Learnable Model Verification through Reinforcement Learning

Yao Feng
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
yaofeng@andrew.cmu.edu

Anita Li
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
weihanl1@andrew.cmu.edu

Abstract

Formal Verification of Cyber-Physical Systems provides safety justification for the modeling of the environment. However, an accurate model is usually hard to acquire due to the exquisite nature of physics in the real world. With the aid of model-based reinforcement learning, this paper proposes an approach that updates model parameters and provides formal verification via differential dynamic logic (d$\mathcal{L}$). We present the learning process for a series of common reinforcement learning tasks in Cyber-Physical Systems and show that the safety of the model is preserved by proving the parameterized model.

1 Introduction

Reinforcement learning (RL) has achieved impressive results in many fields, such as Atari games [1] and control of robots [2]. However, learning an agent usually needs a huge number of samples [3], which may be costly to acquire. As a result, low sample efficiency impedes the application of RL to the real world.

Because of this issue, model-based reinforcement learning (MBRL) is becoming more and more popular, which typically means learning a model to guide the learning process instead of simply learning by trial and error. In fact, MBRL can enhance sample efficiency by orders of magnitude [4].

However, an important assumption here is that the cost of each sample (episode) is constant. This assumption is usually not true. For example, when our car crashes, the cost incurred can be much higher than the case where our car brakes too early. Thus, incorporating safety considerations into MBRL makes it possible to control costs more efficiently.

The complex environment and agent movement in hybrid systems can be naturally integrated with RL in two directions. One direction is using hybrid systems to enhance the safety of RL, which is already explored by a lot of former work. Recent work in this field includes Justified Speculative Learning (JSL) [5], Verifiably Safe Reinforcement Learning (VSRL) [6]. But these methods usually assume that we already have a perfect model or a group of perfect models, which is too good to be true in practice.

Another direction is using information from RL to refine hybrid system models, which is usually ignored. This missing link can be helpful and important especially when we

Preprint. Under review.

do not have a perfect model initially. Verification Preserving Model Updates(VPMU) [7] already uses an ensemble of many models to make their model more generic, but this can be inefficient when the state and action spaces are continuous and high-dimensional.

Therefore, we propose an algorithm built upon JSL, which introduces updates of model parameters in the process of training RL models. Our approach can be viewed as an extension of VPMU by adding the ability to select better models from infinite, continuous model space. Specifically, we aim to make a correct but inefficient model be correct and efficient through updates.

This paper contributes the following:

- Propose Learnable Justified Speculative Learning Algorithm that combines formal theorem proving with model-based reinforcement learning and achieves learnable model verification.
- Build the environment for three hybrid systems (including 1D and 2D systems) with agents that perform common RL tasks, provide formally verified models using automatic theorem prover KeYmaera X [8] and evaluate the performance of the algorithm. We provide a discussion of the learned rewards using reinforcement learning and updated parameters through our learnable model.

Equal work was performed by both project members.

2 Background

2.1 Differential Dynamic Logic

We introduce differential dynamic logic (d$\mathcal{L}$) [9] which is designed specifically to verify hybrid systems which interact with discrete and continuous dynamics. In d$\mathcal{L}$, we use hybrid programs (HPs) to model the controller and physical movement, which is summarized using the syntax below:

$$\alpha, \beta ::= x := e \mid ?Q \mid x' = f(x) \& Q \mid \alpha \cup \beta \mid \alpha; \beta \mid \alpha^*.$$

The syntax of d$\mathcal{L}$ is defined as

$$\phi := \theta_1 \sim \theta_2 \mid \forall x \phi \mid \exists x \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \phi_1 \leftrightarrow \phi_2,$$

where θ_1, θ_2 are real arithmetic terms and $\sim$ stands for real arithmetic operators.

The semantics of a hybrid program α is defined as

$$[\![\alpha]\!] = \{(s_1, s_2) | s_2 \text{ is reachable from } s_1 \text{ by executing } \alpha\}.$$

Here we only summarize an intuition of the semantics, and the formal definitions can be found in [9]. The semantics of $d\mathcal{L}$ formulas follow naturally from first-order logic, with the addition of $[\alpha]\phi$, which is true if and only if for all runs of the hybrid program α, ϕ is true; and $\langle\alpha\rangle\phi$, which is true if and only if there exists a run of the hybrid program α where ϕ is true.

Table 1: Hybrid Program Semantics

Syntax	Semantics
$x := e$	Assign the value of e to the variable x, leaving all other variables unchanged
$?Q$	Test if Q is true, continue running; else terminate
$x' = f(x) \& Q$	Follow the system of differential equation $x' = f(x)$ for a certain amount of time when Q holds true
$\alpha; \beta$	Non-deterministically run HP α or β
$\alpha; \beta$	Sequentially run β after α
α^*	Run α repeatedly for any ≥ 0 amount of iterations

d$\mathcal{L}$ model correctness can be checked via KeYmaera X [8], an automatic verification tool. It uses Bellerophon tactic language [10] to express the d$\mathcal{L}$ sequent proofs.

A verified model can be expressed as

$$\texttt{init} \rightarrow [\{\texttt{ctrl}; \texttt{plant}\}^*]\texttt{safe}$$

which means under certain initial conditions (init), no matter how many times we do the control part (ctrl) and let the system evolve (plant) alternately, the postconditions (safe) are satisfied.

2.2 Reinforcement Learning

Reinforcement learning focuses on making decisions in an environment as an agent to maximize rewards. Here we only consider the Markov decision process (MDP) model for RL, which is a widely-used model.

An MDP defined on a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$. $\mathcal{S}$ is the state space, which is all possible states of the environment. $\mathcal{A}$ is the action space, which is all the actions that the agent can take. $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{P}(\mathcal{S})$ is the transition function, which describes the transition probabilities between states in any given state and action pair. $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{P}(\mathbb{R})$ is the reward function, which describes the distribution of rewards that the agent can get by doing a specific action in a specific state. $\gamma \in [0, 1]$ is the discount factor, which describes the relative importance of rewards between different time steps.

