Gymnasium dL

Synthesizing Gymnasium Environments from Structured Annotations in Differential Dynamic Logic

Thuan Cao[1] and Stefan Mitsch[1 [0000−0002−3194−9759]]

School of Computing, DePaul University
{tcao8,smitsch}@depaul.edu

Abstract. Reinforcement learning (RL) performs well in tasks when target outcomes are well-defined, yet it lacks formal safety guarantees. This limitation stems from the black-box nature of RL policies, finite training horizons, and a semantic mismatch between formal safety models (characterized by nondeterministic physics and action safety envelopes) and the stochastic action spaces typical of RL environments. Consequently, translating formally verified constraints into trainable RL objectives remains a critical, manual, and error-prone process. To bridge this gap, we propose a formal specification language that unifies verified safety models and RL environments through automated synthesis. Building on differential dynamic logic (dL), our approach extends dL with structured annotations that map formal safety conditions to action spaces, environment simulators, and reward signals within the Gymnasium framework. These annotations allow practitioners to precisely align mathematical safety specifications with learning dynamics. Our framework automatically generates both the simulator and its associated reward structure, ensuring that the RL agent optimizes under exactly the same constraints formally proven correct in dL. This eliminates manual reward engineering and guarantees that training objectives remain consistent with verified safety properties.

Keywords: Differential dynamic logic, Reinforcement learning, Gymnasium, Synthesis.

1 Introduction

Reinforcement Learning (RL) has displayed notable success in addressing complex control tasks, including robotics and autonomous navigation, by optimizing policies through trial-and-error interactions with the environment. In safety-critical domains, such as autonomous driving and medical robotics, provable safety is a major goal in the deployment of RL-based controllers [20, 9].

A fundamental challenge in learning provably safe controllers is the significant semantic disconnect between formal verification and reinforcement learning. Formal methods, such as Differential Dynamic Logic (dL) [15], offer rigorous frameworks for proving that a system's continuous-time dynamics will not

violate safety invariants under specified control constraints. In contrast, RL environments, typically implemented in frameworks such as Gymnasium, operate with discrete-time steps, stochastic action spaces, and heuristic reward functions. Translating a formally verified safety model into a trainable RL environment currently requires a manual and labor-intensive process. This manual translation is highly susceptible to errors; any specification drift between the verified formal model and the simulation environment can invalidate the original safety proofs.

To address this gap, this work puts forward Gymnasium dL, a framework that enables automated synthesis of RL environments directly from structured annotations in dL. This approach keeps the learning environment's physics, action spaces, and reward signals consistent with the verified safety specification.

The primary contributions of this work are as follows: (i) Formal Annotation Syntax: A non-invasive annotation language for dL is introduced that maps logical predicates to RL-specific concepts such as observation spaces, reward shaping, and termination conditions. (ii) Automated Synthesis: A transpiler is provided that converts KeYmaera X models into executable Gymnasium environments, automatically generating high-fidelity ODE solvers for dynamics and SMT-seeded samplers for initial states. (iii) Correct-by-Construction Training: It is demonstrated that RL agents trained in these synthesized environments can discover optimal, deadline-aware policies while remaining strictly bounded by the formal safety envelopes established within the original logic.

Through integrating formal safety models with the RL training loop, Gymnasium dL facilitates the reward engineering process by providing a systematic mapping from logical specifications to numerical signals. This fundamentally grounds the agent's objective function in formal verification, lessening the likelihood of specification errors and facilitating formal verification of learned controllers through neural network verification tools (e.g., [20, 9]).

2 Background

This section reviews differential dynamic logic, reinforcement learning, reward shaping, and the Gymnasium reinforcement learning framework.

2.1 Differential Dynamic Logic

Differential dynamic logic dL [15] provides a formal framework for specifying and reasoning about hybrid systems through *hybrid programs* (HP). The syntax of HP is defined by the following grammar, where α and β denote hybrid programs, x represents a variable, e and $f(x)$ are arithmetic terms over the reals using operators $+, -, \cdot, /,$, and Q is a dL formula:

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

The assignment operator $x := e$ updates variable x with the value of term e (e.g., computing stopping distance d from speed v with brakes B as $d := \frac{v^2}{2B}$).

The nondeterministic assignment $x := *$ assigns an arbitrary real value to x. The test operator $?Q$ aborts the current execution path if formula Q evaluates to false, potentially allowing backtracking to other nondeterministic branches. A common modeling technique combines nondeterministic assignments with tests to restrict values (e.g., selecting a control input within actuator limits: $a := *; ? - B \leq a \leq A$). Differential equations $\{x' = f(x) \ \& \ Q\}$ evolve continuously along solutions of $x' = f(x)$ for an arbitrary duration, provided the evolution domain constraint Q remains true throughout. For instance, velocity may change according to acceleration but must not become negative when braking: $\{v' = -B \ \& \ v \geq 0\}$. Finally, hybrid programs incorporate structural operators: nondeterministic choice $\alpha \cup \beta$ executes either α or β (e.g., accelerating or braking); sequential composition $\alpha; \beta$ executes α followed by β on the resulting states (e.g., control action followed by dynamics); and nondeterministic repetition α^* iterates α for an arbitrary number of steps in $\mathbb{N}_0$ (e.g., a discrete-time control loop interacting with the environment).

The logical formulas of dL express properties over these programs. Their syntax follows the grammar below, where P and Q denote dL formulas, f and g are real-arithmetic terms, $\sim \in \{<, \leq, =, \neq, \geq, >\}$ is a comparison operator on reals, x is a variable, and α is a hybrid program:

$$P, Q \ ::= \ f \sim g \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q \mid \forall x \, P \mid \exists x \, P \mid [\alpha] \, P \mid \langle \alpha \rangle P$$

The arithmetic and boolean operators adhere to standard first-order real logic, with quantifiers ranging over the real numbers. For any hybrid program α and dL formula P, the universal modality $[\alpha] \, P$ holds in a given state if and only if P is true after every possible execution of α. Its dual, the existential modality $\langle \alpha \rangle P$, holds if and only if P is true after at least one possible execution.

The formal semantics of dL are grounded in a Kripke structure in which the states of the Kripke model are the states of the hybrid system. A state is defined as a valuation map $\omega : \mathcal{V} \rightarrow \mathbb{R}$ that assigns a real number $\omega(x)$ to each variable $x \in \mathcal{V}$. While the continuous dynamics and nondeterministic branching inherent to HPs render them highly expressive for formal verification, they pose significant challenges when deploying hybrid program semantics directly within reinforcement learning environments.

2.2 Reinforcement Learning and Reward Shaping

Reinforcement learning involves training an agent to achieve a specified goal through sequential environmental interaction, with the objective of maximizing cumulative reward. At each decision step, the agent selects an action $a \in \mathcal{A}$. In response, the environment transitions from its current state $s \in \mathcal{S}$ to a subsequent state $s' \in \mathcal{S}$. A reward function then evaluates the desirability of this transition: positive rewards reinforce trajectories that align with task objectives, whereas negative rewards penalize deviations from safe or desired behavior. Conceptually, this mechanism steers the agent away from undesirable regions of the state space while incentivizing progression toward goal-oriented configurations.

In [13, 16], a principled methodology is introduced for synthesizing safety-critical components of the reward function directly from hybrid systems models.

