifth international verification of neural networks competition (VNN-COMP 2024): Summary and results. CoRR **abs/2412.19985** (2024). <https://doi.org/10.48550/ARXIV.2412.19985>
5. Brix, C., Müller, M.N., Bak, S., Johnson, T.T., Liu, C.: First three years of the international verification of neural networks competition (VNN-COMP). Int. J. Softw. Tools Technol. Transf. **25**(3), 329–339 (2023). <https://doi.org/10.1007/s10009-023-00703-4>
6. Demarchi, S., Guidotti, D., Pulina, L., Tacchella, A.: Supporting standardization of neural networks verification with VNNLIB and coconet. In: Narodytska, N., Amir, G., Katz, G., Isac, O. (eds.) Proceedings of the 6th Workshop on Formal Methods for ML-Enabled Autonomous Systems, FoMLAS@CAV 2023, Paris, France, July 17-18, 2023. Kalpa Publications in Computing, vol. 16, pp. 47–58. EasyChair (2023). <https://doi.org/10.29007/5PDH>
7. Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In: Felty, A., Middeldorp, A. (eds.) CADE. LNCS, vol. 9195, pp. 527–538. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-21401-6_36
8. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. pp. 6485–6492. AAAI Press (2018). <https://doi.org/10.1609/aaai.v32i1.12107>
9. Fulton, N., Platzer, A.: Verifiably safe off-model reinforcement learning. In: Vojnar, T., Zhang, L. (eds.) TACAS, Part I. LNCS, vol. 11427, pp. 413–430. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_28
10. Garcia, L., Mitsch, S., Platzer, A.: HyPLC: Hybrid programmable logic controller program translation for verification. In: Bushnell, L., Pajic, M. (eds.) ICCPS. pp. 47–56 (2019). <https://doi.org/10.1145/3302509.3311036>
11. Kabra, A., Laurent, J., Mitsch, S., Platzer, A.: CESAR: Control envelope synthesis via angelic refinements. In: Finkbeiner, B., Kovács, L. (eds.) TACAS. LNCS, vol.

