Clarity: From Formal System Designs to Verified RL Controllers

Austin O'Quinn
Ohio State University
Columbus, USA
oquinn.18@osu.edu

Conor Snedeker
Ohio State University
Columbus, USA
snedeker.31@osu.edu

Max Taylor
Boise State University
Boise, USA
maxtaylor@boisestate.edu

Lance Joneckis
Idaho National Lab
Idaho Falls, USA
lance.joneckis@inl.gov

Christopher Stewart
Ohio State University
Columbus, USA
stewart.962@osu.edu

Abstract

Safety-critical cyber-physical systems increasingly use reinforcement learning-based controllers, yet existing engineering methodologies cannot assure these systems' safety. Assuring safety requires connecting three concerns that are currently addressed in isolation: the system's design, the reinforcement learning pipeline, and runtime verification. This paper introduces CLARITY, a system that ingests designs written in SysML v2 and, through novel translational semantics to the nuXmv model checker, verifies safety requirements, synthesizes a reinforcement learning pipeline, and generates a runtime monitor. CLARITY enables fully traceable safety assurance arguments for reinforcement learning-based systems, closing the gap between system design, controller synthesis, and runtime assurance. We validate CLARITY on three CPS benchmarks of increasing complexity, demonstrating verified safety requirements, effective learned controllers, and lightweight runtime monitoring from a single SysML v2 specification.

CCS Concepts

• **Software and its engineering** → **Formal software verification**; • **Computing methodologies** → **Reinforcement learning**.

Keywords

cyber-physical systems, reinforcement learning, model checking, SysML v2, runtime monitoring, controller synthesis

ACM Reference Format:

Austin O'Quinn, Conor Snedeker, Max Taylor, Lance Joneckis, and Christopher Stewart. 2026. Clarity: From Formal System Designs to Verified RL Controllers. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834429

1 Introduction

Reinforcement learning (RL) is increasingly used to derive control policies for cyber-physical systems (CPSs), but providing assurance that these systems behave safely despite the presence of a black-box, RL-based controller is challenging. CPSs where RL-based controllers have been proposed include nuclear reactors [11, 26–28, 32], surgical robotics [25], and satellites [7]. These systems adopt RL when classical control strategies struggle with the control task. However, while classical control strategies have well-understood safety assurance arguments, RL-based control strategies do not. To address this problem, this paper introduces CLARITY: **C**ontroller **L**earning with **A**utomated **R**equirements, **I**nvariant checking, and **T**raceabilit**Y**—a system that creates an RL pipeline from a formal description of a CPS's design. Crucially, that same description enables CLARITY to provide assurance that the system meets its safety requirements.

RL-based controllers are challenging for classical CPS safety assurance arguments to handle. Safety-critical CPS development standards like ISO 26262 [14] and IEC 61508 [13] require evidence that systems meet their safety requirements. One widely accepted technique for providing this evidence is model checking. But model checking breaks down when systems contain RL-based controllers because the controller is inscrutable to the model checker. To re-enable model checking, engineers must create an abstraction that stands in for the black-box controller by manually writing a contract, e.g., [3, 15, 29]. But that contract's relationship with the system's requirements and the RL pipeline is tenuous—how does an engineer provide assurance that the controller's contract is sound?

Today, this assurance is piecemeal. Safety assurance arguments for controllers appearing in system designs are constructed using tools including Gamma, MP-firebird, and Imandra [9, 17, 23]. These tools accept a system design, written in the SysML v2 language [19], and formally verify that its safety requirements hold. Then, from the same system design, engineers manually construct an RL pipeline. Finally, runtime monitoring approaches like Aegis, the black-box simplex architecture, and barrier functions provide assurance that neural components maintain safety invariants [16, 30, 34].

However, this approach introduces three blind spots in the engineering methodology of RL-based CPSs: (1) design verification tools are unaware of the needs of the RL pipeline, (2) the RL pipeline is unaware of the system design, and (3) runtime monitoring is unaware of both. The first blind spot arises because SysML v2-based design verification tools lack support for the contract-based reasoning needed to verify RL-based systems. The second blind spot emerges from the disconnect between the system design and RL training process. This disconnect means that the actual system design can disagree with the RL process's assumed design, causing the resulting RL-based controller to misbehave at run time. This is because existing tools lack support for a large subset of SysML

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834429

v2, including executable *actions* used to model environment dynamics, preventing its adoption in the RL pipeline. Finally, runtime verification approaches require engineers to re-encode information about their RL-based controller in whichever formalism the runtime verification approach uses.