The goal of RL is to find an optimal policy $\pi : \mathcal{S} \rightarrow \mathbb{P}(\mathcal{A})$ for the agent. Specifically, if the agent follow the policy to take actions, the expected discounted rewards $\mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i R(s_i, a_i)\right]$ will be maximized.

In some cases, the interaction process may finish due to time limits or illegal actions, we can still use this framework with an extra function $done : \mathcal{S} \rightarrow Bool$. The output of this function is true if and only if the input is a terminal state. In order to cover these cases, we defined the MDP on a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma, done \rangle$ in the following analyses.

2.3 Connecting Hybrid Programs to Reinforcement Learning

In this paper, we focus on one specific type of transition function, which can be decomposed into two parts. We define $T_1 : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ and $T_2 : \mathcal{S} \rightarrow \mathbb{P}(\mathcal{S})$, such that $T \equiv T_2 \circ T_1$. T_1 describes the instantaneous and deterministic state transition right after the control decision, while T_2 describes the evolution in the environment.

We can define controller monitors and model monitors to verify the consistence between the transition function of the RL environment and the transition in the repetition part of the HP ($\{ctrl; plant\}$). A controller monitor $CM : \mathcal{S} \times \mathcal{A} \to Bool$ is used to compare T_1 with $ctrl$, and a model monitor $MM : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to Bool$ is used to compare T_2 with $plant$. Formal definitions are given in definition 1 and definition 2.

Definition 1 *(Controller Monitor). Given an MDP* $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma, done\rangle$ *and a verified model* $\texttt{init} \to [\{\texttt{ctrl};\texttt{plant}\}^*]\texttt{safe}$*, if* $CM(s,a)$*=True,* $(s, T_1(s,a)) \in [\![\texttt{ctrl}]\!]$ [5].

Definition 2 *(Model Monitor). Given an MDP* $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma, done\rangle$ *and a verified model* $\texttt{init} \to [\{\texttt{ctrl};\texttt{plant}\}^*]\texttt{safe}$*, if* $MM(s,a,s')$*=True,* $(T_1(s,a), s') \in [\![\texttt{plant}]\!]$ [5].

With such definitions, we can define when an MDP is accurately modeled by a hybrid program, which is in definition 3.

Definition 3 *(Accurate Modeling). Given an MDP* $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma, done\rangle$ *and a verified model* $\texttt{init} \to [\{\texttt{ctrl};\texttt{plant}\}^*]\texttt{safe}$*, if for any* $s \in \mathcal{S}$*,* $a \in \mathcal{A}$*,* $(s, T_1(s,a)) \in [\![\texttt{ctrl}]\!]$ *implies* $(T_1(s,a), s') \in [\![\texttt{plant}]\!]$ *for any* $s' \in \mathcal{S}$ *reachable from state* s *by doing action* a *in the MDP.*

The only difference between our definitions and the definitions in JSL is that we expand JSL to the case that T_2 is non-deterministic. In this way, we can deal with more generic reinforcement learning models, since the transitions of the MDP are not necessarily deterministic. Besides, if T_2 maps each state to a one-point distribution, our definitions will degenerate to cases in JSL.

Based on these observations, we can define the learning process in the non-deterministic case, which is in definition 4. Here the $learning\ algorithm$ can be any RL algorithm for policy π (for example, the SAC algorithm [11]). Based on this process, we know that if the environment model is accurate, any states s in the learning process are safe ($s \models safe$). The proof idea is to identify a loop invariant [5]. The introduction of a non-deterministic transition function will not lead to extra burdens for this proof, since we consider all reachable states for the non-deterministic transitions in definitions.

Definition 4 *(Learning Process with a Fixed Model). A sequence of tuples* (s_t, a_t, π_t) *is a learning process for* $(\texttt{init}, \langle \mathcal{S}, \mathcal{A}, T, R, \gamma, done\rangle, learning\ algorithm, CM, MM)$ *if and only if it satisfies*

$$a_t \sim \pi(s_t)|_{\{a_t \in \mathcal{A} | actionOK(s,a,t)\}} \tag{1a}$$

$$s_{t+1} \sim T_2(T_1(s_t, a_t)) \tag{1b}$$

$$r_{t+1} \sim R(s_t, a_t) \tag{1c}$$

$$\pi_{t+1} = learning\ algorithm(\pi_t, \gamma, \{(s_i, a_i, s_{i+1}, r_i, done(s_i)) | i \geq 0, i \leq t, i \in \mathbb{Z}\}), \tag{1d}$$

where $s_0 \models \texttt{init}$*, and*

$$actionOK(s,a,t) = CM(s_t, a_t) \vee \neg MM(s_{t-1}, a_{t-1}, s_t).$$

3 Related Work

Here we will talk about some work related to safe RL. Safe RL focuses on making the agent stay in a safe space of states while still pursuing high rewards. Junges et al. purpose a method to ensure safety by schedulers [12]. But their method relies on oracles to build schedulers, which highly restricts its application. Trust Region Policy Optimization (TRPO) [13] is more practical, but it only guarantees the monotonic improvements of policy in the learning process, which can not be a good guarantee for safety. Constrained Policy Optimization (CPO) [14] is an extension of TRPO, which guarantees that the agent nearly satisfies some pre-defined constraints.

