Verification-guided Programmatic Controller Synthesis

Yuning Wang and He Zhu(✉)

Rutgers University, New Brunswick NJ, USA
{yw895,hz375}@cs.rutgers.edu

Abstract. We present a verification-based learning framework VEL that synthesizes safe programmatic controllers for environments with continuous state and action spaces. The key idea is the integration of program reasoning techniques into controller training loops. VEL performs abstraction-based program verification to reason about a programmatic controller and its environment as a closed-loop system. Based on a novel verification-guided synthesis loop for training, VEL minimizes the amount of safety violation in the proof space of the system, which approximates the *worst-case* safety loss, using gradient-descent style optimization. Experimental results demonstrate the substantial benefits of leveraging verification feedback for synthesizing provably correct controllers.

1 Introduction

Controller search is commonly used to govern cyber-physical systems such as autonomous vehicles, where high assurance is particularly important. Reinforcement Learning (RL) of neural network controllers is a promising approach for controller search [19]. State-of-the-art RL algorithms can learn motor skills autonomously through trial and error in simulated or even unknown environments, thus avoiding tedious manual engineering. However, well-trained neural network controllers may still be unsafe since the RL algorithms do not provide any formal guarantees on safety. A learned controller may fail occasionally but catastrophically, and debugging these failures can be challenging [46].

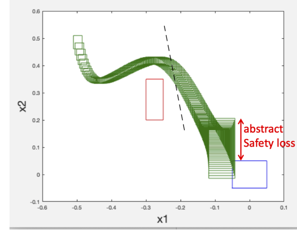
Guaranteeing the correctness of an RL controller is therefore important. Principally, given an environment model, the correctness of a controller can be verified by reachability analysis over a closed-loop system that combines the environment model and the controller. Indeed, the use of formal verification techniques to aid the design of reliable learning-enabled autonomous systems has risen rapidly over the last few years [43,28,41,18,17]. A natural extended question is that in case verification fails, can we exploit verification feedback in the form of counterexamples to synthesize a verifiably correct controller? This turns out to be a very challenging task due to the following reasons.

Verification Scalability. A counterexample-guided controller synthesizer has to *iteratively* conduct reachability analysis and controller optimization as each

© The Author(s) 2023
S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 229–250, 2023.
https://doi.org/10.1007/978-3-031-30820-8_16

if $28.33x_1 + 4.23x_2 + 4.16 \geq 0$
    then $6.79x_1 - 8.56x_2 + 0.35$
    else $11.01x_1 - 13.50x_2 + 8.71$

(a) Oscillator Programmatic Controller



(b) Oscillator Reachability Analysis

Fig. 1: An oscillator programmatic controller and its reachability analysis. In Fig. 1b, the red region represents the oscillator unsafe set $(-03, -0.25) \times (0.2, 0.35)$, and the blue region depicts the target set $[-0.05, 0.05] \times [-0.05, 0.05]$. The initial state set of oscillator is $[-0.51, -0.49] \times [0.49, 0.51]$.

iteration may discover a new counterexample. However, repeatedly calculating the reachable set of a nonlinear system controlled by a neural network controller over a long horizon is computationally challenging. For example, consider designing a controller for the Van der Pol's oscillator system [49]. The oscillator is a 2-dimensional non-linear system whose state transition can be expressed by the following ordinary differential equations:

$$\dot{x_1} = x_2 \qquad \dot{x_2} = (1 - x_1^2)x_2 - x_1 + u \tag{1}$$

where (x_1, x_2) is the system state variables and u is the control action variable. A feedback controller $\pi(x_1, x_2)$ measures the current system state and then manipulates the control input u as needed to drive the system toward its target. The initial set of the control system is $(x_1, x_2) \in [-0.51, -0.49] \times [0.49, 0.51]$. As depicted in Fig. 1b, the controlled system is expected to reach the target region in blue while avoiding the obstacle region in red within 120 timesteps (i.e. control steps). In our experience, even for this simple example, using Verisig [28] and ReachNN* [18] (two state-of-the-art verification tools for neural network controlled systems) to calculate the reachable set of a simple 2-layer neural network feedback controller $\pi_{NN}(x_1, x_2)$ costs more than 100s each. It is even more a costly process to repeatedly conduct reachability analysis of a complex neural network controller in a counterexample-guided learning loop.

Recently, programmatic controllers emerge as a promising solution to address the lack of interpretability problem in deep reinforcement learning [47,27,44,38] by training controllers as *programs*. A programmatic controller to control the oscillator environment learned by a programmatic reinforcement learning algorithm [38] is depicted in Fig. 1a. We depict the decision boundary of the program's conditional statement $(28.33x_1 + 4.23x_2 + 4.16 = 0)$ in solid dash in Fig. 1b. The program can be interpreted as a decomposition of the reach-avoid learning problem into two sub-problems — the linear controller in the *else* branch of the program first pushes the system away from the obstacle and next the linear controller in the *then* branch takes over to make the system reach the

target. As we show in this paper, the compact and structured representation of a programmatic controller lends itself amenable to off-the-shelf hybrid or continuous system reachability tools e.g. [10,20]. Compared with verifying a deep neural network controller, reasoning about a programmatic controller is more feasible. However, the question remains when verification fails – rather than retraining a new controller, how can we leverage verification feedback to construct a verifiably correct controller?

Proof Space Optimization. The other main challenge of verification-guided controller synthesis is that when verification fails, the counterexample path may provide little help or even be spurious due to estimated approximation errors. This is because reachability analyses typically overapproximate the true reachable sets using a computationally convenient representation such as polytopes [20] or Taylor models [10]. This overapproximation leads to quick error accumulation over time, known as the wrapping effect. Even a well-trained controller may fail verification because of approximation errors. For example, we adapted a state-of-the-art reachability analyzer Flow* [10] to conduct reachability analysis of the closed-loop system combined by the programmatic controller in Fig. 1a and the oscillator environment (Equation 1) to compute a reachable state set between each time interval within the episode horizon (the controller is applied to generate a control action at the start of each time interval). The result is depicted in Fig. 1b. Although the programmatic controller empirically succeeds reaching the goal on extensive test simulations, the reachability analysis cannot determine whether the target region can always be reached as it computes a larger reachable region that keeps expansion, which may be an overestimation caused by over-approximation.

We hypothesize that verification failures can be caused by (1) true counterexample of unsafe states, (2) states caused by approximate errors, and (3) states in between the time interval of each control step (RL algorithms only sample states at the start and the end of a time interval). The latter two kinds of states cannot be observed by an RL algorithm during training in the concrete system state space. Thus, counterexample-guided controller synthesis may not work well if counterexamples are in the form of paths within the concrete state space.

To address this challenge, we propose synthesizing controllers in the proof space of a reachability analyzer. Controller synthesis in the proof space is critical to learning a verified controller because it can leverage verification feedback on either true unsafe counterexample states or approximation errors introduced by the verification procedure for searching a provably correct controller. A counterexample detected by a reachability analyzer is a *symbolic rollout* of abstract states of the closed-loop system that combines a (fixed) environment model and a (parameterized) programmatic controller. An abstract state (e.g. depicted as a green region in Fig. 1b) at a timestep over-approximates the set of concrete states reachable during the time interval of the timestep. VEL quantifies the safety and reachability property violation by the abstract states, e.g. there is an *abstract* loss between the approximative abstract state and the target region at the last control step. The loss approximates the *worst-case* reachability loss

of any concrete state subsumed by the abstraction. We introduce lightweight gradient-descent style optimization algorithms to optimize controller parameters to effectively minimize the amount of correctness property violation to zero to refute any verification counterexamples.