Removing these blind spots is challenging. Cluing design verification tools into the needs of the RL pipeline means providing support for an extended subset of SysML v2 to capture environment dynamics. Additionally, we must provide system designers with mechanisms that support specifying which components are obtained from RL, as well as their contracts. To remove the blind spot of the RL pipeline, we must ensure that its environment model and reward feedback agree with the system design and requirements. Finally, we need assurance that the formalisms provided to the runtime monitoring component matches how the RL-based controller was trained.

CLARITY removes these blind spots. Its main idea is to connect design verification, RL training, and runtime monitoring through a single SysML v2 model, so that each concern is grounded in the system design. To remove the blind spot of design verification tools, CLARITY introduces novel translational semantics for an extended subset of SysML v2 grounded in the nuXmv language [6]. In contrast to prior work, our translational semantics fully supports environmental dynamics and contract-based reasoning for RL-based controllers. We also provide system designers with a metadata library for specifying the components obtained from RL and their contracts. To remove the RL pipeline's blind spot, CLARITY customizes the RL training procedure from the SysML v2 design. This is accomplished by introducing a function that masks safety-violating actions known as a *shield* [4, 5, 20] derived from SysML v2 safety specifications. The shield simplifies learning and is deployed with the learning model for safe operation. Our runtime monitor is constructed by the same principle as our shield so that safety in training and deployment coincide.

CLARITY enables system designers to describe a CPS with an RL-based controller in a SysML v2 design. This automates the verification and RL activities associated with the engineering of CPSs with RL-based components. It also provides a fully traceable assurance argument, as each artifact ultimately derives from a single source of truth—the SysML v2 design. We validate CLARITY on three RL-based control systems and provide a detailed evaluation report.

To summarize, we present a system that, for the first time, integrates the engineering of RL-based CPS controllers and their safety assurance into a single automated pipeline driven by a SysML v2 design. We:

(1) **Provide translational semantics for an extended subset of SysML v2**, grounded in the nuXmv language, that supports environment dynamics modeled as *actions*. We also provide annotations enabling contract-based reasoning for RL-based controllers;
(2) **Automatically synthesize an RL pipeline**—including its environment model, safety constraints, and initial condition sampling—directly from the SysML v2 design, eliminating inconsistencies caused by manual translation between the system specification and the training process;

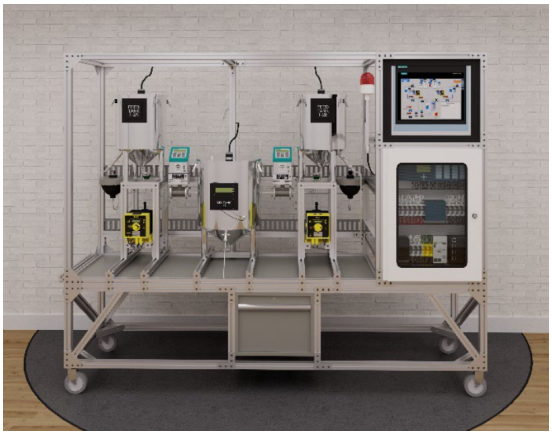

Figure 1: The chemical mixing skid, consisting of two ingredient tanks connected to a mixing tank. Each ingredient tank has its own pump and a valve to control ingredient flow.

```
#Prohibition
requirement def 'No Dead Heading' {
  subject s : MixingSystem;
  require constraint {
   (¬s.valve1.isOpen ⟹ ¬s.pump1.isRunning) ∧
   (¬s.valve2.isOpen ⟹ ¬s.pump2.isRunning)
  }
}
```

Listing 1: A SysML v2 requirement expressing that the controller must not dead head the pumps.

(3) **Generate a runtime monitor from the system requirements**, ensuring that what is monitored at deployment is exactly what was verified at design time;
(4) **Validate CLARITY on three CPS benchmarks**, demonstrating that the approach produces safe controllers with a fully traceable assurance argument from a single SysML v2 design.