In many control tasks, agents must optimize competing objectives, such as minimizing travel time while adhering to speed constraints. To address this multi-objective nature, the reward function is augmented with auxiliary shaping signals, a practice known as *reward shaping*. The primary motivation for reward shaping is to provide dense, informative feedback for intermediate subtasks that contribute to the overall objective. By smoothing the underlying reward landscape and mitigating reward sparsity, reward shaping accelerates convergence and enhances sample efficiency during training. For instance, [16] uses distinct safety-preserving shapes of a reward function to demonstrate the efficacy of this approach to guide constraint-aware policy learning.

3 Related Work

The integration of formal methods into RL is an emerging field that seeks to handle the inherent lack of safety and sample efficiency in black-box neural policies. This work operates at the intersection of formal specification, safe RL, and environment modeling, specifically targeting the semantic gap between continuous-time formal logic and discrete-time RL interaction.

Safe RL and Action Correction. A primary focus in Safe RL is ensuring that the agent adheres to constraints during both training and deployment. Shielding is a prominent approach in which a reactive system monitors and corrects agent actions based on temporal logic specifications [1]. While shielding provides strong runtime guarantees, it frequently depends on discrete state abstractions or external enforcement of agent behavior. In contrast, Gymnasium dL utilizes the continuous-time semantics of dL to synthesize the environment's physics and reward structures directly from the formal model. This approach attempts to inherently align the agent's entire learning context with the verified safety model, rather than retroactively correcting it by an external monitor.

Similarly, Safe Action Projection methods [4] employ a corrective layer to project unsafe actions back into a feasible set using linearized models. While being effective, these methods regularly depend on learned approximations of constraints. The present framework harnesses dL to provide exact symbolic boundaries, forming a foundation for action-correction methods grounded in formal verification rather than empirical approximation.

Neuro-Symbolic and Verifiable Policies. Beyond action-filtering, neuro-symbolic RL seeks to integrate logic directly into the policy. The REVEL framework [2] utilizes a mirror-descent algorithm to transition between verifiable symbolic policies and high-performance neural policies. While REVEL ensures safety by constraining the policy space by means of iterative verification, the present approach focuses on the environment space. By turning dL specifications into Gymnasium dynamics, this method creates a correct-by-construction training environment.

As a result, any policy—whether purely neural or neuro-symbolic—is bounded by the formal physics defined in the original dL model, effectively constructing the laws of the environment directly from the formal model.

Formal Specification of Rewards and Scenes. The use of formal logic to define RL objectives has been explored primarily through Linear Temporal Logic (LTL) and Reward Machines (RMs) [8, 3]. These white-box rewards improve sample efficiency by exposing the task structure to the agent. While LTL is effective for discrete sequences, it lacks primitives for continuous-time hybrid dynamics. The present work complements these approaches by using dL to extend the benefits of formal specification to continuous-state safety invariants.

Furthermore, this work relates to Environment Modeling Languages such as Scenic [6], a domain-specific language for describing spatial scenarios. While Scenic excels at defining the static setup of a scene (the "where"), it does not natively provide formal guarantees regarding the continuous-time dynamics (the "how") during execution. Gymnasium dL resolves this limitation: while Scenic defines the distribution of initial states, the use of dL specifies the allowable evolution of those states, ensuring that the resulting Gymnasium environment remains consistent with the intended safety specifications throughout training.

3.1 Bridging the Semantic Gap in Hybrid Systems

This work builds on the legacy of formal verification for safety-critical systems, e.g., for ground robots [11]. Traditionally, these approaches focus on offline verification of controllers. The semantic gap between the continuous-time differential equations used in verification and the discrete-time step functions used in RL remains a challenge. Existing bridges usually rely on manual discretization, which is prone to specification drift, a state in which the trained agent succeeds in a simulation that no longer is consistent with the verified model's assumptions. While runtime monitoring tools like ModelPlex [13] can detect such specification drift, our strategy eliminates the need for manual translation by automating the synthesis of the Gymnasium environment directly from the dL specification. This ensures that the physics engine, safety monitors, and reward signals are derived from a single source of truth [16], hence bridging the gap between formal verification and active policy synthesis.

4 Approach

This framework joins the gap between formal verification of hybrid systems in KeYmaera X [7], a theorem prover for dL that utilizes an extended ASCII syntax to structure models, and practical control by treating the dL abstract syntax tree (AST) as a blueprint for a Markov Decision Process (MDP). In this context, the MDP provides the mathematical basis for the reinforcement learning agent's environment: the dL state variables define the state space $\mathcal{S}$, the nondeterministic choices in the Hybrid Program define the action space $\mathcal{A}$, the ordinary

differential equations (ODEs) define the Transition Function $\mathcal{P}$, and the annotated predicates specify the reward function $\mathbf{R}$. Following the translation rules in this section, KeYmaera X models are automatically translated into Gymnasium-compatible environments. This process converts symbolic differential equations into executable physics simulations and transforms logical safety requirements into numerical reward signals for the reinforcement learning agent.

4.1 Extended Formal Syntax and RL-Annotations

The RL-Annotation Syntax is a comment-based domain-specific language (DSL) embedded within the `dL` ASCII syntax [12]. The annotations are separated into variable-level and predicate-level annotations to enable a clear mapping to the Gymnasium environment's Observation Space and Reward or Termination Logic, respectively (see Appendix B for a syntax example).

Using a comment-based DSL, we achieve non-invasive integration: the annotated model remains a syntactically valid KeYmaera X file. KeYmaera X ignores the annotated metadata, so that the formal proof is not compromised by training-specific heuristics. Adding annotations to KeYmaera X files ensures that the specification used in the proof is also used to incentivize or penalize the RL agent. This prevents specification drift where the verified model and the trained agent operate on different assumptions.

Variable Annotations (`@var`, `@const`). These annotations are positioned above `Real` declarations within the `ProgramVariables` or `Definitions` sections. They specify the numerical boundaries and metadata for the environment's state. Variable annotations follow the syntax below:

$$
\begin{aligned}
\langle \text{VarAnnotation} \rangle &\rightarrow \langle \text{Var} \rangle \mid \langle \text{Const} \rangle \\
\langle \text{Var} \rangle &\rightarrow \text{“}\texttt{@var}\text{”}\ \text{“}\texttt{:}\text{”}\ \langle \text{VarAttrList} \rangle \\
\langle \text{Const} \rangle &\rightarrow \text{“}\texttt{@const}\text{”}\ \text{“}\texttt{:}\text{”}\ \langle \text{VarAttrList} \rangle \\
\langle \text{VarAttrList} \rangle &\rightarrow \langle \text{VarAttr} \rangle \mid \langle \text{VarAttr} \rangle\ \text{“}\texttt{,}\text{”}\ \text{VarAttrList} \\
\langle \text{VarAttr} \rangle &\rightarrow \langle \text{Bound} \rangle \mid \langle \text{Def} \rangle \mid \langle \text{Plot} \rangle \\
\langle \text{Bound} \rangle &\rightarrow \text{“}\texttt{bound}\text{”}\ \text{“}\texttt{=}\text{”}\ \text{“}\texttt{[}\text{”}\ \langle \text{Num} \rangle\ \text{“}\texttt{:}\text{”}\ \langle \text{Num} \rangle\ \text{“}\texttt{]}\text{”} \\
\langle \text{Def} \rangle &\rightarrow \text{“}\texttt{def}\text{”}\ \text{“}\texttt{=}\text{”}\ \langle \text{String} \rangle \\
\langle \text{Plot} \rangle &\rightarrow \text{“}\texttt{plot}\text{”}\ \text{“}\texttt{=}\text{”}\ \langle \text{PlotType} \rangle \\
\langle \text{PlotType} \rangle &\rightarrow \text{“}\texttt{line}\text{”} \mid \text{“}\texttt{hist}\text{”} \mid \text{“}\texttt{scatter}\text{”} \mid \text{“}\texttt{none}\text{”}
\end{aligned}
$$

where

- `@var` associates a `Real` program variable with an element in the Gymnasium Observation Space. The `bound` attribute specifies the `Box` limits.
- `@const` associates a `Real` definition with a fixed parameter. If a `bound` is specified, the value is sampled once during `reset()`.
- `bound` defines the numerical interval $[min : max]$ for the associated element. For `@var`, this parameter restricts *initial value sampling* during `reset` to ensure that the agent begins within a realistic range specified by the user, e.g., car velocity between -30 and 200 km/h. Values outside this interval are considered unrealistic and are excluded from the starting state distribution. For `@const`, the interval specifies the range used for domain randomization.