- 14570, pp. 144–164. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_9
12. Kamburjan, E., Mitsch, S., Hähnle, R.: A hybrid programming language for formal modeling and verification of hybrid systems. *Leibniz Trans. Embed. Syst.* **8**(2), 04:1–04:34 (2022). <https://doi.org/10.4230/LITES.8.2.4>
 13. König, M., Bosman, A.W., Hoos, H.H., van Rijn, J.N.: Critically assessing the state of the art in neural network verification. *J. Mach. Learn. Res.* **25**, 12:1–12:53 (2024), <https://jmlr.org/papers/v25/23-0119.html>
 14. Könighofer, B., Bloem, R., Ehlers, R., Pek, C.: Correct-by-construction runtime enforcement in AI - A survey. In: Raskin, J., Chatterjee, K., Doyen, L., Majumdar, R. (eds.) *Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*. LNCS, vol. 13660, pp. 650–663. Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_31
 15. Leurent, E.: An environment for autonomous driving decision-making. <https://github.com/eleurent/highway-env> (2018)
 16. Leuschel, M., Vu, F., Rutenkolk, K.: Case study: Safety controller for autonomous driving on highways (v2) (2024), https://raw.githubusercontent.com/hhu-stups/abz2025_casestudy_autonomous_driving/refs/heads/main/case_study/specification_v2.pdf, v2, accessed 11th of February 2025
 17. Loos, S.M., Platzer, A.: Differential refinement logic. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) *LICS*. pp. 505–514. ACM, New York (2016). <https://doi.org/10.1145/2933575.2934555>
 18. Loos, S.M., Platzer, A., Nistor, L.: Adaptive cruise control: Hybrid, distributed, and now formally verified. In: Butler, M., Schulte, W. (eds.) *FM*. LNCS, vol. 6664, pp. 42–56. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-21437-0_6
 19. Lopez, D.M., Althoff, M., Benet, L., Blab, C., Forets, M., Jia, Y., Johnson, T.T., Kranzl, M., Ladner, T., Linauer, L., Neubauer, P., Neubauer, S., Schilling, C., Zhang, H., Zhong, X.: ARCH-COMP24 category report: AINNCS for continuous and hybrid systems plants. In: Frehse, G., Althoff, M. (eds.) *Proceedings of the 11th Int. Workshop on Applied Verification for Continuous and Hybrid Systems*. EPiC Series in Computing, vol. 103, pp. 64–121. EasyChair (2024). <https://doi.org/10.29007/mxld>
 20. Lopez, D.M., Althoff, M., Benet, L., Chen, X., Fan, J., Forets, M., Huang, C., Johnson, T.T., Ladner, T., Li, W., et al.: ARCH-COMP22 category report: AINNCS for continuous and hybrid systems plants. In: *Proceedings of 9th International Workshop on Applied*. vol. 90, pp. 142–184 (2022). <https://doi.org/10.29007/wfgr>
 21. Lopez, D.M., Althoff, M., Forets, M., Johnson, T.T., Ladner, T., Schilling, C.: ARCH-COMP23 category report: (AINNCS) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M. (eds.) *Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23)*, San Antonio, Texas, USA, May 9, 2023. EPiC Series in Computing, vol. 96, pp. 89–125. EasyChair (2023). <https://doi.org/10.29007/X38N>
 22. Mitsch, S., Platzer, A.: ModelPlex: verified runtime validation of verified cyber-physical system models. *Formal Methods Syst. Des.* **49**(1-2), 33–74 (2016). <https://doi.org/10.1007/s10703-016-0241-z>
 23. Mitsch, S., Platzer, A.: Verified runtime validation for partially observable hybrid systems. *CoRR* **abs/1811.06502** (2018), <http://arxiv.org/abs/1811.06502>
 24. Mitsch, S., Platzer, A., Fulton, N., Bohrer, R., Kiam, Y., Immler, F., Quesel, J.D., Ji, R., Gallicchio, J., Völp, M., Prebet, E., Sogokon, A., LSLabBuild, Erthal, T., Kabra, A., Kosaian, K., Laurent, J.: LS-Lab/KeYmaeraX-release: Version 5.1.1 (Jul 2024). <https://doi.org/10.5281/zenodo.13380145>

25. Platzer, A.: Differential dynamic logic for hybrid systems. *J. Autom. Reas.* **41**(2), 143–189 (2008). <https://doi.org/10.1007/s10817-008-9103-8>
26. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010). <https://doi.org/10.1007/978-3-642-14509-4>
27. Platzer, A.: A complete axiomatization of quantified differential dynamic logic for distributed hybrid systems. *Log. Meth. Comput. Sci.* **8**(4:17), 1–44 (2012). [https://doi.org/10.2168/LMCS-8\(4:17\)2012](https://doi.org/10.2168/LMCS-8(4:17)2012), special issue for selected papers from CSL’10
28. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.* **59**(2), 219–265 (2017). <https://doi.org/10.1007/s10817-016-9385-1>
29. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer, Cham (2018). <https://doi.org/10.1007/978-3-319-63588-0>
30. Prebet, E., Platzer, A.: Uniform substitution for differential refinement logic. In: Benzmüller, C., Heule, M.J., Schmidt, R.A. (eds.) IJCAR. LNCS, vol. 14740, pp. 196–215. Springer (2024). https://doi.org/10.1007/978-3-031-63501-4_11
31. Prebet, E., Teuber, S., Platzer, A.: LS-Lab/verified-neural-highway-control: 1.0 (Mar 2025). <https://doi.org/10.5281/zenodo.14959858>
32. Renshaw, D.W., Loos, S.M., Platzer, A.: Distributed theorem proving for distributed hybrid systems. In: Qin, S., Qiu, Z. (eds.) ICFEM. LNCS, vol. 6991, pp. 356–371. Springer (2011). https://doi.org/10.1007/978-3-642-24559-6_25
33. Shperberg, S.S., Liu, B., Allievi, A., Stone, P.: A rule-based shield: Accumulating safety rules from catastrophic action effects. In: Chandar, S., Pascanu, R., Precup, D. (eds.) Conference on Lifelong Learning Agents, CoLLAs 2022, 22-24 August 2022, McGill University, Montréal, Québec, Canada. Proceedings of Machine Learning Research, vol. 199, pp. 231–242. PMLR (2022)
34. Teuber, S., Mitsch, S., Platzer, A.: Provably safe neural network controllers via differential dynamic logic. In: Globerson, A., Mackey, L., Fan, A., Zhang, C., Belgrave, D., Tomczak, J., Paquet, U. (eds.) Advances in Neural Information Processing Systems. Curran Associates, Inc. (2024), <https://doi.org/10.48550/arXiv.2402.10998>
35. Tran, H., Cai, F., Lopez, D.M., Musau, P., Johnson, T.T., Koutsoukos, X.D.: Safety verification of cyber-physical systems with reinforcement learning control. *ACM Trans. Embed. Comput. Syst.* **18**(5s), 105:1–105:22 (2019). <https://doi.org/10.1145/3358230>
36. Treiber, M., Hennecke, A., Helbing, D.: Congested traffic states in empirical observations and microscopic simulations. *Physical review E* **62**(2), 1805 (2000). <https://doi.org/10.1103/PhysRevE.62.1805>
37. Vu, F., Dunkelau, J., Leuschel, M.: Validation of reinforcement learning agents and safety shields with prob. In: Benz, N., Gopinath, D., Shi, N. (eds.) NASA Formal Methods - 16th International Symposium, NFM 2024, Moffett Field, CA, USA, June 4-6, 2024, Proceedings. LNCS, vol. 14627, pp. 279–297. Springer (2024). https://doi.org/10.1007/978-3-031-60698-4_16