2 CLARITY in a Nutshell

Figure 1 shows a safety-critical CPS. This system has a human-machine interface (HMI) that enables the user to configure the amount of each of the two ingredients to transfer to the mixing tank. The transfer is governed by a controller that sends Modbus—a CPS communication protocol—commands to the system's sensors and actuators. Our goal is to synthesize an RL-based controller for this system from its SysML v2 design, while providing assurance that the SysML v2 design meets safety requirements.

There are two important safety requirements:

(1) **No Dead Heading.** Dead heading occurs when a pump operates while its valve is closed. The closed valve causes pressure to continuously increase in the system's tubing, leading to a catastrophic failure.
(2) **No Dry Running.** Dry running occurs when a pump operates without fluid in its ingredient tank. Fluid lubricates

```
part def Tank {
  port outlet, inlet : FluidPort;
  attribute currentLevelMl : ℤ;
  // Other attributes not shown.
  action step {
    in dt : ℝ;
    assign currentLevelMl := clamp(
      currentLevelMl +
      (inlet.flowRateMl - outlet.flowRateMl) *
      dt ); } }
```

Listing 2: The definition of a tank in the SysML v2 design of the chemical mixing skid. This definition is instantiated three times: once for each of the two ingredient tanks, and once for the mixing tank.

and cools the pump. Thus, dry running leads to the pump overheating.

Listing 1 shows the first requirement as it appears in the system's SysML v2 design. We discuss in the next section how CLARITY verifies our synthesized controller abides by its associated contract.

2.1 Design Verification

Verifying SysML v2 designs presents two challenges. First, to prove that safety requirements like the one in Listing 1 hold, we must represent the SysML v2 design in a language suitable for formal verification. We use the nuXmv language, the input formalism of the nuXmv model checker. This raises a natural question: *how do we provide nuXmv-based translational semantics for SysML v2?* Second, supporting RL-based controllers in SysML v2 designs requires new representational mechanisms: contract specifications for RL-based controllers, and a way to bring system requirements into the RL pipeline.

2.1.1 nuXmv-based Translational Semantics. Consider Listing 2, which shows the `Tank` part definition. Part definitions are instantiated to create specific parts. This definition, for example, is instantiated three times: once for each of the two ingredient tanks, and once for the mixing tank. Parts compose a system in SysML v2. We adopt the convention that a system's dynamics are computed by its `step` action (line 5 of Listing 2).

However, actions like `step` are not supported by existing SysML v2 translational semantics. This limitation goes beyond our convention for encoding dynamics. It is natural, for instance, to represent the sequence of steps controlling pumps and valves as an action—yet without **`action`** semantics, this representation is impossible.

How CLARITY Helps: Listing 3 shows the nuXmv code that CLARITY automatically generates from `Tank`'s `step` action. The presence of this action causes CLARITY to generate nuXmv code for each instantiation of the part definition. One such instantiation is `feederTank1`, the first ingredient tank. Because SysML v2 actions execute sequentially, we divide each state into a sequence of *scan phases*. The number of scan phases equals the number of statements in the longest action in the system. The first statement executes during scan phase 0, the second during scan phase 1, and so on. The action in Listing 2 contains a single statement, so it executes entirely during scan phase 0.

```
part def Controller {
  // ...
  #Neural action def Policy {
    in tank1VolumeMl : ℤ;
    in tank2VolumeMl : ℤ;
    in tank1TargetTransferMl : ℤ;
    in tank2TargetTransferMl : ℤ;
    in tank1OriginalMl : ℤ;
    in tank2OriginalMl : ℤ;
    out shouldOpenValve1 : 𝔹;
    out shouldOpenValve2 : 𝔹;
    out shouldTurnOnPump1 : 𝔹;
    out shouldTurnOnPump2 : 𝔹; }
  #NeuralRequirement
  requirement def NeuralControllerSoundness {
    subject p : Policy;
    require constraint {
      (p.shouldTurnOnPump1 ⟹
        p.shouldOpenValve1) ∧
      (p.shouldTurnOnPump2 ⟹
        p.shouldOpenValve2) ∧ ...
    } } }
```

Listing 4: Contract specification for an RL-based controller using CLARITY's metadata library. The `#Neural` and `#NeuralRequirement` annotations enable automated contract extraction for assume/guarantee reasoning.