Several work [5, 6] introduce formal verification of HP models to safe RL. In this way, we can use less prior knowledge, since we have run-time verification instead of guaranteeing safety only by design. Besides, the flexibility of HPs makes it possible to build more generic models. Justified Speculative Learning (JSL) [5] is proposed as the first algorithm that incorporates safety guarantees preserved throughout the learning process, by defining model monitors to evaluate the correctness of models and controllers monitors to choose safe actions when models are correct. As long as the world model is accurate, the trial and error process will not lead to unsafe cases. Verifiably Safe Reinforcement Learning (VSRL) [6] framework extends JSL by building an end-to-end neural network to support visual inputs. These methods can lead to higher accumulative rewards in addition to remaining safe in the learning process. Verification Preserving Model Updates(VPMU) [7] deals with the imperfection of models by using an ensemble of models, which releases the assumption of former methods.

Our method not only uses HPs to make RL safe but also uses RL interactions to make HPs more efficient. The latter part is a missing feedback link previously, and we can make a virtuous cycle between formal verification and RL in this way. It can be seen as an extension of JSL since we introduce a learnable model instead of a fixed perfect model.

Besides, our method can also be viewed as an extension of VPMU. VPMU uses a fixed finite set of models and uses the accurate models to decide which action is safe. Our method can deal with a continuous model space with infinite models. We only use one model in the model space to make a decision, which is more efficient. Besides, we update the model during training, which can be viewed as a shrink of the model space to more efficient parts. In contrast, VPMU only uses a fixed set of models.

4 Methods

4.1 The Learnable Justified Speculative Learning Algorithm

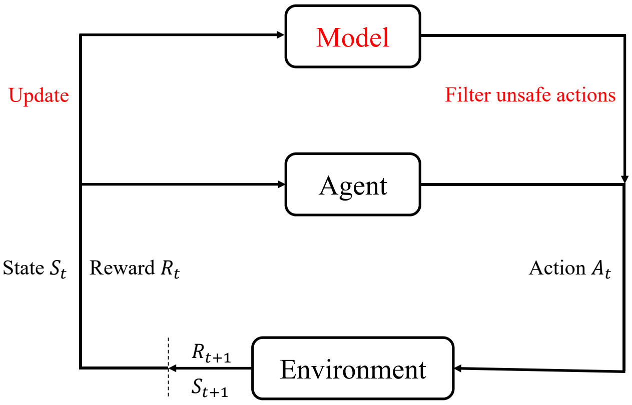


Figure 1: The framework of incorporating a learnable model into reinforcement learning

Our method is called Learnable Justified Speculative Learning (LJSL), which is an extension of JSL by using learnable a world model. In the original version of JSL, we need an accurate model of the environment, which is usually hard to acquire. Instead of having a complete oracle, we can start with a conservative model where safety conditions are guaranteed. After that, we modify our world model in the training process of RL, since we are getting knowledge from interactions with the environment. This process is shown in Figure 1.

On the basis of JSL, we need to define the learning process of our model. For parameters $\theta \in \Theta$, We denote our model as $\mathtt{init}_\theta \rightarrow [\{\mathtt{ctrl}_\theta; \mathtt{plant}\}^*]\mathtt{safe}$, where $CM_\theta : \Theta \times \mathcal{S} \times \mathcal{A} \rightarrow Bool$ is the controller monitor, and $MM : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow Bool$ is the model monitor. The learning process can be defined as $H : \Theta \times \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \Theta$. In other words, we update our model using $\theta_{t+1} = H(s_t, a_t, s_{t+1}, \theta_t)$ in time step t, where s_t, a_t and s_{t+1} are from the transition part of RL. The whole learning process is defined in definition 5. We do not parameterize $plant$ and $safe$, since the physical rules and our goals are usually fixed.

Definition 5 *(Interactive Learning Process). A sequence of tuples* $(s_t, a_t, \pi_t, \theta_t)$ *is a learning process for* $(\mathtt{init}_\theta, \langle \mathcal{S}, \mathcal{A}, T, R, \gamma, done \rangle, learning\ algorithm, CM_\theta, MM)$ *if and only if it satisfies*

$$a_t \sim \pi_t(s_t)|_{\{a_t \in \mathcal{A} | actionOK(s,a,\theta,t)\}} \tag{2a}$$

$$s_{t+1} \sim T_2(T_1(s_t, a_t)) \tag{2b}$$

$$r_{t+1} \sim R(s_t, a_t) \tag{2c}$$

$$\pi_{t+1} = learning\ algorithm(\pi_t, \gamma, \{(s_i, a_i, s_{i+1}, r_i, done(s_i)) | i \geq 0, i \leq t, i \in \mathbb{Z}\}) \tag{2d}$$

$$\theta_{t+1} = H(s_t, a_t, s_{t+1}, \theta_t), \tag{2e}$$

where $s_0 \models \mathtt{init}_{\theta_0}$*, and*

$$actionOK(s, a, \theta, t) = CM_\theta(\theta_t, s_t, a_t) \vee \neg MM(s_{t-1}, a_{t-1}, s_t).$$

The Learnable Justified Speculative Learning algorithm is shown in Algorithm 1. The difference between this algorithm and the JSL algorithm is that our algorithm adds the learning of the control monitor as well as the initialization to learn a more efficient hybrid program.

If the environment is accurately modeled by the system $\{\mathtt{ctrl}_\theta; \mathtt{plant}\}^*$ for any θ during the learning process, the algorithm constructs an interactive learning process. So, we will discuss how to update θ so that the model is always accurate, as well as the safety of the interactive learning process in the next subsection.

4.2 Theoretical Analyses

We already know that if the environment is correctly modeled by a verified model, the learning process is safe. To guarantee safety when the model is changing, we need to give some constraints to H. Here is the formal definition.