Contributions. The main contribution of this paper is twofold. First, we present an efficient controller synthesis approach that integrates formal verification within a programmatic controller learning loop. Second, instead of synthesizing a programmatic controller from *concrete* state and action samples, we optimize the controller using *symbolic rollouts* with abstract states obtained by reachability analysis in the verification proof space. We implement the proposed ideas in a tool called VEL and present a detailed experimental study over a range of reinforcement learning systems. Our experiments demonstrate the benefits of integrating formal verification as part of the training objective and using verification feedback for controller synthesis.

2 Problem Setup

Environment Models. An environment is a structure $M^{\delta}[\cdot] = (S, A, F : \{S \times A \to S\}, R : \{S \times A \to \mathbb{R}\}, \cdot)$ where S is an infinite set of *continuous real-vector* environment states which are valuations of the state variables $x_1, x_2, \ldots, x_n$ of dimension n ($S \subseteq \mathbb{R}^n$); and A is a set of *continuous real-vector* control actions which are valuations of the action variables $u_1, u_2, \ldots, u_m$ of dimension m. F is a state transition function that emits the next environment state given a current state s and an agent action a. We assume that F is defined by an ordinary differential equation (ODE) in the form of $\dot{x} = f(x, u)$ and the function $f : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m$ is Lipschitz continuous in x and continuous in u. $R(s, a)$ is the immediate reward after transition from an environment state $s \in S$ with action $a \in A$. An environment $M^{\delta}[\cdot]$ is parameterized with an (unknown) controller.

Controllers. An agent uses a controller to interact with an environment $M^{\delta}[\cdot]$. We explicitly model the deployment of a (learned) controller $\pi : \{S \to A\}$ in $M^{\delta}[\cdot]$ as a *closed-loop* system $M^{\delta}[\pi]$. The controller π determines which action the agent ought to take in a given environment state. Specifically, it is invoked every δ time period at a timestep. π reads the environment state $s_i = s(i\delta)$ at time $t = i\delta$ ($i = 0, 1, 2, \ldots$) or timestep i, and computes a control action as $a_i = a(i\delta) = \pi(s(i\delta))$. Then the environment evolves following the ODE $\dot{x} = f(x, a(i\delta))$ within the time period $[i\delta, (i+1)\delta]$ and obtain the state $s_{i+1} = s((i+1)\delta)$ at the next timestep $i+1$. In the oscillator example from Sec. 1, the duration δ of a timestep is $0.05s$ and the time horizon is $6s$ (i.e. 120 timesteps).

For environment simulation, given a set of initial states S_0, we assume the existence of a flow function[1] $\phi(s_0, t) : S_0 \times \mathbb{R}^+ \to S$ that maps some initial state s_0 to the environment state $\phi(s_0, t)$ at time t where $\phi(s_0, 0) = s_0$. We note that ϕ is the solution of the ODE $\dot{x} = f(x, a(i\delta))$ in the state transition function F during the time period $[i\delta, (i+1)\delta]$ and $a(i\delta) = \pi(\phi(s_0, i\delta))$.

[1] ϕ may be implemented using scipy.integrate.odeint (or scipy.integrate.solve_ivp).

Reinforcement Learning (RL). Given a set of initial states S_0 and a time horizon $T\delta$ ($T > 0$) with δ as the duration of a timestep, a T-timestep rollout ζ of a controller π is denoted as $(\zeta = s_0, a_0, s_1, \ldots, s_T) \sim \pi$ where $s_i = s(i\delta)$ and $a_i = a(i\delta)$ are the environment state and the action taken at timestep i such that $s_0 \in S_0$, $s_{i+1} = F(s_i, a_i)$, and $a_i = \pi(s_i)$. The aggregate reward of π is

$$J^R(\pi) = \mathbb{E}_{(\zeta=s_0,a_0,\ldots,s_T)\sim\pi}[\sum_{t=0}^{T} \beta^t R(s_i, a_i)] \tag{2}$$

where β is the reward discount factor ($0 < \beta \leq 1$). Controller search via RL aims to produce a controller π that maximizes $J^R(\pi)$.

Controller Correctness Specification. A correctness specification of a controller is a logical formula specifying whether any rollout ζ of the controller accomplishes the task without violating safety properties and reachability properties. To define safety and reachability over rollouts, the user first specifies a set of atomic predicates over environment states s.

Definition 1 (Predicates). *A predicate φ is a quantifier-free Boolean combinations of linear inequalities over the environment state variables x:*

$\langle\varphi\rangle ::= \langle P\rangle \mid \varphi \wedge \varphi \mid \varphi \vee \varphi;$

$\langle P\rangle ::= \mathcal{A} \cdot x \leq b$ where $\mathcal{A} \in \mathbb{R}^{|x|}$, $b \in \mathbb{R}$;

A state $s \in S$ satisfies a preciate φ, denoted as $s \models \varphi$, iff $\varphi(s)$ is true.

The correctness requirement of a controller goes beyond from predicates over environment states s to specifications over controller rollouts ζ.

Definition 2 (Rollout Specifications). *The syntax of our correctness specifications for RL controllers is defined as:*

$$\psi ::= \varphi_I \texttt{ reach } \varphi_1 \texttt{ ensuring } \varphi_2$$

In a rollout specification, φ_I `reach` φ_1 enforces reachability - the controlled agent should eventually reach some goal states evaluated true by the predicate φ_1 from an initial state that satisfies φ_I. For instance, the agent should achieve some goals from an initial state. The constraint `ensuring` φ_2 additionally enforces safety - any rollout of the controller should only visit safe states evaluated true by the predicate φ_2. For example, the agent should remain within a safety boundary or avoid any obstacles throughout a rollout. Formally, the semantics of a rollout specification ψ is defined as follows:

$$[\![\varphi_I \texttt{ reach } \varphi_1 \texttt{ ensuring } \varphi_2]\!](\zeta_{0:T}) \;=\; \varphi_1(s_T) \;\wedge\; (\forall\ 0 \leq i \leq T.\ \varphi_2(s_i))$$

where $\zeta_{0:T} = s_0, s_1, \ldots, s_T$ is a rollout such that $s_0 \in \varphi_I$ and $T > 0$ denotes the total number of timesteps. Our specification implicitly requires that if the target region is reached before the T timestep of a rollout, the controlled agent does not leave the target region at the end of the rollout.

Given a time horizon $T\delta$ ($T > 0$), a controller π is correct for an environment $M^\delta[\cdot]$ with respect to a rollout specification $\psi ::= \varphi_\mathtt{I}$ `reach` φ_1 `ensuring` φ_2 iff for any rollout $\zeta_{0:T} = s_0, s_1, \ldots s_{T-1}, s_T$ of $M^\delta[\pi]$ such that $\varphi_\mathtt{I}(s_0)$ holds, $[\![\psi]\!](\zeta_{0:T})$ is true. Notice that this definition does not consider any states of the continuous environment occurring within the time period of a timestep.