Listing 3 captures these semantics. The `next` state of the `currentLevelMl` attribute of `feederTank1` depends on the current scan phase (line 3). If `scan_phase` is not 0, `currentLevelMl` remains unchanged. Otherwise, it updates according to the assignment in the `step` action (line 4).

```
next(feederTank1_currentLevelMl) :=
case
 scan_phase != 0 : feederTank1_currentLevelMl;
 TRUE : clamp(((feederTank1_currentLevelMl +
    ((feederTank1_inlet_flowRateMl -
    feederTank1_outlet_flowRateMl) * dt))));
esac;
```

Listing 3: Tank's `step` action encoded as nuXmv code.

2.1.2 Contract-based Reasoning. RL-based controllers are inscrutable to model checkers like nuXmv. To verify a system containing an RL-based controller, we need a coarse approximation of the controller's behavior; i.e., a contract that enables classical assume/guarantee reasoning. Today, these contracts are developed outside the system design, raising the possibility that an inaccurate contract causes verification to silently succeed on a flawed design.

How CLARITY Helps: CLARITY provides a SysML v2 metadata library that lets system designers specify contracts directly in their designs. Listing 4 illustrates its use. Line 3 defines an action named `Policy`, annotated `#Neural` to indicate that its behavior comes from reinforcement learning. The learned behavior must obey the contract on line 14, which CLARITY associates with `Policy` via the `#NeuralRequirement` annotation.

CLARITY translates these annotations into nuXmv. Listing 5 shows the generated code. Because the controller's behavior is unknown

```
1  ...
2  IVAR
3  controller_pc_shouldOpenValve1 : boolean;
4  controller_pc_shouldOpenValve2 : boolean;
5  controller_pc_shouldTurnOnPump1 : boolean;
6  controller_pc_shouldTurnOnPump2 : boolean;
7  ...
8  TRANS
9  (controller_pc_shouldTurnOnPump1 ->
10     controller_pc_shouldOpenValve1) &
11 (controller_pc_shouldTurnOnPump2 ->
12     controller_pc_shouldOpenValve2);
```

Listing 5: The contract in Listing 4's nuXmv code.

at verification time, its outputs are modeled as nuXmv `IVAR`s—nondeterministic inputs to the transition relation (lines 3–6). The `TRANS` constraint on line 8 then restricts these outputs to those permitted by the contract.

2.2 Automatic RL Pipelines

Our starting point for controller synthesis is standard proximal policy optimization (PPO) trained on the SysML v2-derived environment. This baseline fails for two reasons, each of which motivates an addition to the pipeline.

2.2.1 Contract-based Pre-training. Standard PPO fails because the contract dramatically restricts the feasible action space. A randomly initialized policy proposes contract-violating actions at a rate that grows exponentially with episode length. For the mixing system, this means at best $\left(\frac{9}{16}\right)^{37} \approx 10^{-10}$ probability of a violation-free episode, so episodes terminate before any useful reward signal is obtained. The policy needs some form of expert demonstration before temporal learning begins. In our automated setting, this demonstration must be derived mechanically from the system design, without manual engineering effort.

How CLARITY Helps: CLARITY affords a reliable, individuated labeling mechanism for pre-training that is native to this SysML-specified setting. The labels are trustworthy because they derive from the controller contract, which is itself a requirement for the pipeline's use. We determine the feasible set of observable states from the given contract specification and construct a labeling function from the `#Neural` and `#NeuralRequirement` annotation to fully generate the pre-training data.

2.2.2 Shield-guided PPO. Even with contract-aware pre-training, PPO's stochastic exploration can still propose actions that violate the contract. For example, the mixing plant controller operates four Boolean actuators, producing 16 possible actions, many of which violate safety requirements such as running a pump while its valve is closed. A safety-critical pipeline cannot tolerate even a single violation, whether during training or at deployment.

How CLARITY Helps: CLARITY compiles the controller's contract into a deterministic function called a *shield* that intercepts every proposed action at run time. Our mixing plant model has a large contract that specifies its allowed behavior over much of its action space. Leveraging this, the final controller is constructed with a real-time verification process that stops any unsafe action specified via the controller contract. This effective removal of unwanted actions is extended to deployment as continual assurance of safe operation.

2.3 Runtime Monitoring

At deployment, the mixing skid's controller must not dead head or dry run—the same requirements verified at design time. Conventionally, ensuring this requires engineers to re-encode these properties in a runtime-specific formalism, separate from both the system design and the training process.