- `def` supplies a human-readable string that describes the variable.
- `plot` defines the visualization strategy for the variable during training diagnostics. `line` tracks values over time, `hist` displays the distribution of visited states, and `scatter` visualizes spatial relationships or phase portraits.

In the following example, the variable x representing the vehicle position will be bounded to the interval $[0, 100]$, updated in the step function of the RL environment, and plotted in a line graph of values over time for debugging purposes, whereas the constant definition μ of the friction constant will be sampled once at learning episode starts from the interval $[\frac{4}{5}, \frac{6}{5}]$.

```
// @var: bound=[0:100], plot=line, def="vehicle position"
Real x;
// @const: bound=[0.8:1.2], def="friction coefficient"
Real mu;
```

Predicate Annotations (`@safety`, `@progress`, `@stability`). Predicate annotations are placed above `Bool` or `Real` definitions. These annotations define how the truth value of a dL formula determines rewards and episode transitions.

⟨PredAnnotation⟩ → ⟨Category⟩ “:” ⟨PredAttrList⟩
⟨Category⟩ → “`@safety`” | “`@progress`” | “`@stability`”
⟨PredAttrList⟩ → ⟨PredAttr⟩ | ⟨PredAttr⟩ “,” ⟨PredAttrList⟩
⟨PredAttr⟩ → ⟨RewardAttr⟩ | ⟨ControlAttr⟩
⟨RewardAttr⟩ → “`reward =`” ⟨Expr⟩ | “`penalty =`” ⟨Expr⟩ | “`shape =`” ⟨Expr⟩
⟨ControlAttr⟩ → ⟨Trigger⟩ “=” ⟨Action⟩ [“:” ⟨Status⟩]
⟨Trigger⟩ → “`on_success`” | “`on_failure`”
⟨Action⟩ → “`terminate`” | “`truncate`” | “`continue`”
⟨Status⟩ → “`success`” | “`failure`” | “`none`”

where

- `@safety` designates invariants and safety conditions and typically triggers `truncate:failure` when the formula evaluates to false.
- `@progress` indicates goal conditions and typically triggers `terminate:success` when the formula evaluates to true.
- `@stability` applies to Lyapunov-like functions to provide continuous `shape` rewards without necessarily terminating the episode.
- `reward` and `penalty` specify sparse rewards given upon the satisfaction or violation of the predicate.
- `shape` defines a continuous reward signal (potential field) used to guide the agent's gradient descent toward a goal or away from a hazard.
- `on_success/failure` defines the control flow of an episode, whether to `terminate` (nominal end), `truncate` (violation/time-out), or `continue`.

For example, the safety and progress logical categories of the RL environment are annotated at predicates in the dL ASCII syntax as follows:

```
// @safety: penalty=-1000, shape=d, on_failure=truncate:failure
Bool safeDist(Real d) <-> d > 0;

// @progress: reward=500, shape={-abs(x - gx)}, on_success=terminate:success
Bool atGoal(Real x, Real gx) <-> abs(x - gx) < 0.5;
```

4.2 Structural AST Design

We parse KeYmaera X models into an intermediate representation designed for RL synthesis, which allows us to map symbolic logic to imperative Python code.

- **Variable & Type Nodes:** `VariableDeclarationConfig` nodes capture the name, type, and physical bounds. These are used to programmatically synthesize the `gym.spaces.Box` dimensions and define the sampling volume for the SMT solver (used for initial value sampling in Section 4.5).
- **Formula Nodes:** These nodes translate symbolic logic into numerical execution. The generator traverses these nodes to convert binary and unary operators into Python lambda functions, enabling the Gymnasium environment to evaluate guards and invariants during runtime.
- **Hybrid Program Nodes:** These map the discrete and continuous statements of the hybrid program. Deterministic assignments ($:=$) are translated into state updates. Crucially, nondeterministic choices ($++$) and assignments ($:= *$) are identified as decision points triggering the automatic synthesis of `Discrete` and `Box` action spaces, respectively.

4.3 Refinement from dL to Python

The transition from symbolic logic in dL to an executable Gymnasium environment is achieved through a syntax-directed translation mapping, denoted by the function $\mathsf{Py}(\cdot)$. Whereas traditional compilers target deterministic execution (e.g., a deterministic subset of hybrid programs in [19]), we resolve logical nondeterminism by mapping it to the Reinforcement Learning action space $\mathcal{A}$, thereby enabling the agent to explore the safety envelope during runtime.

Program and Control Translation The control logic α and continuous dynamics are compiled into state-update functions within the Gymnasium `step()` method. The approach distinguishes between deterministic assignments and the resolution of nondeterministic choices in the action space.

$$
\begin{aligned}
\mathsf{Py}(x := \theta) &\quad\triangleright\quad \texttt{self.state['x']} = \mathsf{Py}(\theta) \\
\mathsf{Py}(?P;\alpha \cup ?\neg P;\beta) &\quad\triangleright\quad \textbf{if } \mathsf{Py}(P) : \mathsf{Py}(\alpha) \textbf{ else: } \mathsf{Py}(\beta) \\
\mathsf{Py}((?P;\alpha)^*;?\neg P) &\quad\triangleright\quad \textbf{while } \mathsf{Py}(P) : \mathsf{Py}(\alpha) \\
\mathsf{Py}(\alpha \cup \beta) &\quad\triangleright\quad a \sim \mathcal{A}_{\text{discrete}};\ \textbf{if } a = 0 : \mathsf{Py}(\alpha) \textbf{ else: } \mathsf{Py}(\beta) \\
\mathsf{Py}(x := *) &\quad\triangleright\quad x \sim \mathcal{A}_{\text{box}}; \texttt{self.state['x']} = x \\
\mathsf{Py}(\{x' = f(x)\ \&\ H\}) &\quad\triangleright\quad \texttt{solve\_ivp}(\mathsf{Py}(f(x)), \text{events}{=}\mathsf{Py}(H))
\end{aligned}
\tag{1}
$$

For $\alpha \cup \beta$ and $x := *$, we automatically synthesize the Gymnasium `ActionSpace` by aggregating all nondeterministic operators present within the control sequence. The evolution domain H is compiled as a termination event for the ordinary differential equation (ODE) solver to maintain physical validity.