Definition 6 *(Valid Update Functions).* $H : \Theta \times \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \Theta$ *is a valid update function if and only if for any accurate model* $\mathtt{init}_\theta \rightarrow [\{\mathtt{ctrl}_\theta;\mathtt{plant}\}^*]\mathtt{safe}$ *of an MDP* $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma, done \rangle$*, learning algorithm,* $\mathtt{init}_{H(\theta,s,a,s')} \rightarrow [\{\mathtt{ctrl}_{H(\theta,s,a,s')};\mathtt{plant}\}^*]\mathtt{safe}$ *is still an accurate model for the MDP for any non-terminal state* $s \in \mathcal{S}$*, action* $a \in \mathcal{A}$*, and* $s' \sim T(s, a)$*.*

If we already have a valid update function, that is not enough to guarantee safety. When we switch to a new model, $\mathtt{init}_\theta$ does not necessarily hold. A simple case is that $\mathtt{init}_\theta$ does

Algorithm 1 Learnable Justified Speculative Learning

Input: $\texttt{init}_\theta$, $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma, done \rangle$, $learning\ algorithm$, CM_θ, MM, θ_0, π_0
Output: π, θ

```
  Initialize $s_0$ such that $s_0 \models \texttt{init}_{\theta_0}$
  $\texttt{history} = \{\}, t = 0$
  while $!done(s_t)$ do
    if $MM(s, a, s)\ \forall(s, a, s', r, d) \in history$ then
      Sample $a_t$ from $\pi_t(s_t)|_{\{a_t \in \mathcal{A} | actionOK(s,a,\theta,t)\}}$
    else
      Sample $a_t$ from $\pi_t(s_t)$
    end if
    $s_{t+1} \sim T(s_t)$
    $r_{t+1} \sim R(s_t, a_t)$
    $\pi_{t+1} = learning\ algorithm(\pi_t, \gamma, \{(s_i, a_i, s_{i+1}, r_i, done(s_i)) | 0 \leq i \leq t, i \in \mathbb{Z}\})$
    $\theta_{t+1} = H(s_t, a_t, s_{t+1}, \theta_t)$
    $history = history \cup \{(s_t, a_t, s_{t+1}, r_t, done(s_t))\}$
    $t = t + 1$
  end while
  $\pi = \pi_t$
  $\theta = \theta_t$
```

not depend on θ, and $\texttt{init}_\theta$ is invariant between loops. In this case, we can use the former model to guarantee safety in that step, and also initialize the needed conditions for the next model. In this way, we can successfully connect the safety proofs of sequential models. A more generic and formal version is shown in Theorem 1.

Theorem 1 *(Safety of LJSL). We assume that for any* $\theta \in \Theta$*,* $\texttt{init}_\theta \rightarrow [\{\texttt{ctrl}_\theta;\texttt{plant}\}^*](\texttt{init}_\theta \wedge \texttt{safe})$ *is valid, and the sequence of tuples* $(s_t, a_t, \pi_t, \theta_t)$ *is a learning process for* $(\texttt{init}, \langle \mathcal{S}, \mathcal{A}, T, R, \gamma, done \rangle, learning\ algorithm, CM_\theta, MM)$*, where* CM_θ *and* MM *are the controller monitor and model monitor of* $\texttt{init}_\theta \rightarrow [\{\texttt{ctrl}_\theta;\texttt{plant}\}^*]\texttt{safe}$*. We further assume that* H *is a valid update function, and* $\texttt{init}_\theta \rightarrow \texttt{init}_{H(\theta,s,a,s')}$ *holds for any non-terminal state* $s \in \mathcal{S}$*, action* $a \in \mathcal{A}$*, and* $s' \sim T(s,a)$*. Then* $\texttt{init}_{\theta_t} \rightarrow [\{\texttt{ctrl}_{\theta_t};\texttt{plant}\}^*]\texttt{safe}$ *is accurate and* $s_t \models \texttt{safe}$ *for any* $t \geq 0$*,* $t \in \mathbb{Z}$ *(assuming that the agent stays in the 'done' state with reward 0 after entering one such state).*

Proof. This theorem can be easily proved by concatenating the safety proofs for each time step. When $t = 0$ or $t = 1$, we know that the we explore safely ($s_0 \models \texttt{safe}$ and $s_1 \models \texttt{safe}$) from JSL (since there only one fixed, accurate model). Meanwhile, $\texttt{init}_{\theta_0}$ holds since $\texttt{init}_\theta \rightarrow [\{\texttt{ctrl}_\theta;\texttt{plant}\}^*]\texttt{init}_\theta \wedge \texttt{safe})$ is valid. Then we change θ from θ_0 to θ_1. Because $\texttt{init}_\theta \rightarrow \texttt{init}_{H(\theta,s_0,a_0,s_1)}$, we know that $\texttt{init}_{\theta_1}$ is true. Besides, since H is valid, $\texttt{init}_{\theta_1} \rightarrow [\{\texttt{ctrl}_{\theta_1};\texttt{plant}\}^*]\texttt{safe}$ is still an accurate model.

Similarly, we can prove that for any time step t (as long as the episode is not terminated), if $\texttt{init}_{\theta_t}$ is true, we explores safely in that time step, and $\texttt{init}_{\theta_{t+1}}$ is true. By induction, this holds for any $t \in \mathbb{Z}$, $t \geq 0$. we prove the theorem. □

We need to consider a lot of conditions to make LJSL safe in general cases. Here we only consider a simplified case in experiments: the model does not fully know the ability of the agent. For example, the driver knows the range of the acceleration of the car, but the dispatcher does not. So the dispatcher gives strict restrictions of the speed to avoid crashes in the beginning. But the dispatcher gradually knows that the car's brake is much more efficient by observations, and he adjusts the restrictions to make the system more efficient.

In this case, H only depends on the current parameter θ and action a, where θ is used for modeling the action space. H is valid as long as our model can handle the actual action space, and the action space described by θ is not larger than the actual action space, which is not hard to guarantee. $\texttt{init}_\theta \to \texttt{init}_{H(\theta,s,a,s')}$ for any $s' \sim T(s,a)$ is usually true, since when the action space is larger (using $H(\theta, s, a, s')$ instead of θ), the original initial conditions can be released.