How CLARITY Helps: CLARITY reuses the shield from training (§3.2.2) as the runtime monitor: every action the controller proposes passes through the same safety check used during PPO fine-tuning. For the mixing skid, the contract that prevented dead heading and dry running during training is enforced identically in deployment. No separate runtime formalism is needed. §3.3 establishes why this per-action check, combined with design verification, is sufficient to guarantee safety without predictive reasoning.

3 Methodology

This section explains how CLARITY works. We proceed through its design verification, automatic RL pipeline generation, and runtime monitoring components. In our discussion on design verification, we present the subset of SysML v2 which we support (§3.1.1), its translational semantics (§3.1.2), and support for contract-based-reasoning (§3.1.3).

Figure 2 shows how CLARITY organizes these components into a single pipeline. The engineer supplies one SysML v2 artifact, which CLARITY reads along two axes: the system model, which captures parts, ports, and dynamics, and the requirements, which capture safety properties and task goals. The formal verification path extracts a nuXmv model from the system model using the translational semantics of §3.1.2, discharges each requirement by inductive verification, and returns a per-requirement proof. The RL training path interprets the same system model as an executable environment, pre-trains a policy from contract-derived labels, and fine-tunes the recurrent policy with shielded PPO to produce the trained policy. The runtime monitoring path compiles the requirements into a shield, which masks the policy's proposed actions during fine-tuning and is reused unchanged as the runtime monitor that checks the deployed policy at every step. Note that no arrow in the figure originates outside the SysML v2 specification: the nuXmv model, the training environment, the shield, and the monitor are all downstream of a single source of truth, which is what makes the resulting assurance argument traceable end to end.

3.1 Design Verification

3.1.1 SysML v2 Subset. Figure 3 shows the subset of SysML v2 that CLARITY supports. This subset is designed to capture CPSs with discrete controllers, continuous dynamics, and message-passing communication.

The core language elements are **`part def`**, **`port def`**, and **`item def`** declarations. Parts carry typed attributes; ports define communication interfaces; and items model message types, with single inheritance via `:>`. A system-level part definition instantiates these components with **`part`**, wires them together with **`flow`** for dataflow

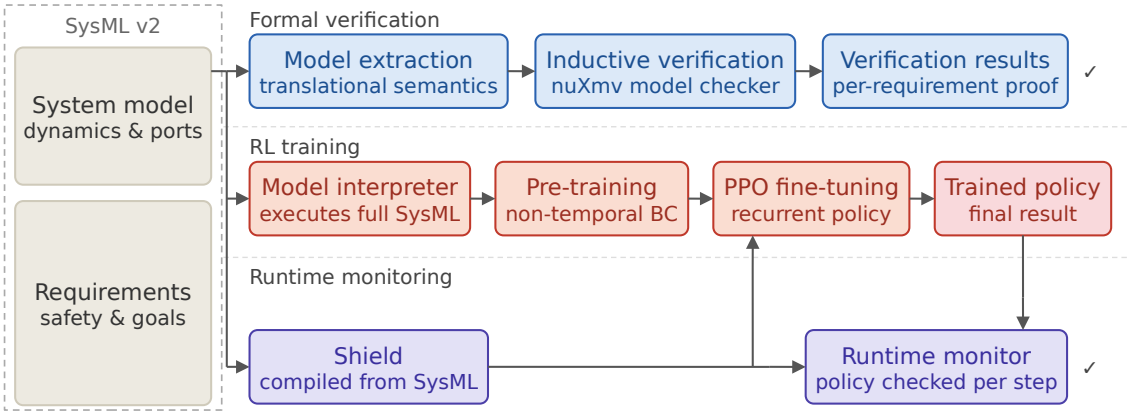

Figure 2: Full CLARITY pipeline diagram from SysML v2 specification to runtime monitoring and control system deployment.