Arithmetic and Logical Expressions To enable high-performance simulation, symbolic terms θ, η and formulas P, Q are translated into vectorized Python expressions. This translation allows the environment to evaluate guards, invariants, and rewards as native Boolean checks.

Arithmetic Terms: The mapping for arithmetic employs a recursive structure to ensure that dL operators correspond to their `NumPy` or native Python equivalents.

$$\begin{aligned} \mathsf{Py}(z) &\triangleright z && \text{(for number literal } z\text{)} \\ \mathsf{Py}(x) &\triangleright \texttt{self.state['x']} && \text{(for variable } x\text{)} \\ \mathsf{Py}(\theta+\eta) &\triangleright \mathsf{Py}(\theta)+\mathsf{Py}(\eta) && \text{(similar for } -,\cdot,/,**\text{)} \\ \mathsf{Py}(\mathrm{abs}(\theta)) &\triangleright \texttt{np.abs}(\mathsf{Py}(\theta)) \end{aligned} \tag{2}$$

Boolean Formulas: Formulas are evaluated as Python `bool` values. The dL constants for truth ($\top$) and falsity ($\bot$) are Python's `True` and `False`, respectively.

$$\begin{aligned} \mathsf{Py}(\top) &\triangleright \text{True} \qquad \mathsf{Py}(\bot) \triangleright \text{False} \\ \mathsf{Py}(\theta<\eta) &\triangleright \mathsf{Py}(\theta)<\mathsf{Py}(\eta) \quad \text{(similar for } \leq,=,\neq,\geq,>\text{)} \\ \mathsf{Py}(\neg P) &\triangleright \textbf{not } \mathsf{Py}(P) \\ \mathsf{Py}(P\wedge Q) &\triangleright \mathsf{Py}(P) \textbf{ and } \mathsf{Py}(Q) \\ \mathsf{Py}(P\vee Q) &\triangleright \mathsf{Py}(P) \textbf{ or } \mathsf{Py}(Q) \end{aligned} \tag{3}$$

4.4 Mapping Logic to RL Components

To reconcile discrete logical satisfaction with continuous optimization, a mapping is established from formal predicates to reward signals. This approach transforms Boolean logic into a continuous metric space.

Predicate Normalization as Value. When a predicate name Q is used as a value within a reward expression, such as in metadata fields like `shape` or `penalty`, we apply predicate normalization. This process converts formulas into a continuous degree-of-truth signal. For a predicate $Q \equiv (e_1 \bowtie e_2)$, where $\bowtie$ denotes a relational operator, the value is represented as a distance-to-satisfaction metric $\delta(Q) \in \mathbb{R}$. To ensure that the agent maximizes the reward by satisfying the logical constraints, δ is defined so that a higher value consistently indicates a state closer to, or further within, the set Q:

$$\delta(e_1 \bowtie e_2) := \begin{cases} \omega \cdot (\mathsf{Py}(e_2) - \mathsf{Py}(e_1)) & \text{if } \bowtie \in \{<,\leq,>,\geq\} \\ -|\mathsf{Py}(e_1) - \mathsf{Py}(e_2)| & \text{if } \bowtie \in \{=\} \end{cases} \tag{4}$$

where the directional multiplier ω is defined as:

$$\omega = \begin{cases} 1 & \text{if } \bowtie \in \{<,\leq\} \\ -1 & \text{if } \bowtie \in \{>,\geq\} \end{cases} \tag{5}$$

This formulation guarantees that for inequalities, $\delta > 0$ indicates satisfaction, whereas for equalities, the maximum attainable value is 0, achieved exclusively when $e_1 = e_2$. This structure provides a continuous gradient that the reinforcement learning agent can exploit to approach the safety envelope.

Predicate-to-Reward Mapping. To map logic to reinforcement learning, we define the metadata space $\mathcal{M}$ as the set of possible reward annotations (e.g., safety or progress tags) and $\mathcal{E}$ as the set of real-valued arithmetic expressions over the system variables. Predicates P augmented with metadata $m \in \mathcal{M}$ are compiled into the environment's `reward` calculation. The reward components ρ (penalty), γ (reward), and σ (shaping) are defined as elements of the expression set $\mathcal{E}$. The mapping $\mathsf{Py}(P, m)$ separates the logical truth value into discrete and continuous components. Here, $\mathbb{1}_{\mathsf{Py}(P)}$ denotes the indicator function for the truth value of the translated predicate. The synthesized reward r_t is defined as follows:

$$\begin{aligned} \mathsf{Py}(P, @\text{safety: penalty}{=}\rho, \text{shape}{=}\sigma) \quad &\triangleright \quad r_t = (1 - \mathbb{1}_{\mathsf{Py}(P)}) \cdot \mathsf{Py}(\rho) + \mathsf{Py}(\sigma) \\ \mathsf{Py}(P, @\text{progress: reward}{=}\gamma, \text{shape}{=}\sigma) \quad &\triangleright \quad r_t = \mathbb{1}_{\mathsf{Py}(P)} \cdot \mathsf{Py}(\gamma) + \mathsf{Py}(\sigma) \end{aligned} \tag{6}$$

Episode Lifecycle Mapping. The metadata attributes `on_success` and `on_failure` specify the discrete boundaries and outcomes of an episode. According to the grammar, each attribute comprises an action, which determines the Gymnasium control flag, and an optional status, which specifies the categorical outcome.

The mapping $\mathsf{Py}(\cdot)$ converts these attributes into state updates for the environment's `step` function. Recording the outcome involves updating the `info` dictionary, which is standard practice for tracking metrics such as success rate in reinforcement learning.

$$\begin{aligned} \mathsf{Py}(P, @\text{on\_fail}{=}\text{trunc}{:}S) \quad &\triangleright \text{ if } \neg\mathsf{Py}(P) : \text{trunc}{=}\text{T, info['stat']}{=}S \\ \mathsf{Py}(P, @\text{on\_fail}{=}\text{term}{:}S) \quad &\triangleright \text{ if } \neg\mathsf{Py}(P) : \text{term}{=}\text{T, info['stat']}{=}S \\ \mathsf{Py}(P, @\text{on\_succ}{=}\text{term}{:}S) \quad &\triangleright \text{ if } \mathsf{Py}(P) : \text{term}{=}\text{T, info['stat']}{=}S \\ \mathsf{Py}(P, @\text{on\_succ}{=}\text{cont}{:}S) \quad &\triangleright \text{ if } \mathsf{Py}(P) : \text{info['stat']}{=}S \end{aligned} \tag{7}$$

Here, $S \in \{\text{success, failure, none}\}$. This mechanism enables us to differentiate among various types of episode terminations: (i) Safety Violations: mapped via `@safety` to `trunc:failure`; indicates the agent exited the valid state space. (ii) Goal Achievement: mapped via `@progress` to `term:success`; indicates task completion. (iii) Soft Constraints: uses `continue` for status updates (e.g., checkpoints) without interrupting the simulation.

By extracting the physical dynamics into a high-precision ODE solver and the logical invariants into a dense reward function, we actively guide the RL agent by the formal safety properties of the original dL model.

4.5 SMT-Seeded Initial State Sampling

To train the agent on a distribution strictly aligned with the formal proof, Gymnasium dL automates state initialization based on the model's preconditions P (in a typical $P \to [\alpha]Q$ safety shape of dL models). However, standard SMT solvers such as Z3 are typically deterministic, leading the agent to consistently start from the same state and thereby limiting generalization.