Example 1 *Continue the oscillator example. Assume an oscillator initial state is from* $x_1, x_2 \in [-0.51, -0.49] \times [0.49, 0.51]$*. Specify the initial state constraint:*

$$\varphi_I(x_1, x_2) \equiv -0.51 \leq x_1 \leq -0.49 \wedge 0.49 \leq x_2 \leq 0.51$$

The unsafe set of oscillator is $(-03, -0.25) \times (0.2, 0.35)$ *(depicted as the red region in Fig. 1b). The safety* φ_{safe} *of the system is specified as:*

$$\varphi_{safe}(x_1, x_2) \equiv x_1 \leq -0.3 \vee x_1 \geq -0.25 \vee x_2 \leq 0.2 \vee x_2 \geq 0.35$$

For this example, the target region is $[-0.05, 0.05] \times [-0.05, 0.05]$ *(the blue region in Fig. 1b). The reachability of the system* φ_{reach} *is specified as:*

$$\varphi_{reach}(x_1, x_2) \equiv -0.05 \leq x_1 \leq 0.05 \wedge -0.05 \leq x_2 \leq 0.05$$

The target region should be eventually reached by the end of a control episode while avoiding the unsafe state region. We express the rollout specification as:

$$\varphi_I(x_1, x_2) \ \texttt{reach} \ \varphi_{reach}(x_1, x_2) \ \texttt{ensuring} \ \varphi_{safe}(x_1, x_2)$$

The following specification formulates that a desired controller stabilizes the oscillator around the target region over an infinite time horizon:

$$\varphi_{reach}(x_1, x_2) \ \texttt{reach} \ \varphi_{reach}(x_1, x_2) \ \texttt{ensuring} \ \varphi_{safe}(x_1, x_2)$$

3 Programmatic Controllers

Programmatic controllers have emerged as a promising solution to address the lack of interpretability in deep reinforcement learning [47,38,27,8] by learning controllers as programs. This paper focuses on programmatic controllers structured as *differentiable* programs [38].

Our programmatic controllers follow the high-level context-free grammar depicted in Fig. 2 where E is the start symbol, θ represents real-valued parameters of the program. The nonterminals E and B stand for program expressions that evaluate to action values in $\mathbb{R}^m$ and Booleans, respectively, where m is the action dimension size, $\theta_1 \in \mathbb{R}$ and $\theta_2 \in \mathbb{R}^n$. We represent a state input to a programmatic controller as $s = \{x_1 : \nu_1, x_2 : \nu_2, \ldots, x_n\}$ where n is the state dimension size and $\nu_i = s[x_i]$ is the value of x_i in s. As usual, the unbounded variables in $\mathcal{X} = [x_1, x_2, \ldots, x_n]$ are assumed to be input variables (i.e., state variables). C is a low-level affine controller that can be invoked by a programmatic controller where $\theta_3, \theta_c \in \mathbb{R}^m, \theta_4 \in \mathbb{R}^{m \cdot n}$ are controller parameters. Notice that C can be as simple as some (learned) constants θ_c.

$$
\begin{aligned}
E &::= C \mid \textbf{if } B \textbf{ then } C \textbf{ else } E \\
B &::= \theta_1 + \theta_2^T \cdot \mathcal{X} \geq 0 \\
C &::= \theta_3 + \theta_4 \cdot \mathcal{X} \mid \theta_c
\end{aligned}
$$

Fig. 2: A context-free grammar for programmatic controllers.

The semantics of a programmatic controller in E is mostly standard and given by a function $[\![E]\!](s)$, defined for each language construct. For example, $[\![x_i]\!](s) = s[x_i]$ reads the value of a variable x_i in a state s. A controller may use an **if-then-else** branching construct. To avoid discontinuities for differentiability, we interpret its semantics in terms of a smooth approximation:

$$
[\![\textbf{if } B \textbf{ then } C \textbf{ else } E]\!](s) = \sigma([\![B]\!](s)) \cdot [\![C]\!](s) + (1 - \sigma([\![B]\!](s))) \cdot [\![E]\!](s) \quad (3)
$$

where σ is the sigmoid function. Thus, any controller programmed in this grammar is a differentiable program. During execution, a programmatic controller invokes a set of low-level affine controllers under different environment conditions, according to the activation of the B conditions in the program.

Programmatic Reinforcement Learning. We use the programmatic reinforcement learning algorithm [38] to learn a programmatic controller. Compared with other programmatic reinforcement learning approaches [27,47], this algorithm stands out by jointly learning both program structures and program parameters. Empirical results show that learned programmatic controllers achieve comparable or even better reward performance than deep neural networks [38].

4 Proof Space Optimization

The main challenge of using a verification procedure to guide controller synthesis is that verifiers are in general incomplete. When verification fails, it does not necessarily mean the system under verification has a true counterexample as the verifier may introduce states caused by over-approximation errors, commonly seen in reachability analysis. Even a well-trained controller may fail verification because of approximation errors. In our context, for soundness, reachability analysis of continuous or hybrid systems additionally takes environment states in between the time interval of a timestep into account. Both of these kinds of states cannot be observed by RL agents during training in the concrete state space, which renders the importance of controller optimization in the proof space of verification. In the following, Sec. 4.1 defines a verification procedure for environment models governed by programmatic controllers. Sec. 4.2 encodes verification feedback as a loss function of controller parameters over the verification proof space. Finally, Sec. 4.3 defines an optimization procedure that iteratively minimizes the loss function for correct-by-construction controller synthesis.

4.1 Controller Verification

We formalize controller synthesis as a verification-based controller optimization problem. A synthesized controller π is certified by a formal verifier against an environment model $M^\delta[\cdot]$ and a rollout specification ψ (Definition 2). The verifier returns true if π can be verified correct.

Reinforcement learning algorithms typically discretize a continuous environment model $M^\delta[\cdot]$ to sample environment states every δ time period (as a timestep) for controller learning (Sec. 2). For soundness, in verification our approach instead considers all states reachable by the original continuous system. Formally, given a set of initial states S_0, we use S_i $(i > 0)$ to represent the set of reachable concrete states during the time interval of $[(i-1)\delta,\ i\delta]$:

$$S_i = \{\phi(s_0, t) \mid \forall s_0 \in S_0, \forall t \in [(i-1)\delta,\ i\delta]\}$$

where ϕ is the flow function for environment state transition defined in Sec. 2. Our algorithm uses abstract interpretation to soundly approximate the set of reachable states S_i at each time step by reachability analysis.

Definition 3 (Symbolic Rollouts). *Given an environment model $M^\delta[\pi] = (S, A, F, R, \pi)$ deployed with a controller π, a set of initial states S_0, and an abstract domain $\mathcal{D}$, a symbolic rollout of $M^\delta[\pi]$ over $\mathcal{D}$ is $\zeta^{\mathcal{D}} = S_0^{\mathcal{D}}, S_1^{\mathcal{D}}, \ldots$ where $S_0^{\mathcal{D}} = \alpha(S_0)$ is the abstraction of the initial states S_0 in $\mathcal{D}$. Each symbolic state $S_i^{\mathcal{D}} = F^{\mathcal{D}}[\pi]\big(S_{i-1}^{\mathcal{D}}\big)$ over-approximates S_i - the set of reachable states from the initial state S_0 during the time interval $[(i-1)\delta, i\delta]$ of the timestep i. $F^{\mathcal{D}}$ is an abstract transformer for $M^\delta[\pi]$'s state transition function F.*

Our implementation of the abstract interpreter $F^{\mathcal{D}}$ is based on Flow* [10], a reachability analyzer for continuous or hybrid systems, where the abstract domain $\mathcal{D}$ is Taylor Model (TM) flowpipes. Formally, for reachability computation at each timestep i (where $i > 0$), we firstly use Flow* to evaluate the TM flowpipe $\hat{S}_{i-1}$ for the reachable set of states at time $t = (i-1)\delta$. To obtain a TM representation for the output set of the programmatic controller at timestep i, we use TM arithmetic to evaluate a TM flowpipe $\hat{A}_{i-1}$ for $[\![\pi]\!](s)$ for all states $s \in \hat{S}_{i-1}$. Here $[\![\pi]\!]$ encodes the semantics of π (Equation 3). For example, the semantics of the oscillator controller in Fig. 1a is:

$$\begin{aligned}&\sigma(28.33x_1 + 4.23x_2 + 4.16) \times (6.79x_1 - 8.56x_2 + 0.35)\\ &+ (1 - \sigma(28.33x_1 + 4.23x_2 + 4.16)) \times (11.01x_1 - 13.50x_2 + 8.71)\end{aligned}$$

where the sigmoid function σ can be handled by TM arithmetic. The resulting TM representation $\hat{A}_{i-1}$ can be viewed as an overapproximation of the controller's output at timestep i. Finally, we use Flow* to construct the TM flowpipe overapproximation $S_i^{\mathcal{D}}$ for all reachable states during the time period at timestep i by reachability analysis over the ODE dynamics of the transition function $\dot{x} = f(x, a)$ for δ time period with initial state $x(0) \in \hat{S}_{i-1}$ and the control action $a \in \hat{A}_{i-1}$.