Top-level structure

Model ::= **package** *Id* { *Def** }
Def ::= *ItemDef* | *PortDef* | *PartDef*
| **metadata def** *Id* ;

Item and port definitions

ItemDef ::= **item def** *Id* [**:>** *Id*]
[{ *AttrDecl** }] ;
PortDef ::= **port def** *Id* { *PortMem** }
PortMem ::= [**in** | **out**] **item** *Id* : *Id* ;
| **attribute** *Id* : *Type* ;

Part definitions

PartDef ::= **part def** *Id* { *PartMem** }
PartMem ::= **attribute** *Id* : *Type* [= *Expr*] ;
| **port** *Id* : [~]*Id* ;
| **ref** *Id* : *Id* ;
| [*Ann*] **constraint** *Id* { *Expr* }
| [*Ann*] *ReqDef*
| **action** *Id* { *Stmt** }
| [*Ann*] **action def** *Id* { *Param** }
| **state def** *Id* { *SmMem** }
| **exhibit state** *Id* : *Id* ;
| **bind** *Path* = *Path* ;
| **part** *Id* : *Id* [{ *Override** }]
| **flow** [: *Id*] **from** *Path* **to** *Path* ;
| **connect** *Path* **to** *Path* ;
ReqDef ::= **requirement def** *Name* {
subject *Id* : *Id* ;
require constraint { *Expr* } }

State machines

SmMem ::= **state** *Id* ;
| **transition** *Id* **first** *Id*
[**accept** [*Id* :] *Id* [**via** *Path*]]
[**if** *Expr*]
[**do action** { *Stmt** }]
then *Id* ;

Action statements

Stmt ::= **in** *Id* : *Type* ;
| **item** *Id* : *Id* ;
| **assign** *Path* := *Expr* ;
| **send** *Id* **via** *Path* ;
| **accept** [*Id* :] *Id* **via** *Path* ;
| **attribute** *Id* : *Type* [= *Expr*] ;
| **if** *Expr* { *Stmt** } [**else** { *Stmt** }]
| **perform action** *Id* ;
| **action** *Id* : *Id* { *Binding** }
Binding ::= **in** *Id* := *Expr* ; | **out** *Id* := *Expr* ;
Param ::= **in** *Id* : *Type* ; | **out** *Id* : *Type* ;

Annotations

Ann ::= **#***Id*
Id ∈ { **Neural**, **Prohibition**, **Obligation**,
ScenarioInput, **ScenarioConstraint**,
NeuralRequirement, **Termination** }

Figure 3: Abstract syntax of the supported SysML v2 subset. X^* denotes zero or more repetitions; square brackets denote optional elements; keywords are in `bold monospace`; non-terminals are in *italics*. Expressions (*Expr*) support arithmetic (`+`, `-`, `*`, `/`), comparison (`==`, `>=`, `<=`, `>`, `<`), logical operators (`and`, `or`, `implies`, `not`), ternary conditionals, and dot-separated paths. Types are `Integer`, `Real`, and `Boolean`. Identifiers (*Id*) follow `[a-zA-Z_][a-zA-Z0-9_]*`; names (*Name*) may additionally be quoted strings.

and `connect` for command routing, and aliases port attributes with `bind`.

Behavior is specified through two complementary mechanisms: state machines and actions. State machines (`state def`) model reactive components. Each transition specifies a source state, an optional `if` expression called its *guard*, and a target state; the transition fires only when the machine is in the source state and the guard evaluates to true. Transitions may also `accept` messages arriving via ports and execute `do action` side effects, including `send`ing responses. Meanwhile, actions model continuous plant dynamics as discrete-time update equations. The `step` action, introduced in Listing 2, is the primary example: each `assign` statement computes the next value of a state variable from its current value, a timestep parameter, and current flow rates. Actions compose with `if`/`perform` for control logic and with sub-action calls for invoking `action def` blocks, including `#Neural`-annotated policy interfaces whose outputs are treated as nondeterministic during model checking. The `if` construct also serves as a guard within action bodies: statements nested inside an `if` only execute when its condition holds.

Declarative constructs round out our subset of SysML v2 semantics. Constraints express algebraic relationships such as flow-rate