To introduce sufficient variability for training, a *Soft-Constraint Targeting* strategy is implemented. At each episode reset, the framework performs the following steps: (i) Target Selection: A randomized target point $\mathbf{s}_{target}$ is sampled uniformly from the operational range defined by the bounds. (ii) Constraint Formulation: The formal dL preconditions are encoded as hard constraints that must be satisfied. (iii) Optimization: The Z3 optimizer is tasked with finding a state $\mathbf{s}_0$ such that: $\mathbf{s}_0 \models$ Preconditions with $\min \|\mathbf{s}_0 - \mathbf{s}_{target}\|_2$. This approach guarantees that each initial state is formally valid and enhances coverage of the reachable state space $\mathcal{S}$ during training.

5 Evaluation

The primary objective of this evaluation is to demonstrate that Gymnasium dL effectively bridges the semantic gap between formal dL specifications and practical reinforcement learning. We structure our assessment around three core questions: (i) Does Gymnasium dL correctly encode formal safety invariants as termination signals? (ii) Do the synthesized dynamics and safety monitors behave predictably under constrained control? (iii) Can an RL agent discover optimal, deadline-aware policies when both liveness and safety constraints are annotated?

5.1 Experimental Setup

All experiments were conducted using environments synthesized directly from ground robot models [11] via Gymnasium dL (see Appendix B for annotations). The configuration below details the shared infrastructure and agent designs.

Environment Synthesis and Dynamics. Continuous-time dynamics were simulated using `scipy.integrate.solve_ivp` [21], with ODEs extracted directly from the `stepFunc` blocks of the source dL models. Initial states for each episode were sampled via an SMT-seeded strategy, leveraging the Z3 solver [5] to satisfy `assumption` predicates and ensure diverse, valid starting conditions.

Reward Mapping and MDP Formulation. The base reward structure was automatically derived from `@safety` and `@progress` annotations. Safety violations triggered immediate truncation (`truncate:failure`), while progress toward goals yielded shaped rewards (details below for each agent configuration). The environment's state space $\mathcal{S}$ and action space $\mathcal{A}$ are derived directly from the dL program variables and the `controlFunc` choices, respectively. The representation of the action space varied between the experimental setups.

Agent Configurations and Training Protocols. Details of configurations are in Appendix C. Tests 1 and 2 use the Stable Baselines3 (SB3) [17] implementation of Proximal Policy Optimization (PPO) [18]. The environment exposed a `Dict` action space, combining continuous assignments and discrete control choices. A custom `FlattenActionDictWrapper` converted this into a unified $\mathbb{R}^5$ Box space for SB3 compatibility, clipping the discrete component to its valid integer range. Training proceeded for $10,000$ timesteps with a constant learning rate.

Test 3 employed a custom PyTorch-based [14] PPO implementation featuring dynamic action masking. The observation space was augmented with six derived features (e.g., waypoint distances, velocity flags), yielding a normalized 14D input vector bounded to $[-1.5, 1.5]$. Logit-level masking enforced `dL` guards at decision epochs, restricting exploration to logically valid actions. Training ran for $10,000$ episodes, with parameter updates applied every $4,096$ environment steps. In addition to the base rewards from `@safety` and `@progress` annotations, a multi-component reward shaping scheme was applied. This included dense progress rewards based on changes in distance to the waypoint, penalties for high velocity near the goal, and a small penalty for time progression, alongside explicit bonuses/penalties for successful goal achievement or invariant violations. The Adam optimizer's [10] learning rate was linearly annealed from 1×10^{-4} down to 5×10^{-6} over the course of training.

5.2 Test 1: Safety Invariants

The first experiment isolates safety synthesis using a purely invariant-driven model ([11, Theorem 1: Static Safety]). Its objective is to verify whether Gymnasium `dL` accurately translates the collision avoidance guard `isSafe` into an episode termination signal.

```
// @safety: penalty=-1000, shape={(abs(x-xo) - stopDist(v)) +
    ↪ (abs(y-yo) - stopDist(v))}, on_failure=truncate:status=failure,
    ↪ on_success=continue:status=none
Bool isSafe(Real x, Real y, Real v, Real xo, Real yo) <-> (abs(x-xo) >
    ↪ stopDist(v) | abs(y-yo) > stopDist(v));

HP controlFunc ::= {
  if (v + A*ts < maxV && dist > stopDist(v + A*ts)) { a := A }
  else { a := -b; } // Fallback to braking
}
```

Results and Analysis. The agent achieved a high mean reward ($\approx 1.97 \times 10^5$) and maximized episode lengths by maintaining a constant velocity of $v = 0$. This safe-lock phenomenon confirms the fidelity of Gymnasium `dL`: the agent indefinitely satisfies the safety invariant by consistently selecting the braking/stalled branch of `controlFunc`. While formally safe, the absence of liveness annotations renders functional movement unreachable. These results validate the correctness of the safety-to-termination mapping while highlighting the necessity of progress-oriented predicates for non-trivial control tasks.

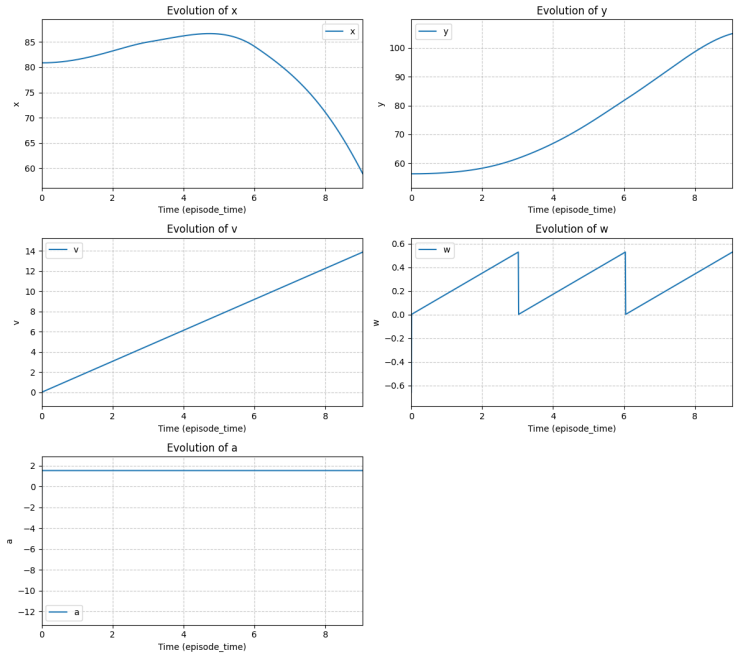

Fig. 1. Forced dynamics illustrate safety monitor's reaction to stopping distance

5.3 Test 2: Stress-testing Dynamics and Safety Monitors

To probe the environment's behavior in the presence of unsafe control, we manually restrict the action space to force continuous acceleration ($a := A$). This isolates the interaction between the ODE solver and the safety monitor.

Results and Analysis. As expected, velocity increased linearly. However, episodes were consistently truncated after approximately ten steps. This outcome validates two critical framework components: (i) Physics Integration: The smooth velocity trajectory confirms accurate symbolic-to-numeric translation of the underlying ODEs. (ii) Safety Monitor Fidelity: The rapid truncation demonstrates correct evaluation of the quadratic stopping distance $\text{stopDist}(v) = \frac{v^2}{2b}$. As v increased, the required safety buffer exceeded the available separation, correctly triggering a formal violation. The monitor's reaction aligns precisely with the mathematical bounds defined in the source dL model, as illustrated in Fig. 1.