5 Experiments

A traditional RL problem is shortest path planning. We propose to do experiments from Continuous Adaptive Cruise Control (CACC) to eventually solve the robot cleaner problem in a 2D plane with obstacles (penalty) and trash (rewards). The trained model would strive to find the shortest path where it collects all rewards without crossing obstacles, which can be verified using d$\mathcal{L}$.

We perform experiments in three environments.

1. Continuous Adaptive Cruise Control (CACC): This task consists of an agent moving in a 1D-track, controlling the speed in response to a static obstacle in the track. The rewards come from moving in range without crashing into the obstacle.
2. Goal Finding: This task consists of an agent in a 2D plane that moves on circular tracks, a final goal to reach, and a set of obstacles randomly distributed in the plane. The rewards come from successfully reaching the goal without crashing into any obstacle.
3. Pointmess: This task builds upon Goal Finding environment, with additional positive rewards coming from separated collectibles in the plane.

In our experiments, we use the SAC algorithm [11] implemented by Spinning Up [15]. We add support for using GPU to train the agent to increase the speed of training. The environment setups are based on VSRL [6] and OpenAI Gym.

5.1 CACC Task

To create a more challenging task, we modify the ACC problem to make the action space continuous. Specifically, the agent can choose any action between $[-A,\ B]$ $(A,\ B > 0)$ as its acceleration for the next time period (the maximum length of any time period is T). We need to guarantee that the position of the car stays in the region $[0,\ 100]$, where $(-\infty, 0)$ is the crash region. Here we call the modified environment the CACC environment.

To avoid crashing, we build a parameterized model with parameter A and B:

$$
\begin{aligned}
&\texttt{init} \to [\{\texttt{ctrl}; \texttt{plant}\}^*]\texttt{safe},\ \textit{where}\\
&\texttt{init} \equiv x \geq 0 \ \wedge\ (v < 0\ \to\ x \geq \frac{v^2}{2A})\ \wedge\ A > 0\ \wedge\ B > 0\ \wedge\ T > 0,\\
&\texttt{ctrl} \equiv \{t := 0; a := *; ?(a \geq -B\ \wedge\ a \leq A\ \wedge\ (v + aT < 0\ \to\ x + vT + 1/2aT^2\\
&\geq \frac{(v + a * T)^2}{2A})\ \wedge\ (v < 0\ \wedge\ v + aT \geq 0\ \to\ x \geq \frac{v^2}{2a})),\\
&\texttt{plant} \equiv \{x' = v, v' = a, t' = 1\ \&\ t \leq T\},\\
&\texttt{safe} \equiv x \geq 0.
\end{aligned}
$$

The `init` part defines the ranges of constant T and parameters A and B, and insures that the initial state is safe ($x \geq 0$ means the car is in the safe region, and $v < 0\ \to\ x \geq v^2/(2A)$

means if the car is moving to the unsafe region, it can stop before crashing by choosing $a = A$).

The `ctrl` part initializes the clock t, and assign a value in $[-B, A]$ to acceleration a. Besides, we need to guarantee that the acceleration is safe. If the speed is negative after time T, we need to guarantee that we can still stop before crashing by choosing $a = A$ after time T. If the speed is positive after time T, we need to guarantee that if the current speed is negative (otherwise, we are moving away from the unsafe region all the time), the point where $v = 0$ is safe.

The `plant` part describes the movement with acceleration a for at most time T, and the `safe` part is simply $x \geq 0$. By choosing a loop invariant $x \geq 0 \land (v \leq 0 \rightarrow x \geq v^2/(2A))$, this model can be proved automatically by KeYmaera X [8].

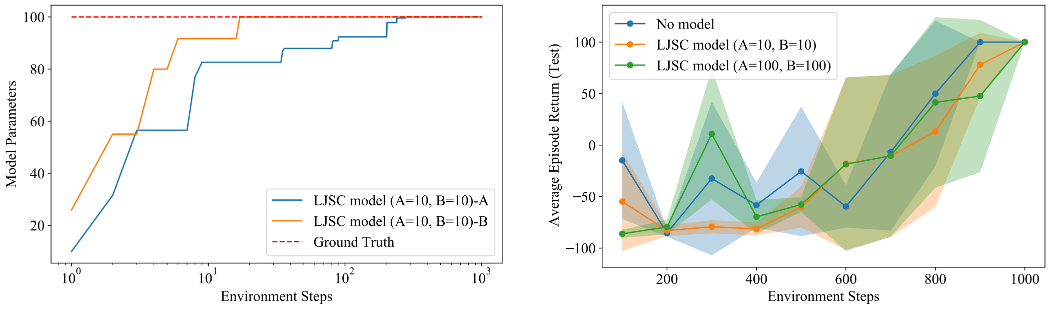
Our model can execute controls successfully, because if $v \geq 0$, at least we can choose any $a \in [0, A]$; if $v < 0$, at least we can choose $a = A$ (from the loop invariant, we know $x \geq v^2/(2A)$), so we can past the test in the `ctrl`).

The CACC model is more realistic than the ACC model since we can choose the acceleration from a continuous space instead of only from discrete options. This makes the proof more difficult, but we complete the proof.

We update our model by observing the acceleration that our car can achieve. For example, if the upper bound of acceleration A is 10, and we find that our agent successfully set its action (acceleration) to 20, then we will update the value of A to 20. So $H(\{A, B\}, s, a, s') = \{\text{min}(-a, A), \text{max}(a, B)\}$. It is obvious that all conditions in Theorem 1 are satisfied.