Verification Procedure. Given a closed-loop system $M^\delta[\pi]$, a time horizon $T\delta$ ($T > 0$), and a rollout specification $\psi ::= [\![\varphi_I \texttt{ reach } \varphi_1 \texttt{ ensuring } \varphi_2]\!]$, we obtain the symbolic rollout of $M^\delta[\pi]$ as $\zeta^{\mathcal{D}}_{0:T} = S^{\mathcal{D}}_0, S^{\mathcal{D}}_1, \ldots, S^{\mathcal{D}}_T$ where $S^{\mathcal{D}}_0$ is the abstraction of all states in φ_I in the abstract domain $\mathcal{D}$. For formal verification, we extend the semantics definition of the rollout specification $[\![\psi]\!]$ over concrete rollouts (Definition 2) to support symbolic rollouts. Formally, $[\![\psi]\!](\zeta^{D}_{0:T})$ holds iff:

$$\forall s \in \gamma(S^{\mathcal{D}}_T).\ \varphi_1(s) \ \bigwedge\ \forall\ 0 \leq i \leq T,\ s \in \gamma(S^{\mathcal{D}}_i).\ \varphi_2(s)$$

where γ is the concretization function of the abstract domain $\mathcal{D}$. The closed-loop system $M^\delta[\pi]$ satisfies ψ, denoted as $M^\delta[\pi] \models \psi$, iff $[\![\psi]\!](\zeta^{\mathcal{D}}_{0:T})$ holds. The abstract domain $\mathcal{D}$ is the proof space of controller verification.

Example 2 *To verify the closed-loop system composed by the oscillator ODE in Eq. 1 and the learned controller in Fig. 1a, we have conducted reachability analysis to overapproximate the reachable state set during the time period of each timestep within the episode horizon. The result of the TM flowpipes are depicted as a sequence of green regions in Fig. 1b. The verification procedure cannot guarantee that the target be reached eventually due to the approximation errors.*

4.2 Correctness Property Loss in the Proof Space

To facilitate controller optimization in the presence of verification failures, our approach measures the amount of correctness property violation as verification feedback. To this end, we firstly define correct property violation over the concrete environment state space and then lift this definition to the proof space of controller verification.

We note that a controller rollout that fails correctness property verification violates desired properties at some states. The following definition characterizes a correctness loss function to quantify the correctness property violation of a state.

Definition 4 (State Correctness Loss Function). *For a predicate φ over states $s \in S$, we define a non-negative loss function $\mathcal{L}(s, \varphi)$ such that $\mathcal{L}(s, \varphi) = 0$ iff s satisfies φ, i.e. $s \models \varphi$. We define $\mathcal{L}(s, \varphi)$ recursively, based on the possible shapes of φ (Definition 1):*

- $\mathcal{L}(s, \mathcal{A} \cdot x \leq b) := \max(\mathcal{A} \cdot s - b, 0)$
- $\mathcal{L}(s, \varphi_1 \wedge \varphi_2) := \max(\mathcal{L}(s, \varphi_1), \mathcal{L}(s, \varphi_2))$
- $\mathcal{L}(s, \varphi_1 \vee \varphi_2) := \min(\mathcal{L}(s, \varphi_1), \mathcal{L}(s, \varphi_2))$

Notice that $\mathcal{L}(s, \varphi_1 \wedge \varphi_2) = 0$ iff $\mathcal{L}(s, \varphi_1) = 0$ and $\mathcal{L}(s, \varphi_2) = 0$, and similarly $\mathcal{L}(\varphi_1 \vee \varphi_2) = 0$ iff $\mathcal{L}(\varphi_1) = 0$ or $L(\varphi_2) = 0$.

Our objective is to use verification feedback to improve controller safety. To this end, we lift the correctness loss function over concrete states (Definition 4) to an *abstract correctness loss* function over abstract states.

Definition 5 (Abstract State Correctness Loss Function). *Given an abstract state $S^{\mathcal{D}}$ and a predicate φ, we define an abstract correctness loss function:*

$$\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi) = \max_{s \in \gamma(S^{\mathcal{D}})} \mathcal{L}(s, \varphi)$$

where γ is the concretization function of the abstract domain $\mathcal{D}$. The abstract correctness loss function applies γ to obtain all concrete states represented by an abstract state $S^{\mathcal{D}}$. It measures the worst-case correctness loss of φ among all concrete states subsumed by $S^{\mathcal{D}}$. Given an abstract domain $\mathcal{D}$, we can usually approximate the concretization of an abstract state $\gamma(S^{\mathcal{D}})$ with a tight interval $\gamma_I(S^{\mathcal{D}})$. As exemplified in Fig. 1b, it is straightforward to represent Taylor model flowpipes as intervals in Flow. Based on the possible shape of φ, we redefine $\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi)$ as:*

- $\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \mathcal{A} \cdot x \leq b) := \max_{s \in \gamma_I(S^{\mathcal{D}})} \big(\max(\mathcal{A} \cdot s - b, 0)\big)$
- $\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_1 \wedge \varphi_2) := \max(\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_1), \mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_2))$
- $\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_1 \vee \varphi_2) := \min(\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_1), \mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi_2))$

Theorem 1 (Abstract State Correctness Loss Function Soundness). *Given an abstract state $S^{\mathcal{D}}$ and a predicate φ, we have:*

$$\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}, \varphi) = 0 \implies \forall s \in \gamma_I(S^{\mathcal{D}})\ s \models \varphi.$$

We further lift the definition of the correctness loss function over abstract states (Definition 5) to a correctness loss function over symbolic rollouts.

Definition 6 (Symbolic Rollout Correctness Loss). *Given a rollout specification $\psi := \varphi_I$ **reach** φ_1 **ensuring** φ_2 and a symbolic rollout $\zeta^{\mathcal{D}}_{0:T} = S^{\mathcal{D}}_0, \ldots, S^{\mathcal{D}}_T$ where $S^{\mathcal{D}}_0$ is the abstraction of all states in φ_I in the abstract domain $\mathcal{D}$, we define an abstract safety loss function $\mathcal{L}_{\mathcal{D}}(\zeta_{0:T},\ \psi)$ measuring the degree to which the rollout specification is violated:*

$$\mathcal{L}_{\mathcal{D}}(\zeta_{0:T},\ \varphi_I\ \texttt{reach}\ \varphi_1\ \texttt{ensuring}\ \varphi_2) = \max(\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}_T, \varphi_1),\ \max_{0<i\leq T}(\mathcal{L}_{\mathcal{D}}(S^{\mathcal{D}}_i, \varphi_2)))$$

Definition 6 enables a quantitative metric for the correctness loss of a controller in the verification proof space. Given a closed loop system $M^{\delta}[\pi]$, a time horizon $T\delta$, a rollout specification ψ, and the corresponding symbolic rollout $\zeta^{\mathcal{D}}_{0:T}$ of $M^{\delta}[\pi]$, the correctness loss of $M^{\delta}[\pi]$ with respect to ψ, denoted as $\mathcal{L}_{\mathcal{D}}(M^{\delta}[\pi], \psi)$, is defined over the symbolic rollout i.e. $\mathcal{L}_{\mathcal{D}}(M^{\delta}[\pi], \psi) = \mathcal{L}_{\mathcal{D}}(\zeta^{\mathcal{D}}_{0:T}, \psi)$.