5.4 Test 3: Learning to Meet Liveness and Deadlines

The final experiment evaluated the framework's ability to handle complex temporal logic, specifically [11, Theorem 14: Reach waypoint with deadline], which

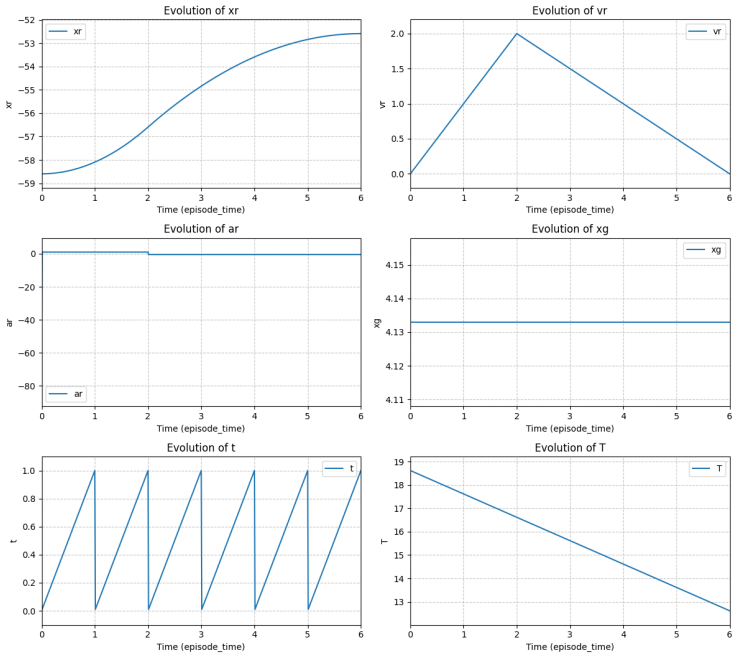

Fig. 2. Agent state evolution in Test 3 exhibiting an emergent triangular velocity profile

introduces a goal region (G_Δ) and a global time bound (T). While the full automation of this environment is the ultimate goal, for this specific test, the mapping from dL specifications to the Gymnasium reward function involved manual refinement to account for the current development state of Gymnasium dL. Despite this manual bridge, the reward signals themselves remain strictly grounded in the formal semantics of the dL annotations.

Emergence of Optimal Control. As shown in Fig. 2, the agent autonomously identified a velocity profile indicative of optimal control. Although the theoretical profile is trapezoidal, it appears as a triangular profile in this context because the distance to the waypoint and the stringent deadline T did not provide sufficient room for a constant-velocity cruise phase. The resulting policy consists of two symmetric phases: maximal acceleration followed by precise deceleration to stop within G_Δ, effectively finding the degenerate trapezoidal solution.

Deadline Management. The agent demonstrated strict temporal compliance by efficiently allocating its acceleration to meet the time bound T while maintaining the braking safety buffer. Training progress, as illustrated in Fig. 3, indicates that the agent sustained a high success rate throughout the process, reaching a peak of approximately 75%. The observed oscillations in the moving average suggest ongoing exploration of the narrow boundary between maximal velocity and the

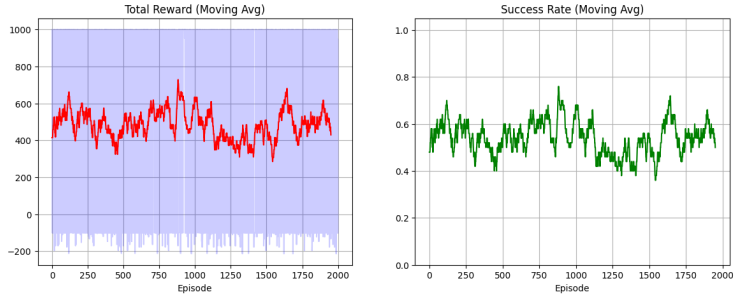

Fig. 3. Training progress and deadline compliance for Test 3.

quadratic braking constraint. These results confirm that dL-derived signals provide a sufficiently dense reward landscape to maintain high-performance policies that consistently meet temporal requirements and manage the safety-liveness trade-off without traditional manual reward engineering.

5.5 Evaluation Summary

The experimental results from the three test scenarios comprehensively address the initial research questions and illustrate the efficacy of Gymnasium dL.

Q1: Safety Encoding. Test 1 demonstrates that Gymnasium dL achieves high semantic fidelity in translating dL invariants into termination signals. Identification of the safe-lock state indicates that the `isSafe` predicate was interpreted as a strict boundary. The 100% safety rate in Test 1 confirms that formal safety requirements are preserved as primary constraints within the reinforcement learning environment.

Q2: Dynamics and Monitor Predictability. Test 2 confirms that the synthesized environments exhibit mathematical predictability. The stress-test results (Figure 1) show that the interaction between the `scipy`-based ODE solver and the safety monitors (stopDist) aligns with the symbolic dL specification. This result validates that Gymnasium dL does not introduce simulation gap errors during the conversion from continuous-time logic to discrete-time execution.

Q3: Autonomous Policy Discovery. Test 3 yields the most significant result, demonstrating the autonomous emergence of optimal velocity control. The agent balances maximal acceleration with braking constraints, resulting in the triangular profile shown in Fig. 2. This outcome confirms that a RL agent can navigate the complex trade-off between liveness (reaching a waypoint) and safety (braking distance) using only rewards derived from dL annotations. The 68% success rate, achieved without manual reward engineering, demonstrates that formal specifications provide a sufficiently dense signal for learning deadline-aware policies.

Table 1. Quantitative comparison across experimental tests.

Experiment	Strategy Observed	Safety	Liveness	Success Rate
Test 1	Static (Stay Still)	Absolute	None	100% (Trivial)*
Test 2	Aggressive (Forced Accel)	Failed	High	0% (Violation)
Test 3	**Optimal (Trapezoidal)**	**Verified**	**Timed Success**	**68%**

*Note: Test 1 is "successful" only in terms of safety; it fails all progress goals.

In summary, the evaluation confirms that Gymnasium dL bridges the gap between formal verification and reinforcement learning. It transforms abstract dL models into concrete training environments, enabling the development of agents that are grounded in formal logic.

6 Conclusion

This paper presented Gymnasium dL, a framework developed to address the semantic gap between formal verification and reinforcement learning. By designating the dL abstract syntax tree as the definitive source for environment synthesis, high-fidelity physics simulators, safety monitors, and reward structures can be derived directly from a verified formal model. This approach aligns the training objectives of reinforcement learning agents with safety invariants established through formal logic, thereby supporting effective reward engineering and reducing the risk of specification drift.

The evaluation demonstrates the effectiveness of this procedure. Results indicate that Gymnasium dL accurately encodes safety invariants as strict episode boundaries, thereby preventing agents from violating established safety envelopes. Additionally, the emergence of optimal, deadline-aware control strategies, such as the trapezoidal velocity profile, illustrates that such formal specifications offer a sufficiently informative signal for learning complex, high-performance behaviors. By anchoring agent exploration in the environment defined by dL, Gymnasium dL provides a foundation for developing neuro-symbolic systems in an iterative and provably safe way.