Results are shown in Figure 2 and Table 2. Here LJSC model ($A = 10,\ B = 10$) means a model with A initialized to 10 and B initialize to 10 (similarly for LJSC model ($A = 100,\ B = 100$)). We run 1000 steps for each model and test the performance after every 200 steps, and use 3 different seeds. The maximum episode length is 100, which means the maximum reward for one episode is 100.

From Figure 2 (a), we can find that even if we use a very conservative LJSC model (A, B initialized to 10), it can learn to be more efficient quickly. From Figure 2 (b), we find that LJSC models can achieve comparable rewards. From Table 2, we find that LJSC models eliminated the crashing cases.



(a) (b)

Figure 2: (a) the learning process of parameters for an imperfect LJSC model (b) test rewards of agents with no model, an LJSC model with perfect initialization ($A = 100,\ B = 100$) and an LJSC model with imperfect initialization ($A = 10,\ B = 10$)

Table 2: Number of crashes during 1000 training steps for different models (5 runs)

Model	Number of crashes
No model	28
LJSC model ($A = 10,\ B = 10$)	0
LJSC model ($A = 100,\ B = 100$)	0

5.2 Goal Finding Task

We define the task of goal finding to be an agent moving in a 2D plane on circular tracks, a fixed position for the final goal (g_x, g_y) and a series of obstacles $\bigcup_{i=1}^{n}(obs_{ix}, obs_{iy})$ separated randomly in the plane. The goal is for the agent to successfully reach the final goal, without crashing into any of the obstacles. An illustration of the environment is defined below in Figure 3.

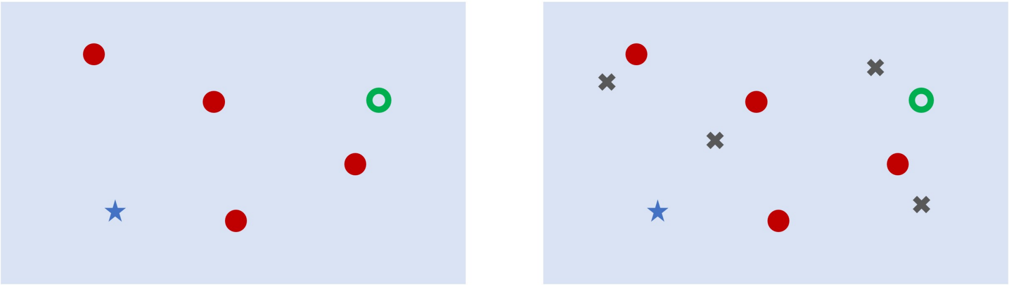

(a) (b)

Figure 3: (a) An example of the environment Goal Finding (b) An example of the environment Pointmess. The star is the moving agent, the green circle is the goal, the red circles are the obstacles, and the black crosses are pointmesses.

The agent is defined with a controller of two variables: a track radius r of the current circular track it is on, and an acceleration a. The movement of the agent is modeled using the following variables: the current position (x, y), a line velocity v, and the direction vector (dx, dy). We define the agent to crash an obstacle in the sense that the distance between the two is less than a constant buffer> 0.

For the sake of simplicity, we don't enforce an area bound on the movement of the agent, but it should ideally not stay too far from the goal as the reward is calculated using the distance towards the goal.

We define the model using the following d$\mathcal{L}$ formulas:

$$\texttt{constBounds} \equiv A > 0 \wedge B > 0 \wedge T > 0 \wedge r_{min} < -\texttt{buffer} \wedge r_{max} > \texttt{buffer} \wedge \texttt{buffer} > 0$$

$$\texttt{valid\_env} \equiv \bigwedge_{i=1}^{n} \{(x - dy \cdot r - obs_{ix})^2 + (y + dx \cdot r - obs_{iy})^2) \geq \texttt{buffer}^2 \wedge$$

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1, j \neq i}^{n} \{(obs_{ix} - obs_{jx})^2 + (obs_{iy} - obs_{jy})^2) \geq \texttt{buffer}^2$$

$$\texttt{init} \equiv \texttt{constBounds} \wedge \texttt{valid\_env} \wedge dx^2 + dy^2 = 1$$

$$\texttt{safe}(r, obs_i) = (x - dy \cdot r - obs_{ix})^2 + (y + dx \cdot r - obs_{iy})^2 \geq (r + \texttt{buffer})^2$$

$$\texttt{ctrl} \equiv \{t := 0; a := *; ?(-B \leq a \leq A); r := *; ?(r_{min} \leq r \leq r_{max} \wedge$$
$$\bigwedge_{i=1}^{n} \texttt{safe}(r, obs_i)) \wedge r! = 0\}$$
$$\texttt{plant} \equiv \{x' = dx \cdot v, y' = dy \cdot v, dx' = -\frac{v \cdot dy}{r}, dy' = \frac{v \cdot dx}{r}, v' = a,$$
$$t' = 1 \& t \leq T \& v \geq 0\}$$
$$\texttt{safe} \equiv \bigwedge_{i=1}^{n} \{(x - obs_{ix})^2 + (y - obs_{iy})^2) \geq \texttt{buffer}^2\} \wedge dx^2 + dy^2 = 1 \wedge v \geq 0.$$

Here `constBounds` summarizes initial requirements for the defined constants. We have defined them using actual constants in the experiments, but we can prove them using broader bounds.

`valid_env` requires the initial position of the agent does not crash into any obstacle. When there's more than one, the random obstacles must not crash into each other.

$\texttt{safe}(r, obs_i)$ is a bool function that takes in the radius and the position of the obstacle and determines whether the agent will crash into the obstacle. Using this we can define `ctrl`, which requires the acceleration a to be bounded, radius r to be bounded and the proposed radius must satisfy the safety requirement. We won't allow radius=0 which is undefined, and there's always a valid radius to choose from as $r_{max} > \texttt{buffer}$.

`plant` defines the actual agent dynamics within one timestep T.