Example 3 *In Fig. 1b, there is a correctness loss (depicted as a red arrow) between the abstract state at the last timestep of the oscillator symbolic rollout and the desired reachable region φ_{reach} defined in Example 1. We characterize it as an abstract state correctness loss. The whole symbolic rollout has the same correctness loss with respect to the rollout specification defined in Example 1.*

Theorem 2 (Symbolic Rollout Correctness Soundness). *Given an environment $M^{\delta}[\cdot]$ deployed with a controller π and a rollout specification ψ, we have*

$$\mathcal{L}_{\mathcal{D}}(M^{\delta}[\pi], \psi) = 0 \implies M^{\delta}[\pi] \models \psi.$$

Algorithm 1 VEL: **Ve**rification-based **l**earning framework for controller synthesis. In line 8, ω_k is a Gaussian noise and ν is a small positive real number.

Require: Environment model $M^\delta[\cdot]$, rollout specification ψ, initial controller π_θ trained using the programmatic RL algorithm [38].
Ensure: Optimized controller π_θ such that $M^\delta[\pi_\theta] \models \psi$.

1: **procedure** VEL
2: $\theta \leftarrow$ all parameters in π_θ for optimization
3: **while** `true` **do**
4: $\ell_\mathcal{D} \leftarrow \mathcal{L}_\mathcal{D}(M^\delta[\pi_\theta], \psi)$
5: **if** $\ell_\mathcal{D} = 0$ **then**
6: Dump π_θ to a verified controller list
7: **end if**
8: $\nabla_\theta \mathcal{L}_\mathcal{D} \leftarrow \frac{1}{N}\sum_{k=1}^{N} \frac{\mathcal{L}_\mathcal{D}(M^\delta[\pi_{\theta+\nu\omega_k}],\ \psi) - \mathcal{L}_\mathcal{D}(M^\delta[\pi_{\theta-\nu\omega_k}], \psi)}{\nu}\omega_k$
9: $\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}_\mathcal{D}$ where η is a learning rate
10: **end while**
11: **end procedure**

4.3 Controller Synthesis

The unique feature of our controller synthesis algorithm is that it leverages verification feedback on either true unsafe states or overapproximation errors introduced by verification to search for a provably correct controller.

Controller Synthesis in the Proof Space. We deem a programmatic controller π with trainable parameters θ (e.g. from the grammar in Fig. 2) as π_θ. Given a closed-loop system $M^\delta[\pi_\theta]$, the correctness loss function $\mathcal{L}_\mathcal{D}(M^\delta[\pi_\theta], \psi)$ is essentially a function of π_θ's parameters θ. To reduce the correctness loss of π_θ over the proof space $\mathcal{D}$, we leverage a gradient-descent style optimization to update θ by taking steps proportional to the negative of the gradient of $\mathcal{L}_\mathcal{D}(M^\delta[\pi_\theta], \psi)$ at θ. As opposed to standard gradient descent optimization, we optimize π_θ based on symbolic rollouts in the proof space $\mathcal{D}$, favouring the abstract interpreter (i.e. Flow*) directly for verification-guided controller updates.

Black-box Gradient Estimation. Directly deriving the gradients of $\mathcal{L}_\mathcal{D}$, however, requires the controller verification procedure be differentiable, which is not supported by reachability analyzers such as Flow*. To overcome this challenge, our algorithm effectively estimates the gradients of $\mathcal{L}_\mathcal{D}$ based on random search [34]. Given a closed-loop environment $M^\delta[\pi_\theta]$, at each training iteration, we obtain perturbed systems $M^\delta[\pi_{\theta+\nu\omega}]$ and $M^\delta[\pi_{\theta-\nu\omega}]$ where we add sampled Gaussian noise ω to the current controller π_θ's parameters θ in both directions and ν is a small positive real number. By evaluating the abstract correctness losses of the symbolic rollouts of $M^\delta[\pi_{\theta+\nu\omega}]$ and $M^\delta[\pi_{\theta-\nu\omega}]$, we update θ with a finite difference approximation along an unbiased estimator of the gradient:

$$\nabla_\theta \mathcal{L}_\mathcal{D} \leftarrow \frac{1}{N}\sum_{k=1}^{N} \frac{\left(\mathcal{L}_\mathcal{D}(M^\delta[\pi_{\theta+\nu\omega_k}],\ \psi) - \mathcal{L}_\mathcal{D}(M^\delta[\pi_{\theta-\nu\omega_k}], \psi)\right)}{\nu}\omega_k$$

We update controller parameters θ as follows where η is a learning rate:

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}_\mathcal{D}$$

Our high-level controller synthesis algorithm is depicted in Algorithm. 1. The algorithm takes as input an environment model $M^\delta[\cdot]$, a rollout specification ψ, and a programmatic controller π learned using the programmatic reinforcement learning technique [38]. When verification fails (line 4), it uses the correctness loss of the symbolic rollout of $M^\delta[\pi]$ for optimization (line 8-9). The algorithm repeatedly performs the gradient-based update until a verified controller is synthesized. As the controller verification procedure is undecidable in general, it is possible that Algorithm 1 converges with a nonzero correctness loss. Our empirical results in Sec. 5 demonstrate that the algorithm works well in practice.

5 Experimental Results

We have implemented the verification-guided controller synthesis technique in Algorithm 1 in a tool called VEL (**VE**rification-based **L**earning) [50]. Given an environment and a rollout specification ψ (Definition 2), VEL uses the programmatic reinforcement learning algorithm [38] to learn a programmatic controller π. The controller π is trained to satisfy the safety and reachability requirements as set by ψ. We do so by shaping a reward function that is consistent with ψ - this function rewards actions leading to goal states and penalizes actions leading to unsafe states. As the RL algorithm does not provide any correctness guarantees and the verification procedure may introduce large approximation errors, even well-trained controllers may fail verification. In case of verification failures, VEL applies Algorithm 1 to optimize π based on the verification feedback.

We evaluated VEL on several *nonlinear* continuous or hybrid systems taken from the literature. These are problems that are widely used for evaluating state-of-the-art verification tools for learning-enabled cyber-physical systems. Benchmarks B1 - B5 were introduced by [18]; adaptive cruise control (ACC) was presented in [43]; mountain car (MC) and quadrotor with model-predictive control (QMPC) were introduced by [28]; Pedulum and CartPole were taken from [29]; Tora and Unicyclecar were presented in the ARCH-COMP21 competition on formal verification of Artificial Intelligence and Neural Network Control Systems (AINNCS). We present the dynamics and the detailed description of each benchmark in [50]. The rollout specifications (Definition 2) are depicted in Table 1. The specifications define for each benchmark the initial states, the goal regions to reach, and the safety properties describing the safety boundary or the obstacles to avoid. On three benchmarks we verify the controller correctness over an infinite horizon. For the classic control problem Pendulum, to verify that the pendulum does not fall in an infinite time horizon, the rollout specification requires that any rollout starting from the region $x_1, x_2 \in [-0.1, 0.1]$ (representing pendulum angle and angular velocity) eventually turns back to it and any rollout states must be safe (including those that temporarily leave this region). Similarly, Tora models a moving cart attached to a wall with a spring.

Table 1: Benchmark Rollout Specifications ($\mathcal{T}$ represents True).