As Gymnasium dL remains under active development, future work will address three primary areas. First, efforts will focus on supporting a broader range of hybrid program shapes. Second, we plan to generate more idiomatic Gymnasium environments to support manual fine-tuning and facilitate long-term code maintenance. Finally, further automation of the mapping from logical predicates to reward signals is planned, with the goal of making the transition from formal KeYmaera X proofs to trainable reinforcement learning environments increasingly seamless and robust.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF2427581.

References

1. Alshiekh, M., Bloem, R., Ehlers, R., Hennig, O., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (2018)
2. Anderson, G., Verma, A., Dillig, I., Chaudhuri, S.: REVEL: Partially observed real-time verification of reinforcement learning policies. In: Proceedings of the 37th International Conference on Machine Learning (ICML) (2020)
3. Camacho, A., Icarte, R.T., Klassen, R.A., Valenzano, R., McIlraith, S.A.: LTL and reward machines: Learning to satisfy complex LTL formulas via reinforcement learning. In: Proceedings of the 3rd International Workshop on Formal Methods for ML-Enabled Autonomous Systems (2019)
4. Dalal, G., Dvijotham, K., Vecerik, M., Hester, T., Paduraru, C., Tassa, Y.: Safe exploration in continuous action spaces. In: arXiv preprint arXiv:1801.08757 (2018)
5. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
6. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2019)
7. Fulton, N., Mitsch, S., Quesel, J., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9195, pp. 527–538. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_36
8. Icarte, R.T., Klassen, R.A., Valenzano, R., McIlraith, S.A.: Using reward machines for high-level task specification and learning in reinforcement learning. In: Proceedings of the 35th International Conference on Machine Learning (ICML) (2018)
9. Ivanov, R., Carpenter, T.J., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig 2.0: Verification of neural network controllers using taylor model preconditioning. 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. 249–262. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_11
10. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
11. Mitsch, S., Ghorbal, K., Vogelbacher, D., Platzer, A.: Formal verification of obstacle avoidance and navigation of ground robots. Int. J. Robotics Res. **36**(12), 1312–1340 (2017). https://doi.org/10.1177/0278364917733549
12. Mitsch, S., Patel, I., Kannan, H.H.S., Jin, X., Zhan, B., Wang, S.: ARCH-COMP25 category report: Hybrid systems theorem proving. In: Frehse, G., Althoff, M. (eds.) Proceedings of 12th Int. Workshop on Applied Verification for Continuous and Hybrid Systems. EPiC Series in Computing, vol. 108, pp. 152–168. EasyChair (2025). https://doi.org/10.29007/pd1w
13. Mitsch, S., Platzer, A.: ModelPlex: verified runtime validation of verified cyber-physical system models. Formal Methods Syst. Des. **49**(1-2), 33–74 (2016). https://doi.org/10.1007/S10703-016-0241-Z

14. Paszke, A., et al.: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems (2019)
15. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. J. Autom. Reason. **59**(2), 219–265 (2017). https://doi.org/10.1007/S10817-016-9385-1
16. Qian, M., Mitsch, S.: Reward shaping from hybrid systems models in reinforcement learning. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May 16-18, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13903, pp. 122–139. Springer (2023). https://doi.org/10.1007/978-3-031-33170-1_8
17. Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N.: Stable-baselines3: Reliable reinforcement learning implementations. Journal of Machine Learning Research **22**(268), 1–8 (2021)
18. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
19. Strauss, M., Mitsch, S.: Slow down, move over: A case study in formal verification, refinement, and testing of the responsibility-sensitive safety model for self-driving cars. In: Prevosto, V., Seceleanu, C. (eds.) Tests and Proofs - 17th International Conference, TAP 2023, Leicester, UK, July 18-19, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14066, pp. 149–167. Springer (2023). https://doi.org/10.1007/978-3-031-38828-6_9
20. Teuber, S., Mitsch, S., Platzer, A.: Provably safe neural network controllers via differential dynamic logic. In: Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J.M., Zhang, C. (eds.) Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024 (2024)
21. Virtanen, P., et al.: SciPy 1.0: Fundamental algorithms for scientific computing in python. Nature Methods **17**, 261–272 (2020)

A KeYmaera X ASCII Syntax by Example

The KeYmaera X ASCII syntax is illustrated in the example below.

```
ArchiveEntry "Benchmark Example 1"

Definitions                          /* definitions cannot change their value */
  import kyx.math.abs;               /* import absolute value function */
  Real A = 5;                        /* real-valued maximum acceleration defined to
       ↪ be 5 */
  Real b;                            /* real-valued braking, undefined so unknown
       ↪ value */
  Bool geq(Real x, Real y) <-> x>=y; /* predicate geq defined to be formula x>=y */
  HP drive ::= {                     /* program drive defined to choose either */
       ?v<=5; a:=A;                  /* maximum acceleration if slow enough */
    ++ a:=-b;                        /* or braking, nondeterministically */
  };
End.

ProgramVariables              /* program variables may change their value over time */
  Real x;                     /* real-valued position */
  Real v;                     /* real-valued velocity */
  Real a;                     /* current acceleration chosen by controller */
End.

Problem                       /* conjecture in differential dynamic logic */
```

```
  v>=0 & b>0                  /* initial condition */
  ->                          /* implies */
  [                           /* all runs of this hybrid program */
    {                         /* braces {} group programs */
      drive;                  /* expand program drive here as defined above */
      { x'=v, v'=a & v>=0 }   /* differential equation system */
    }* @invariant(v>=0)       /* loop repeats, with @invariant contract */
  ] v>=0                      /* safety/postcondition after hybrid program */
End.

End.
```

B KeYmaera X Model Files

The following listing provides the full formal specification for the waypoint reachability task with a deadline (Test 3). This model includes the hybrid programs for control and dynamics, the loop invariant, and the safety/liveness properties.