`safe` defines the safety requirement at the end of one loop where the current agent position does not crash into any obstacles and adds restrictions on the model dynamics.

And this completes the model by implementing

$$\texttt{init} \rightarrow [\{\texttt{ctrl}; \texttt{plant}\}^*]\texttt{safe}$$

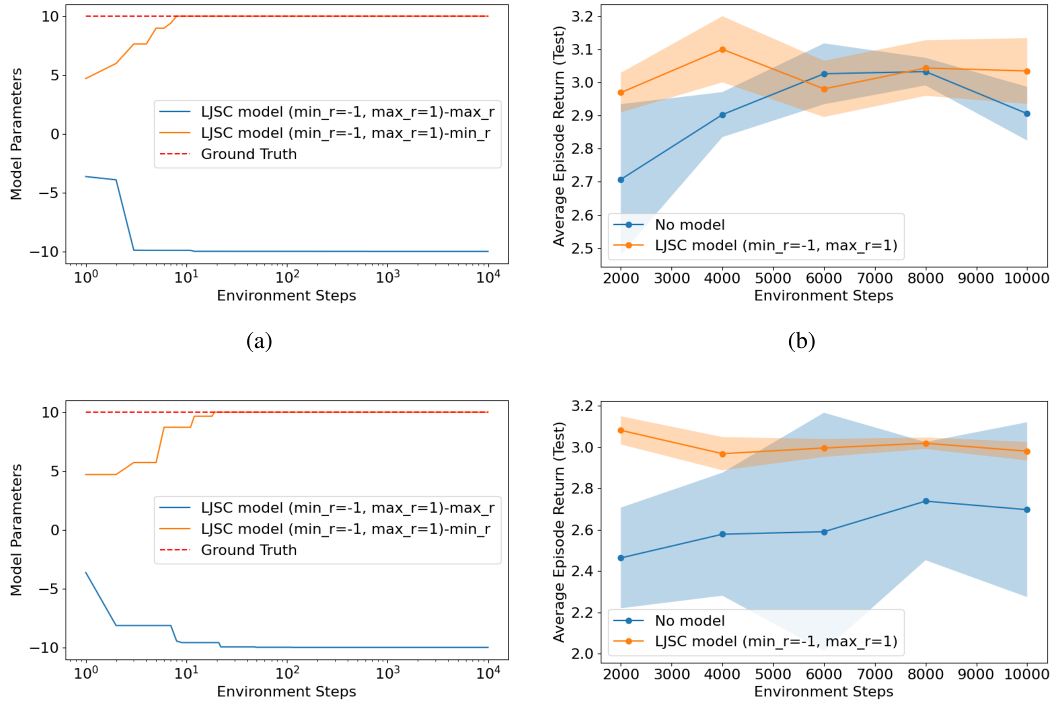
for Goal Finding Task. Observe that with a different number of obstacles, the KeYmaera X implementation would be slightly different. A formal KeYmaera X proof with 2 obstacles can be found in `GoalFinding_Proof.kyx`.

To prove this model, we use the loop invariant

$$\bigwedge_{i=1}^{n} \texttt{safe}(r, obs_i) \wedge dx^2 + dy^2 = 1 \wedge v \geq 0$$

We prove this model is safe by arguing that the controller always makes sure the agent always stays on the track that does not coincide with any obstacle, because $\texttt{safe}(r, obs_i)$ is true in a state that on the current track where the agent moves with radius r, the distance between the current position of the agent and the position of the obstacle $obs_i = (obs_{ix}, obs_{iy})$. This property holds after a single run of `{ctrl;plant}`, because the agent always stays on that track by the definition of dynamics in the differential equation. It can be proved by the simple dI rule.

Results are shown in Figure 4 and Table 3. We have conducted experiments on environments initialized with 1 obstacle and 10 obstacles each. We run 10000 steps for each model and test the performance after every 2000 steps. We run the tests 3 times with random initialization. Figure 4(b) shows the average and standard deviation. As the safety requirement is conservative and does not take into account speed requirements, we omit the step of updating acceleration and instead only consider the radius.



(a) (b)

(c) (d)

Figure 4: (a) the learning process of parameters for an imperfect LJSC model (b) test rewards of agents with no model, an LJSC model with imperfect initialization ($r_{min} = -1, r_{max} = 1$) (c) LJSC model for environment initiated with 10 obstacles with imperfect initialization ($r_{min} = -1, r_{max} = 1$) (d) test rewards of agents for models with 10 obstacles environment.

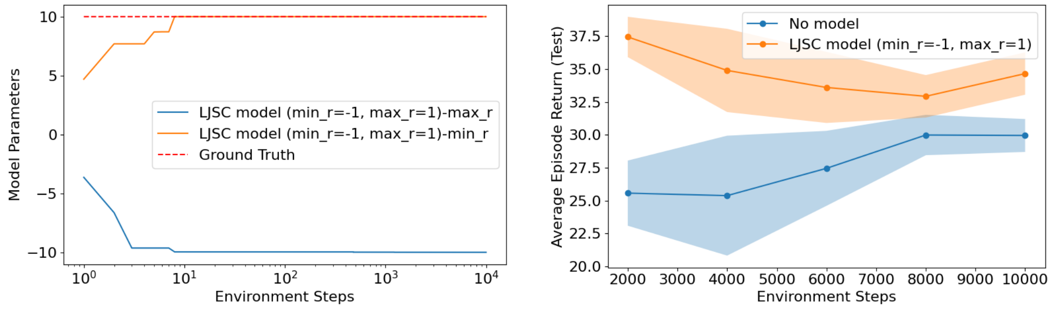
Table 3: Number of crashes during 10000 training steps for different models

Model	Number of crashes
No model (1 obstacle)	8
LJSC model($r_{max} = 1, r_{min} = -1$) (1 obstacle)	0
No model (10 obstacles)	72
LJSC model($r_{max} = 1, r_{min} = -1$) (10 obstacles)	0

For the environment of 1 obstacle, there's not much difference in gained rewards, potentially because it's hard to crash in such an environment even without a safety guarantee. This result is also shown by Table 3. In comparison, the LJSC model achieves more rewards gained for 10 obstacles environment with the same amount of training steps. However, we don't see an upward growth trend in the learned rewards, and this might be because we consider the distance towards the goal as a part of the reward and there are some variances. It could also be the case that we should increase the number of training steps.