Syntactic pattern	nuXmv section	Generated form
State variables		
PartMem: state def S { state s_1; … state s_n; … }	VAR	$v_S : \{s_1, \ldots, s_n\}$;
PartMem: attribute $x : \tau$, $x \in$ targets(*step* ∪ *do-action*)	VAR	$\hat{x} : [\![\tau]\!]$;
Step bodies containing N total statements	VAR	scan_phase : 0..$N-1$;
Input and frozen variables		
PartMem: #Neural action def D { … out $p : \tau$ }	IVAR	$\hat{p} : [\![\tau]\!]$;
PartMem: #ScenarioInput attribute $a : \tau$	FROZENVAR	$\hat{a} : [\![\tau]\!]$;
Definitions (combinational logic)		
PartMem: attribute $a : \tau = e$, $a \notin$ targets(*step* ∪ *do-action*)	DEFINE	$\hat{a} := [\![e]\!]$;
PartMem: constraint *Id* { x == e }	DEFINE	$\hat{x} := [\![e]\!]$;
PartMem: constraint *Id* { (c_1 implies x == e_1) and … }	DEFINE	$\hat{x}$:= case $[\![c_1]\!] : [\![e_1]\!]$; … TRUE : 0; esac;
PartMem: flow from $A.p$ to $B.q$	DEFINE	$\widehat{B.q.a} := \widehat{A.p.a}$; ∀ attr. a (fixed-point)
PartMem: bind $l = r$	DEFINE	$\hat{l} := \hat{r}$;
PartMem: connect $A.p$ to $B.q$; sends at phase k	DEFINE	$\widehat{trig_B} := (\phi{=}k \wedge [\![g]\!])$;
Initialization		
PartMem: attribute $x : \tau = v_0$, $x \in$ targets(*step*)	ASSIGN	init($\hat{x}$) := $[\![v_0]\!]$;
SmMem: first state s_0; in state def S	ASSIGN	init(v_S) := s_0;
PartMem: #ScenarioConstraint constraint *Id* { C }	INIT	$[\![C]\!]$;
Transition relation		
$Stmt_k$: assign $x := e$, guard g	ASSIGN	next($\hat{x}$) := case $\phi{=}k \wedge [\![g]\!] : [\![e]\!]$; TRUE : $\hat{x}$; esac;
$Stmt_k$: send m via p, guard g	ASSIGN	(side-effects of m guarded by $\phi{=}k \wedge [\![g]\!]$; triggers receiver)
SmMem: transition first s if g then s', phase k	ASSIGN	next(v_S) := case $\phi{=}k \wedge v_S{=}s \wedge [\![g]\!] : s'$; . . . esac;
$Stmt_k$: assign $y := e$ in do action of t	ASSIGN	next($\hat{y}$) := case $\phi{=}k \wedge$ *firing*(t) : $[\![e]\!]$; TRUE : $\hat{y}$; esac;
Phase counter	ASSIGN	next(ϕ) := case $\phi < N{-}1 : \phi + 1$; TRUE : 0; esac;
Verification		
PartMem: #NeuralRequirement *ReqDef* { R }	TRANS	$[\![R]\!]$;
PartMem: #Prohibition or #Obligation *ReqDef* { R }	INVARSPEC	$[\![R]\!]$

Figure 4: Translational semantics from the SysML v2 subset (Figure 3) to nuXmv SMV. The translator linearizes each step body into N statements, each assigned a unique phase $k \in \{0, \ldots, N-1\}$; scan_phase cycles through all N phases per scan. $\hat{x}$: flattened SMV name (e.g., feederTank1::currentLevelMl ↦ feederTank1_currentLevelMl); ϕ: scan_phase; $g = \top$ when no enclosing if; targets(k): attributes assigned in context k; *firing*(t) $= v_S = s \wedge [\![g]\!]$ for transition t with source s and guard g. $[\![\cdot]\!]$ translates expressions by operator substitution and types by: $[\![$Boolean$]\!]$ = boolean, $[\![$Real$]\!]$ = real, $[\![$Integer$]\!] = l\,..\,u$ (bounded).

equations and conditional invariants. Requirements with **subject** bindings state the safety properties to be verified. Metadata annotations—#Prohibition, #Obligation, #ScenarioInput, #NeuralRequirement, and others—classify these declarations for other components of CLARITY, including specification validation and our runtime monitor.

3.1.2 nuXmv-based Translational Semantics. Figure 4 defines the translational semantics from the SysML v2 subset to nuXmv. The translation maps each syntactic pattern in Figure 3 to a specific nuXmv language section—VAR, IVAR, DEFINE, ASSIGN, TRANS, or INVARSPEC—producing a complete nuXmv model from a SysML v2 design. The core language elements translate straightforwardly: part attributes become state variables, flow and bind declarations become combinational definitions, and state machines become enumerated variables with guarded case transitions.