Listing 1.1. KeYmaera X Model: Reach waypoint with deadline

```
Theorem "IJRR17/Theorem 14: Reach waypoint with deadline"

/*
 *      Robot must stop within distance delta at goal.
 *
 *  Robot
 *      - must stop within distance delta of goal
 *  - can only drive straight and forward
 *  - ensures progress towards goal
 *
 *  Liveness property:
 *  - Robot can stop at goal
 *
 */

Definitions
    // @const: bound=[0.1:0.5]; def="time limit for control decisions"
    Real ep;      /* time limit for control decisions */
    // @const: bound=[0.5:5.0]; def="minimum braking capability of the
        ↪ robot"
    Real b;       /* minimum braking capability of the robot */
    // @const: bound=[1.0:10.0]; def="maximum acceleration"
    Real A;       /* maximum acceleration -b <= a <= A */
    // @const: bound=[1.0:10.0]; def="goal area size"
    Real GDelta;         /* goal area size */
    // @const: bound=[5.0:30.0]; def="maximum robot velocity"
    Real Vmax;    /* robot cannot go faster than this */

    Real waypointStartDist(Real xg) = xg-GDelta;
    Real waypointEndDist(Real xg)   = xg+GDelta;

    Real minV = A*ep;
    Real maxTravelTime(Real g) = waypointStartDist(g)/minV;
    Real stopTime(Real v)      = v/b;
    Real speedUpTime(Real v)   = ep-v/A;

    Real stopDist(Real v) = v^2/(2*b);
    Real accComp(Real v)  = ( (A/b + 1)*(A/2*ep^2 + ep*v) );

    Bool bounds() <-> (
        A > 0                               /* Working engine */
        & b > 0                             /* Working brakes */
        & ep > 0                            /* Controller reaction time */
```

```
43          & Vmax >= 2*A*ep
44          & GDelta > Vmax*ep + Vmax^2/(2*b) /* waypoint is large enough
                ↪ that robot can start driving and still stop inside */
45      );
46
47      Bool initialState(Real xr, Real vr, Real xg, Real T) <-> (
48          vr = 0
49          & xr < waypointStartDist(xg)
50          & T > ep + maxTravelTime(xg-xr) + ep + stopTime(Vmax)
51      );
52
53      Bool assumptions(Real xr, Real vr, Real xg, Real T) <->
54          bounds() & initialState(xr, vr, xg, T);
55
56      // @safety: penalty=-1000, on_failure=truncate:status=failure
57      Bool loopinv(Real xr, Real vr, Real xg, Real T) <-> (
58          0 <= vr & vr <= Vmax & xr + stopDist(vr) < waypointEndDist(xg)
59                  & (waypointStartDist(xg) < xr -> (vr = 0 | T >=
                        ↪ stopTime(vr)))
60                  & (xr <= waypointStartDist(xg) ->
61                      (vr >= minV & T > maxTravelTime(xg-xr) + ep +
                            ↪ stopTime(Vmax))
62                      | (vr <= minV & T > speedUpTime(vr) +
                            ↪ maxTravelTime(xg-xr) + ep + stopTime(Vmax)))
63      );
64
65      HP controlFunc ::= {
66          // Choice 1: If in the goal area, decide to brake
67          { ? (xr > waypointStartDist(xg));  ar := -b; }
68          ++
69          // Choice 2: If in the goal area AND already stopped, decide to
                ↪ stay stopped
70          { ? (xr > waypointStartDist(xg) & vr = 0);  ar := 0; }
71          ++
72          // Choice 3: If outside the goal area AND can accelerate,
                ↪ decide to accelerate
73          { ? (xr <= waypointStartDist(xg) & (xr + stopDist(vr) +
                ↪ accComp(vr) < waypointEndDist(xg) & vr+A*ep <= Vmax));
                ↪ ar := A; }
74          ++
75          // Choice 4: If outside the goal area AND cannot accelerate
                ↪ (must coast), decide to coast
76          { ? (xr <= waypointStartDist(xg) & !(xr + stopDist(vr) +
                ↪ accComp(vr) < waypointEndDist(xg) & vr+A*ep <= Vmax));
                ↪ ar := 0; }
77      };
78
79      HP stepFunc ::= {xr' = vr, vr' = ar, t' = 1, T'=-1 & t <= ep & vr
            ↪ >= 0};
80
81      HP dwwpdl ::= {
82          {       { controlFunc; t := 0; }
83              { stepFunc; }
84          }*@invariant(loopinv(xr, vr, xg, T))
85      };
86
87      // @progress: reward=1000, on_success=terminate:status=success,
            ↪ on_failure=terminate:status=failure
88      Bool ensure(Real xr, Real vr, Real xg) <-> (
89          waypointStartDist(xg) < xr & vr = 0
90      );
91
92  End.
93
94  ProgramVariables
95      // @var: bound=[0:100], def="robot position: x"
96      Real xr;
97      // @var: bound=[0:20], def="robot translational velocity"
```

```
    Real vr;
    // @var: bound=[-10:10], def="robot translational acceleration"
    Real ar;
    // @var: bound=[0:100], def="goal position: x"
    Real xg;
    // @var: bound=[0:1], def="time elapsed in current control step"
    Real t;
    // @var: bound=[0:1000], def="global time remaining until deadline"
    Real T;
End.

Problem
    assumptions(xr, vr, xg, T)
        -> [ dwwpdl; ](xr < waypointEndDist(xg) & (T <= 0 -> ensure(xr,
            ↪ vr, xg)))
        & < dwwpdl; > ensure(xr, vr, xg)
End.
```

C Agent Configurations

C.1 Test 1 and 2: Dynamics, Safety Invariants, and Safety Monitors

For Test 1 and Test 2, an agent based on the Stable Baselines3 (SB3) library was employed to leverage its robust and optimized Proximal Policy Optimization (PPO) implementation.

- **Environment:** The environment for these tests presented a `Dict` action space, comprising both continuous assignment actions (e.g., `assign_r`, `assign_w`, `assign_xo`, `assign_yo`) and a discrete choice action (`choice_ChoiceLevel_0`).
- **Action Space Transformation:** To interface with SB3's PPO, which typically expects `Box` or `Discrete` action spaces, a custom `FlattenActionDictWrapper` was implemented. This wrapper converted the `Dict` action space into a single `Box` action space of shape $(5,)$. The first four elements corresponded to the continuous assignment actions, while the fifth element represented the discrete choice. This discrete value was rounded to the nearest integer and clipped to its valid range before being passed to the environment.
- **Training Algorithm:** Proximal Policy Optimization (PPO) from Stable Baselines3 was used. The policy network was an `MlpPolicy`, suitable for `Box` observation and action spaces.
- **Learning Rate:** A constant learning rate was employed for the PPO optimizer.
- **Training Duration:** The agent was trained for a total of $10,000$ timesteps.

C.2 Test 3: Learning to Meet Liveness and Temporal Constraints

Test 3 utilized a more advanced, custom-implemented Proximal Policy Optimization (PPO) agent in PyTorch to explore sophisticated control strategies and fine-grained reward shaping.

- **Environment:** The environment for this test exposed a purely discrete action space, representing the selection of different control branches within the d$\mathcal{L}$ program.

- **Observation Space:** The raw 8-dimensional observation from the environment was augmented with 6 derived features, such as distances to waypoints, time remaining in ODE steps, and velocity flags. This resulted in a 14-dimensional observation vector, which was robustly normalized and clipped to a range of $[-1.5, 1.5]$ before being fed to the policy network. **Training Algorithm:** A custom PPO implementation was developed to incorporate Action Masking. Logit-level masking, derived from dL guards, mathematically restricted the agent to logically valid actions, thereby ensuring exploration remained within the safe control envelope.
- **Policy Network:** The `ActorCritic` network featured a shared backbone with increased capacity (three linear layers with 512, 256, and 256 neurons, respectively, using `Tanh` activations), followed by separate actor and critic heads.
- **Action Masking:** A key component was the dynamic action masking mechanism, which synchronized the agent's discrete choices with the *tests* in the d$\mathcal{L}$ hybrid program. At each decision epoch, the environment's parser extracted the formal guards corresponding to each choice branch in the control function. These guards were evaluated against the current state vector (ν), which substitutes state variables into the logical formulas. The resulting boolean vector (the *action mask*) was provided to the PPO policy, which imposed a logit-level penalty on logically invalid branches. This process ensured that the agent explored only those actions for which the d$\mathcal{L}$ preconditions were satisfied, thereby enforcing the hybrid program's control structure as a strict constraint during both training and inference.
- **Reward Shaping:** In addition to the base rewards from `@safety` and `@progress` annotations, a multi-component reward shaping scheme was applied. This included dense progress rewards based on changes in distance to the waypoint, penalties for high velocity near the goal, and a small penalty for time progression, alongside explicit bonuses/penalties for successful goal achievement or invariant violations.
- **Learning Rate:** The learning rate for the Adam optimizer was linearly annealed from an initial 1×10^{-4} down to 5×10^{-6} over the course of training.
- **Duration of training:** The agent was trained for $10,000$ episodes, with updates occurring every 4096 environment step.