A Additional Figures

$$\begin{aligned}
& x_o + L \leq x_e \wedge (A_{\min} \leq a_e \wedge \overline{\text{pos}}_o + L < \overline{\text{pos}}_e(A_{\min})) \\
& \vee a_e \leq A_{\min} \wedge \tilde{v}_e + a_e T > 0 \wedge \overline{\text{pos}}_o + L < \overline{\text{pos}}_e(a_e) \\
& \vee a_e \leq A_{\min} \wedge \tilde{v}_e + a_e T \leq 0 \wedge \overline{\text{pos}}_o + L < \overline{\text{pos}}_e(A_{\min}) + \left(\frac{-a_e}{A_{\min}} + 1 \right) \left(\frac{a_e}{2} T^2 + \tilde{v}_e T \right)
\end{aligned}$$

Fig. 4. safeFront

Table 3. Verification using KeYmaera X: the number of steps is the size of the implicit proof tree from the kernel including lemmas. QE time is included in the total duration.

| Proof | | Status | Tactic Size | Duration [ms] | QE [ms] | Steps |
|--|--|--------|-------------|---------------|---------|--------|
| safeBack invariant of <code>dyn</code> | | proved | 48 | 21,362 | 10,648 | 15,378 |
| safeFront invariant of <code>dyn</code> | | proved | 97 | 14,471 | 9,858 | 17,456 |
| <code>model(ctrl_e) - Back (2)</code> | | proved | 84 | 14,701 | 8,241 | 21,068 |
| <code>model(ctrl_e) - Front (3)</code> | | proved | 110 | 16,407 | 6,865 | 25,689 |
| Controllers Refinement | | proved | 143 | 4,093 | 1,875 | 2,446 |
| Models Refinement (4) | | proved | 88 | 2,878 | 0 | 3,716 |
| <code>model(ctrl_{NN}) - Back</code> | | proved | 8 | 1,249 | 0 | 24942 |
| ModexPlex simp (Lemma 2) | | proved | 326 | 104,634 | 70,533 | 11902 |