Tasks	Rollout Specifications
B_1	$x_1 \in [.8, .9] \wedge x_2 \in [.5, .6]$ `reach` $x_1 \in [0, .2] \wedge x_2 \in [.05, .3]$ `ensuring` $x_1, x_2 \in [-1.5, 1.5]$
B_2	$x_1 \in [.7, .9] \wedge x_2 \in [.7, .9]$ `reach` $x_1 \in [-.3, .1] \wedge x_2 \in [-.35, .5]$ `ensuring` $x_1, x_2 \in [-1.5, 1.5]$
B_3	$x_1 \in [.8, .9] \wedge x_2 \in [.4, .5]$ `reach` $x_1 \in [0, .2] \wedge x_2 \in [.05, .3]$ `ensuring` $\mathcal{T}$
B_4	$x_1, x_3 \in [.25, .27] \wedge x_2 \in [.08, .1]$ `reach` $x_1 \in [-.3, .1] \wedge x_2 \in [-.35, .5]$ `ensuring` $\mathcal{T}$
B_5	$x1 \in [.38, .4] \wedge x_2 \in [.45, .47] \wedge x_3 \in [.25, .27]$ `reach` $x_1 \in [0, .2] \wedge x_2 \in [.05, .3]$ `ensuring` $\mathcal{T}$
Oscillator`inf`	$x_1 \in [-.51, -.49] \wedge x_2 \in [.49, .51]$ `reach` $x_1, x_2 \in [-.05, .05]$ `ensuring` $x_1 \leq -.3 \vee x_1 \geq -.25 \vee x_2 \leq .2 \vee x_2 \geq .35$, $x_1, x_2 \in [-.05, .05]$ `reach` $x_1, x_2 \in [-.05, .05]$ `ensuring` $x_1 \leq -.3 \vee x_1 \geq -.25 \vee x_2 \leq .2 \vee x_2 \geq .35$
ACC	$x_1 \in [90, 110] \wedge x_2 \in [32, 32.05] \wedge x_4 \in [10, 11] \wedge x_5 \in [30, 30.05]$ `reach` $-x_1 + x_4 - 102 \leq 0$ `ensuring` $-x_1 + 1.4 \cdot x_2 + x_4 + 10 \leq 0$
MountainCar	$x_1 \in [-.6, -.4]$ `reach` $x_1 > .45$ `ensuring` $x_1 \leq .15 \vee x_2 \geq .25 \vee x_2 \geq .02$
QMPC	$.025 \leq x_1 \leq .05 \wedge 0 \leq x_2 \leq .025$ `reach` $\mathcal{T}$ `ensuring` $-.32 \leq x_1, x_2, x_3 \leq .32$
Pendulum`inf`	$x_1, x_2 \in [-.1, .1]$ `reach` $x_1, x_2 \in [-.1, .1]$ `ensuring` $x_1, x_2 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$
CartPole	$x_1, x_2, x_3, x_4 \in [-.05, .05]$ `reach` $\mathcal{T}$ `ensuring` $x_1 \in [-2.4, 2.4] \wedge x_2 \in [-.21, .21]$
UnicycleCar	$x_1 \in [9.5, 9.55] \wedge x_2 \in [-4.5, -4.45] \wedge x_3 \in [2.1, 2.11] \wedge x_4 \in [1.5, 1.51]$ `reach` $x_1 \in [-.6, .6] \wedge x_2 \in [-.2, .2] \wedge x_3 \in [-.06, .06] \wedge x_4 \in [-.3, .3]$ `ensuring` $\mathcal{T}$
Tora	$x_1 \in [-.77, -.75] \wedge x_2 \in [-.45, -.43] \wedge x_3 \in [.51, .54] \wedge x_4 \in [-.3, -.28]$ `reach` $x_1 \in [-.1, .2] \wedge x_2 \in [-.9, .6]$ `ensuring` $x_1, x_2, x_3, x_4 \in [-1.5, 1.5]$
Tora`inf`	$x_1, x_2, x_3, x_4 \in [-.1, .1]$ `reach` $x_1, x_2, x_3, x_4 \in [-.1, .1]$ `ensuring` $x_1, x_2, x_3, x_4 \in [-1.5, 1.5]$

On Tora`inf`, we prove that the controller for the arm of the cart connecting to the spring can stabilize the cart over an infinite horizon while maintain safety around the origin. On Oscillator`inf`, we verify that the controller can stabilize the oscillator around a target region over an infinite horizon while the process of reaching the target region from the initial states is safe.

The experimental results are given in Table 2. VEL synthesized provably correct programmatic controllers for all the benchmarks. Table 2 shows the total time spent on each benchmark (T.T) as well as the verification time of the final controller (V.T). Half of the benchmarks can be directly verified with the initial programmatic controller (in Table 2, T.T for these benchmarks is empty as they only need one pass of verification in V.T). The other half must go through the verification-guided controller learning loop due to approximation errors in verification although these controllers achieved satisfactory test performance. We depict the learning performance of VEL on these benchmarks in Fig. 3 averaged over 5 random seeds. The results show that VEL can robustly and reliably reduce the correctness loss over symbolic rollouts (i.e. the verification feedback) to zero.

Table 2: Experiment Results. `Depth` shows the height of the abstract syntax tree of a programmatic controller. `T.T` shows the overall execution time of VEL including both the time for reachability analysis and verification-guided controller synthesis. `V.T` measures only the verification time for the final controller. If a controller can be verified directly without verification-guided optimization, the value of `T.T` is empty. The execution times for ReachNN* and Verisig measure the cost of verifying a neural network controlled system (NNCS). The notation of the size ($n \times k$) indicates a neural network (with sigmoid activations) with n hidden layers and k neurons per layer. If a property could not be verified, it is marked as Unknown. N/A means that the tool is not applicable to a benchmark.

	VEL (ours)			NNCS		
Task	Depth	V.T	T.T	Size	ReachNN*	Verisig
B_1	2	27.32s	86.57s	2×20	69s	49s
B_2	2	0.25s	-	2×20	32s	Unknown
B_3	2	1.96s	-	2×20	130s	47s
B_4	2	0.63s	-	2×20	20s	12s
B_5	2	0.64s	2.01s	3×100	31s	196s
Oscillator$_{\texttt{inf}}$	2	1.74s	25.72s	2×20	Unknown	Unknown
ACC	3	5.56s	196.03s	3×20	Unknown	1512s
MountainCar	3	233.45s	-	2×16	N/A	52s
QMPC	5	2.21s	16.54s	2×20	N/A	697s
Pendulum$_{\texttt{inf}}$	2	0.95s	-	3×64	57s	Unknown
CartPole	3	8.97s	-	2×64	Unknown	Unknown
Unicycle	3	0.75s	16.52s	3×20	N/A	Unknown
Tora	2	3.71s	-	3×20	Unknown	83s
Tora$_{\texttt{inf}}$	2	0.86s	150.86s	3×20	Unknown	Unknown

Table 2 also shows the results of verifying the benchmarks as neural network controlled systems (NNCS) using two state-of-the-art verification tools ReachNN* [18] and Verisig [28] where the controllers are trained as neural networks. We note that VEL is designed for *programmatic* controllers and uniquely has a verification-guided learning loop. Here our intention is not to compare the tools' performance. Instead, Table 2 demonstrates that integrating verification in training loops for programmatic controllers is more tractable than for neural network controllers. It shows that programmatic controller verification (column V.T) has a much lower computation cost compared to verifying neural network controllers using ReachNN* and Verisig except for MountainCar[2]. When ReachNN* and Verisig produces Unknown, the tools are not able to verify the rollout specification due to the large estimated approximation errors in verification. On Tora, ReachNN* spent over 13000s to produce imprecise flowpipes with large approximation errors that cannot be used for verification. In this case, repeatedly conducting neural network controller verification in a learning loop is

[2] MountainCar is a hybrid system model. VEL is not yet optimized for hybrid system verification.