5.3 Pointmess Task

We define the task of pointmess to be a forcing reinforcement-learning upgrade upon goal finding task, as the previous task does not enforce agents to move close to obstacles. In addition to the previous environment setting, we add m number of messes ($\bigcup_{i=1}^{m}(pm_{ix}, pm_{iy})$) in the ground for the agent to pick up and gain positive reward. The agent's dynamics stay the same. An illustration of the environment is defined in Figure 3. This addition does



(a)

(b)

Figure 5: (a) the learning process of parameters for an imperfect LJSC model (b) test rewards of agents with no model, an LJSC model with imperfect initialization ($r_{min} = -1, r_{max} = 1$)

Table 4: Number of crashes during 10000 training steps for different models

Model	Number of crashes
No model (10 obstacle, 10 pointmesses)	49
LJSC model($r_{max} = 1, r_{min} = -1$) (10 obstacle, 10 pointmesses)	0

not affect the safety guarantee of the agents' movement and we can use the same safety verification as in Section 5.2.

Results are shown in Figure 5 and Table 4. We have conducted experiments on environments initialized with 10 obstacles and 10 pointmesses. We run 10000 steps for each model and test the performance after every 2000 steps. We run the tests 3 times with random initialization Figure 5(b) shows the average and standard deviation. We observe that the LJSC model gains more rewards with the same amount of training steps.

6 Conclusion

We have proposed the LJSC algorithm and investigated its performance on common reinforcement learning tasks. We have shown that we can successfully update the parameters efficiently while preserving the safe learning guarantee. It also performs better than regular RL without safety control.

We have learnt to model different environments and simulate agent dynamics using Python that transforms from d$\mathcal{L}$ models, and building controller monitor and model monitor as our algorithm proposed.

However, due to lack of resources, the training is timewise-inefficient as one training epoch on tasks Goal Finding and Pointmess would take hours. We also suspect it has not reached the best performance based on the gained rewards. We defined the rewards that better encourage either model to train more efficiently, but there might still be room for improvements. While the model monitor is accurate and guarantees safety, they are inherently conservative on the efficiency of agent movement in reaching the final goal. For example, in the goal finding task, the agents should be able to move in tracks that even collide with the obstacles, will not crash in the current time step. We can also improve by creating finer-grained safety bounds. There's also potential to investigate the algorithm when simultaneously updating more parameters, as well as more general usage of our theorem.

7 Deliverables

The deliverables include:

- KeYmaeraX models and proofs for 3 environments: CACC and Goal Finding(=Pointmesses)
- An implementation of the algorithm and the training code in Python

References

[1] Julian Schrittwieser et al. "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609.

[2] Loris Roveda et al. "Model-based reinforcement learning variable impedance control for human-robot collaboration". In: *Journal of Intelligent & Robotic Systems* (2020), pp. 1–17.

[3] Eric Langlois et al. "Benchmarking model-based reinforcement learning". In: *arXiv preprint arXiv:1907.02057* (2019).

[4] Danijar Hafner et al. "Dream to control: Learning behaviors by latent imagination". In: *International Conference on Learning Representations (ICLR)* (2020).

[5] Nathan Fulton and André Platzer. "Safe reinforcement learning via formal methods: Toward safe control through proof and learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[6] Nathan Hunt et al. "Verifiably safe exploration for end-to-end reinforcement learning". In: *HSCC '21: 24th ACM International Conference on Hybrid Systems: Computation and Control, Nashville, Tennessee, May 19-21, 2021*. Ed. by Sergiy Bogomolov and Raphaël M. Jungers. ACM, 2021, 14:1–14:11. DOI: `10.1145/3447928.3456653`. URL: `https://doi.org/10.1145/3447928.3456653`.

[7] Nathan Fulton and André Platzer. "Verifiably safe off-model reinforcement learning". In: *arXiv preprint arXiv:1902.05632* (2019).

[8] Jean-Baptiste Jeannin et al. "A Formally Verified Hybrid System for the Next-generation Airborne Collision Avoidance System". In: *TACAS*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. LNCS. Springer, 2015, pp. 21–36. DOI: `10.1007/978-3-662-46681-0_2`.

[9] André Platzer. "Differential Logic for Reasoning about Hybrid Systems". In: *HSCC*. Ed. by Alberto Bemporad, Antonio Bicchi, and Giorgio Buttazzo. Vol. 4416. LNCS. Springer, 2007, pp. 746–749. ISBN: 978-3-540-71492-7. DOI: `10.1007/978-3-540-71493-4_75`.

[10] Nathan Fulton et al. "Bellerophon: Tactical Theorem Proving for Hybrid Systems". In: *Interactive Theorem Proving*. Ed. by Mauricio Ayala-Rincón and César A. Muñoz. Cham: Springer International Publishing, 2017, pp. 207–224. ISBN: 978-3-319-66107-0.

[11] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[12] Sebastian Junges et al. "Safety-constrained reinforcement learning for MDPs". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2016, pp. 130–146.

[13] John Schulman et al. "Trust region policy optimization". In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

[14] Joshua Achiam et al. “Constrained policy optimization”. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 22–31.

[15] Joshua Achiam. “Spinning Up in Deep Reinforcement Learning”. In: (2018).