$$\begin{aligned}
& y_1^+ \geq y_2^+ \wedge y_1^+ \geq y_3^+ \\
& \vee y_2^+ > y_1^+ \wedge y_2^+ \geq y_3^+ \wedge \\
& \quad (B_{\min} \leq 0 \leq A_{\max} \wedge v_e \geq 0 \wedge \text{pos}_e(B_{\min}) + (\frac{0}{B_{\min}} + 1)T v_e + L < \text{pos}_o) \\
& \vee y_3^+ > y_1^+ \wedge y_3^+ > y_2^+ \wedge (B_{\max} \leq A_{\max} \leq B_{\min} \wedge \text{pos}_e(B_{\min}) + L < \text{pos}_o \\
& \quad \vee B_{\min} \leq A_{\max} \wedge v_e + A_{\max}T < 0 \wedge \text{pos}_e(A_{\max}) + L < \text{pos}_o \\
& \quad \vee B_{\min} \leq A_{\max} \wedge v_e + A_{\max}T \geq 0 \wedge \text{pos}_e(B_{\min}) + (\frac{-A_{\max}}{B_{\min}} + 1)(\frac{A_{\max}}{2}T^2 + T v_e) + L < \text{pos}_o)
\end{aligned}$$

Fig. 5. $\text{mon}_{\text{simp}}(\overline{\text{in}}, \overline{\text{out}})$

$$\begin{aligned}
& p_e = 1 \quad \wedge 0 \leq \frac{x_e}{5 * V} \leq 1 \wedge 0 \leq \frac{v_e}{2 * V} \leq 1 \quad \wedge y_e = 0 \wedge w_e = 0 \wedge \\
& p_o = 1 \quad \wedge -1 \leq \frac{x_o - x_e}{5 * V} \leq 1 \wedge 0 \leq \frac{v_o - v_e}{2 * V} \leq 1 \wedge y_o = 0 \wedge w_o = 0 \wedge \\
& 0 \leq p_3 \leq 1 \wedge -1 \leq \frac{x_3 - x_e}{5 * V} \leq 1 \wedge 0 \leq \frac{v_3 - v_e}{2 * V} \leq 1 \wedge y_3 = 0 \wedge w_3 = 0 \wedge \\
& 0 \leq p_4 \leq 1 \wedge -1 \leq \frac{x_4 - x_e}{5 * V} \leq 1 \wedge 0 \leq \frac{v_4 - v_e}{2 * V} \leq 1 \wedge y_4 = 0 \wedge w_4 = 0 \wedge \\
& 0 \leq p_5 \leq 1 \wedge -1 \leq \frac{x_5 - x_e}{5 * V} \leq 1 \wedge 0 \leq \frac{v_5 - v_e}{2 * V} \leq 1 \wedge y_5 = 0 \wedge w_5 = 0 \wedge \\
& -1000 \leq y_1 \leq 1000 \wedge -1000 \leq y_2 \leq 1000 \wedge -1000 \leq y_3 \leq 1000 \wedge \\
& (p_3 = 0 \vee p_3 = 1) \wedge (p_4 = 0 \vee p_4 = 1) \wedge (p_5 = 0 \vee p_5 = 1) \wedge \\
& (p_3 = 0 \rightarrow (x_3 = 0 \wedge v_3 = 0)) \wedge \\
& (p_4 = 0 \rightarrow (x_4 = 0 \wedge v_4 = 0)) \wedge \\
& (p_5 = 0 \rightarrow (x_5 = 0 \wedge v_5 = 0)) \wedge \\
& (p_3 = 1 \rightarrow (x_o + L \leq x_3 \wedge v_o \leq v_3)) \wedge \\
& (p_4 = 1 \rightarrow (x_3 + L \leq x_4 \wedge v_3 \leq v_4 \wedge p_3 = 1)) \wedge \\
& (p_5 = 1 \rightarrow (x_4 + L \leq x_5 \wedge v_4 \leq v_5 \wedge p_4 = 1))
\end{aligned}$$

Fig. 6. $\text{nnSpec}_{\text{add}}$: p is the presence indicator, y the latitudinal position and w the latitudinal velocity. Additionally, we must configure the verifier so that the inputs are considered w.r.t. their normalized value, e.g. the ego car position input has to be $\frac{x_e}{5V}$. Moreover, the other car inputs are relative to the front car, e.g. the front car position is $\frac{x_o - x_e}{5V}$. Finally, we instantiate the constants ($V = 40, T = 1, L = 5, B_{\max} = -5, A_{\max} = 5, A_{\min} = 5$ and depending on query $B_{\min} \in \{-3, -5\}$).