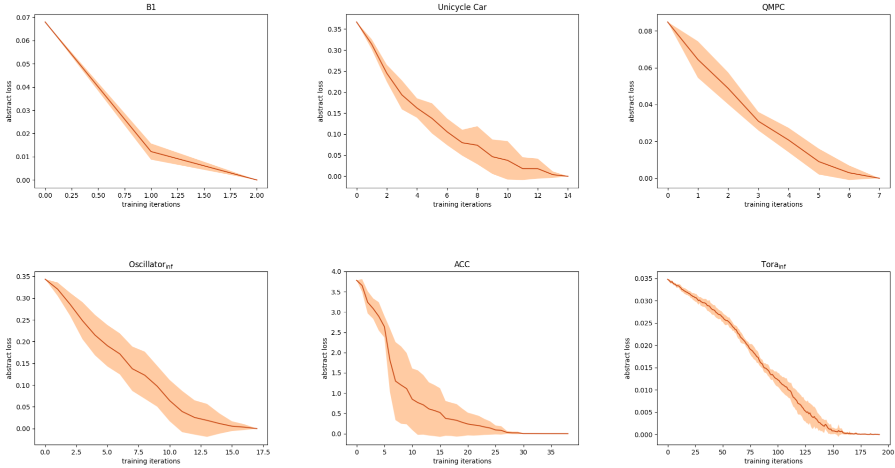


Fig. 3: Learning Performance of Verification-guided Controller Synthesis on B1, UnicycleCar, QMPC, Oscillator, ACC, and $\text{Tora}_{\texttt{inf}}$. The y-axis records the correctness loss of symbolic rollouts over abstract states. The results are averaged over 5 random seeds. VEL reliably reduces the symbolic rollout correctness loss to zero across the learning loop iterations (the x axis) for each benchmark.

computationally infeasible. On the other hand, VEL makes verification-guided controller synthesis feasible as evidenced in Table 2 and Fig. 3. It efficiently uses the programmatic controller verification feedback to reduce the correctness loss over the abstraction of controller reachable states to 0 in the verification proof space (even if the abstraction may introduce approximation errors).

6 Related Work

Robust Machine Learning. Our work on using abstract interpretation [14] for controller synthesis is inspired by the recent advances in verifying neural network robustness, e.g. [23,5,40,51]. These approaches apply abstract interpretation to relax nonlinearity of activation functions in neural networks into convex representations, based on linear approximation [52,51,39,40,55] or interval approximation [26,35]. Since the abstractions are differentiable, neural networks can be optimized toward tighter concertized bounds to improve verified robustness [35,7,55,48,33]. Principally, abstract interpretation can be used to verify the reachability properties of nonlinear dynamics systems [30,37,4]. Recent work [43,28,41,18,17,29,13] has already achieved initial results about verifying neural network controlled autonomous systems by conducting reachability analysis. However, these approaches do not attempt to leverage verification feedback for controller synthesis within a learning loop partially because of the high com-

putation demand of repeatedly verifying neural network controllers. VEL demonstrates the substantial benefits of using verification feedback in a proof space for learning correct-by-construction programmatic controllers. Related works [25,16] conduct trajectory planning from temporal logic specifications but do not provide formal correctness guarantees. Extending VEL to support richer logic specifications is left for future work.

Safe Reinforcement Learning. Safe reinforcement learning is a fundamental problem in machine learning [36,45]. Most safe RL algorithms form a constraint optimization problem by specifying safety constraints as cost functions in addition to reward functions [1,9,15,31,42,54,53]. Their goal is to train a controller that maximizes the accumulated reward and bound the aggregate safety violation under a threshold. However, aggregate safety costs do not support reachability constraints in the Safe RL context. In contrast, VEL ensures that a learned controller be formally verified correct and can better handle reachability constraints beyond safety. Model-based safe learning is combined with formal verification in [22] where an environment model is updated as learning progresses to take into account the deviations between the model and the actual system behavior. We leave combing VEL and model-based learning in future work.

Safe Shielding. The general idea of shielding is to use a backup controller to enforce the safety of a deep neural network controller [3]. The backup controller is less performant than the neural controller but is safe by construction using formal methods. The backup controller runs in tandem with the neural controller. Whenever the neural controller is about to leave the provably safe state space governed by the backup controller, the backup controller overrides the potentially unsafe neural actions to enforce the neural controller to stay within the certified safe space [2,11,21,22,24,56,6,32]. In contrast, VEL directly integrates formal verification into controller learning loops to ensure that learned controllers are correct-by-construction and hence eliminates the need for shielding.

7 Conclusion

We present VEL that bridges formal verification and synthesis for learning correct-by-construction programmatic controllers. VEL integrates formal verification into a controller learning loop to enable counterexample-guided controller optimization. VEL encodes verification feedback as a loss function of the parameters of a programmatic controller over the verification proof space. Its optimization procedure iteratively reduces both controller correctness violation by true counterexamples and overapproximation errors caused by abstraction. Our experiments demonstrate that controller updates based on verification feedback can lead to provably correct programmatic controllers. For future work, we plan to extend VEL to support controller safety during exploration in noisy environments. When a worst-case environment model is provided, this can be achieved by repeatedly leveraging the verification feedback on safety violation to project a controller back onto the verified safe space [12] after each reinforcement learning step taken on the parameter space of the controller.

Data-Availability Statement VEL is available at the repository [50]. The instructions for reproducing our experiment results are included in this repository.

Acknowledgments This work was supported in part by NSF CCF-2007799 and NSF CCF-2124155.

References

1. Achiam, J., Held, D., Tamar, A., Abbeel, P.: Constrained policy optimization. In: Precup, D., Teh, Y.W. (eds.) Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017. Proceedings of Machine Learning Research, vol. 70, pp. 22–31. PMLR (2017)
2. Akametalu, A.K., Kaynama, S., Fisac, J.F., Zeilinger, M.N., Gillula, J.H., Tomlin, C.J.: Reachability-based safe learning with gaussian processes. In: 53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014. pp. 1424–1431. IEEE (2014)
3. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. 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. 2669–2678. AAAI Press (2018)
4. Althoff, M.: An introduction to cora 2015. In: Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)
5. Anderson, G., Pailoor, S., Dillig, I., Chaudhuri, S.: Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 731–744 (2019)
6. Anderson, G., Verma, A., Dillig, I., Chaudhuri, S.: Neurosymbolic reinforcement learning with formally verified exploration. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020)
7. Balunovic, M., Vechev, M.T.: Adversarial training and provable defenses: Bridging the gap. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net (2020)
8. Bastani, O., Pu, Y., Solar-Lezama, A.: Verifiable reinforcement learning via policy extraction. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. pp. 2499–2509 (2018)
9. Berkenkamp, F., Turchetta, M., Schoellig, A.P., Krause, A.: Safe model-based reinforcement learning with stability guarantees. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA. pp. 908–918 (2017)

10. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 258–263. Springer (2013)
11. Chow, Y., Nachum, O., Duéñez-Guzmán, E.A., Ghavamzadeh, M.: A lyapunov-based approach to safe reinforcement learning. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. pp. 8103–8112 (2018)
12. Chow, Y., Nachum, O., Faust, A., Duéñez-Guzmán, E.A., Ghavamzadeh, M.: Safe policy learning for continuous control. In: Kober, J., Ramos, F., Tomlin, C.J. (eds.) 4th Conference on Robot Learning, CoRL 2020, 16-18 November 2020, Virtual Event / Cambridge, MA, USA. Proceedings of Machine Learning Research, vol. 155, pp. 801–821. PMLR (2020)
13. Christakis, M., Eniser, H.F., Hermanns, H., Hoffmann, J., Kothari, Y., Li, J., Navas, J.A., Wüstholz, V.: Automated safety verification of programs invoking neural networks. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12759, pp. 201–224. Springer (2021)
14. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252 (1977)
15. Dalal, G., Dvijotham, K., Vecerík, M., Hester, T., Paduraru, C., Tassa, Y.: Safe exploration in continuous action spaces. CoRR **abs/1801.08757** (2018)
16. Dawson, C., Fan, C.: Robust counterexample-guided optimization for planning from differentiable temporal logic. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2022, Kyoto, Japan, October 23-27, 2022. pp. 7205–7212. IEEE (2022)
17. Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: Ozay, N., Prabhakar, P. (eds.) Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019. pp. 157–168. ACM (2019)
18. Fan, J., Huang, C., Chen, X., Li, W., Zhu, Q.: Reachnn*: A tool for reachability analysis of neural-network controlled systems. In: Hung, D.V., Sokolsky, O. (eds.) Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12302, pp. 537–542. Springer (2020)
19. François-Lavet, V., Henderson, P., Islam, R., Bellemare, M.G., Pineau, J.: An introduction to deep reinforcement learning. Foundations and Trends® in Machine Learning **11**(3-4), 219–354 (2018)
20. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: Spaceex: Scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011.

Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 379–395. Springer (2011)
21. 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)
22. Fulton, N., Platzer, A.: Verifiably safe off-model reinforcement learning. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11427, pp. 413–430. Springer (2019)
23. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA. pp. 3–18 (2018)
24. Gillula, J.H., Tomlin, C.J.: Guaranteed safe online learning via reachability: tracking a ground target using a quadrotor. In: IEEE International Conference on Robotics and Automation, ICRA 2012, 14-18 May, 2012, St. Paul, Minnesota, USA. pp. 2723–2730. IEEE (2012)
25. Gilpin, Y., Kurtz, V., Lin, H.: A smooth robustness measure of signal temporal logic for symbolic control. IEEE Control. Syst. Lett. **5**(1), 241–246 (2021)
26. Gowal, S., Dvijotham, K., Stanforth, R., Bunel, R., Qin, C., Uesato, J., Arandjelovic, R., Mann, T.A., Kohli, P.: On the effectiveness of interval bound propagation for training verifiably robust models. CoRR **abs/1810.12715** (2018)
27. Inala, J.P., Bastani, O., Tavares, Z., Solar-Lezama, A.: Synthesizing programmatic policies that inductively generalize. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net (2020)
28. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Ozay, N., Prabhakar, P. (eds.) Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019. pp. 169–178. ACM (2019)
29. Jin, P., Tian, J., Zhi, D., Wen, X., Zhang, M.: Trainify: A cegar-driven training and verification framework for safe deep reinforcement learning. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 193–218. Springer (2022)
30. Koller, T., Berkenkamp, F., Turchetta, M., Krause, A.: Learning-based model predictive control for safe exploration. In: 57th IEEE Conference on Decision and Control, CDC 2018, Miami, FL, USA, December 17-19, 2018. pp. 6059–6066. IEEE (2018)
31. Le, H.M., Voloshin, C., Yue, Y.: Batch policy learning under constraints. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA. Proceedings of Machine Learning Research, vol. 97, pp. 3703–3712. PMLR (2019)
32. Li, S., Bastani, O.: Robust model predictive shielding for safe reinforcement learning with stochastic dynamics. In: 2020 IEEE International Conference on Robotics

and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020. pp. 7166–7172 (2020)
33. Lin, X., Zhu, H., Samanta, R., Jagannathan, S.: Art: Abstraction refinement-guided training for provably correct neural networks. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 148–157. IEEE (2020)
34. Mania, H., Guy, A., Recht, B.: Simple random search of static linear policies is competitive for reinforcement learning. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. pp. 1805–1814 (2018)
35. Mirman, M., Gehr, T., Vechev, M.T.: Differentiable abstract interpretation for provably robust neural networks. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 3575–3583. PMLR (2018)
36. Moldovan, T.M., Abbeel, P.: Safe exploration in markov decision processes. In: Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012. icml.cc / Omnipress (2012)
37. Oulamara, M., Venet, A.J.: Abstract interpretation with higher-dimensional ellipsoids and conic extrapolation. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 415–430. Springer (2015)
38. Qiu, W., Zhu, H.: Programmatic reinforcement learning without oracles. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net (2022)
39. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.T.: Fast and effective robustness certification. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. pp. 10825–10836 (2018)
40. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. **3**(POPL), 41:1–41:30 (2019)
41. Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. In: Ozay, N., Prabhakar, P. (eds.) Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019. pp. 147–156. ACM (2019)
42. Tessler, C., Mankowitz, D.J., Mannor, S.: Reward constrained policy optimization. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net (2019)
43. Tran, H., Yang, X., Lopez, D.M., Musau, P., Nguyen, L.V., Xiang, W., Bak, S., Johnson, T.T.: NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 3–17. Springer (2020)
44. Trivedi, D., Zhang, J., Sun, S.H., Lim, J.J.: Learning to synthesize programs as interpretable and generalizable policies. In: Beygelzimer, A., Dauphin, Y., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems (2021)

45. Turchetta, M., Berkenkamp, F., Krause, A.: Safe exploration in finite markov decision processes with gaussian processes. In: Lee, D.D., Sugiyama, M., von Luxburg, U., Guyon, I., Garnett, R. (eds.) Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain. pp. 4305–4313 (2016)
46. Uesato, J., Kumar, A., Szepesvári, C., Erez, T., Ruderman, A., Anderson, K., Dvijotham, K.D., Heess, N., Kohli, P.: Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net (2019)
47. Verma, A., Murali, V., Singh, R., Kohli, P., Chaudhuri, S.: Programmatically interpretable reinforcement learning. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 5052–5061. PMLR (2018)
48. Wang, S., Chen, Y., Abdou, A., Jana, S.: Mixtrain: Scalable training of formally robust neural networks. CoRR **abs/1811.02625** (2018)
49. Wang, Y., Huang, C., Wang, Z., Wang, Z., Zhu, Q.: Design-while-verify: correct-by-construction control learning with verification in the loop. In: Oshana, R. (ed.) DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022. pp. 925–930. ACM (2022)
50. Wang, Y., Zhu, H.: VEL: Verification-guided Programmatic Controller Synthesis. https://doi.org/10.5281/zenodo.7574512, `https://github.com/RU-Automated-Reasoning-Group/VEL`
51. Weng, T., Zhang, H., Chen, H., Song, Z., Hsieh, C., Daniel, L., Boning, D.S., Dhillon, I.S.: Towards fast computation of certified robustness for relu networks. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 5273–5282. PMLR (2018)
52. Wong, E., Kolter, J.Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 5283–5292. PMLR (2018)
53. Yang, C., Chaudhuri, S.: Safe neurosymbolic learning with differentiable symbolic execution. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net (2022)
54. Yang, T., Rosca, J., Narasimhan, K., Ramadge, P.J.: Projection-based constrained policy optimization. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net (2020)
55. Zhang, H., Chen, H., Xiao, C., Gowal, S., Stanforth, R., Li, B., Boning, D.S., Hsieh, C.: Towards stable and efficient training of verifiably robust neural networks. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net (2020)
56. Zhu, H., Xiong, Z., Magill, S., Jagannathan, S.: An inductive synthesis framework for verifiable reinforcement learning. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 686–701. ACM (2019)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.