The novelty in these semantics is in the action semantics. Existing SysML v2 formalization tools like Gamma handle statechart transitions and their effects but do not support **action** bodies—the constructs CPS engineers use to model plant dynamics and sequential control logic. Our translation linearizes each step body into a sequence of N statements, assigns each statement a unique phase k, and introduces a scan_phase variable that cycles through all N phases per scan. Each **assign** statement then becomes a next expression that fires only at its designated phase, leaving the variable unchanged otherwise. This mechanism generalizes to **if**/**else** branching, **send**/**accept** message passing, and sub-action invocations via **perform action**, giving the translation full coverage of the action fragment in Figure 3.

The remaining rules connect the behavioral semantics to verification and RL. #Prohibition and #Obligation requirements become INVARSPEC properties checked by nuXmv. #ScenarioInput attributes become FROZENVARs and #ScenarioConstraint constraints become INIT conditions, together defining the space of initial configurations for both verification and RL training.

3.1.3 Contract-based Reasoning. CLARITY supports contract-based reasoning over RL-based controllers. This reasoning requires engineers to use the following two featherweight annotations: #Neural and #NeuralRequirement. For #Neural-annotated actions, our translational semantics map the action's **out** parameters to nuXmv IVARs. IVARs are non-deterministic inputs to the system; they model external, uncontrollable state. These semantics align perfectly with black-box, RL-based controllers.

On the other hand, if all behaviors of an RL-based controller were to be allowed, nuXmv would fail to verify the system's design. To prevent this, engineers must create #NeuralRequirement-annotated

requirements—i.e., a coarse-grained contract specifying the invariants of the RL-based controller. These requirements map to nuXmv TRANS constraints in our semantics. A TRANS constraint restricts permissible combinations of IVAR values. Combined, our approach allows us to model RL-based controllers non-deterministically, while still specifying behavioral constraints.

3.2 Automatic RL Pipelines

Unlike a typical RL setting, we have access to a contract designed to constrain any action our neural network may take to complete a goal. This allows us to construct a function to reduce training times in our RL pipeline and guarantee compliance with safety specifications at run time. Our RL pipeline is a two-step process: we first construct a problem-dependent deterministic function called a *shield* [4, 5, 20] that rejects contract-violating actions. We then train a neural network that uses the shield as a decision mask to learn the time-dependent policy of the controller deployed by CLARITY.

Engineers provide their system's SysML v2 design to CLARITY, which processes the contract into learning problem specifications. The RL pipeline takes the specification as input to train a neural network controller that we call our *policy model*. The learning of our controller is modeled as a *partially observable Markov decision process* (POMDP) consisting of the *state space* S of n-dimensional real vectors, and the actuator *action space* A corresponding to the SysML `action` semantics. The observation space O consists of the $m \leq n$ dimensions of S visible to the agent, and for $(s, s') \in S \times S$ and action $a \in A$ the training reward function $R(s, a, s')$ with values in $[-1.0, 1.0]$. We denote by H the state space of the recurrent component our policy model with values $h = (h_1, \ldots, h_k)$ and a trainable value head $Q : H \to [-1.0, 1.0]$ to predict optimal future rewards.

The policy model is a function $\pi : O \times H \to p(A)$ realized as a neural network that maps pairs of observations and hidden states to discrete distributions over the action space A. Our pipeline is specified by the `#NeuralRequirement` annotation and space $A \times S \times O$ determined by the formal semantics of SysML v2, corresponding to the learnable controller. CLARITY passes this problem specification to our RL pipeline, which mechanically encodes a shield function with a decision process that constrains our policy model to contract-compliant actions.

Policy decisions are learned via a composite neural network architecture featuring a *multi-layer perceptron* (MLP) encoder and *gated recurrent-unit* (GRU) for learning temporal properties. The combination of this policy model with shield-based verification (Figure 5) constitutes the synthesized controller output by CLARITY. The shield is called at inference time by the policy model to verify its final decision, including in deployment. Training consists of two phases, detailed below.

3.2.1 Contract-based Pre-training. As described in §2.2.1, a randomly initialized policy cannot obtain reward signal because contract violations terminate episodes before temporal learning begins. Phase 1 eliminates this by using behavioral cloning [10, 31] from the contract-compliant labeling function to teach the policy the contract, reducing dependence on the shield. We derive expert knowledge via a contract-compliant function constructed like our shield, sampling an observation o and using the contract-compliant

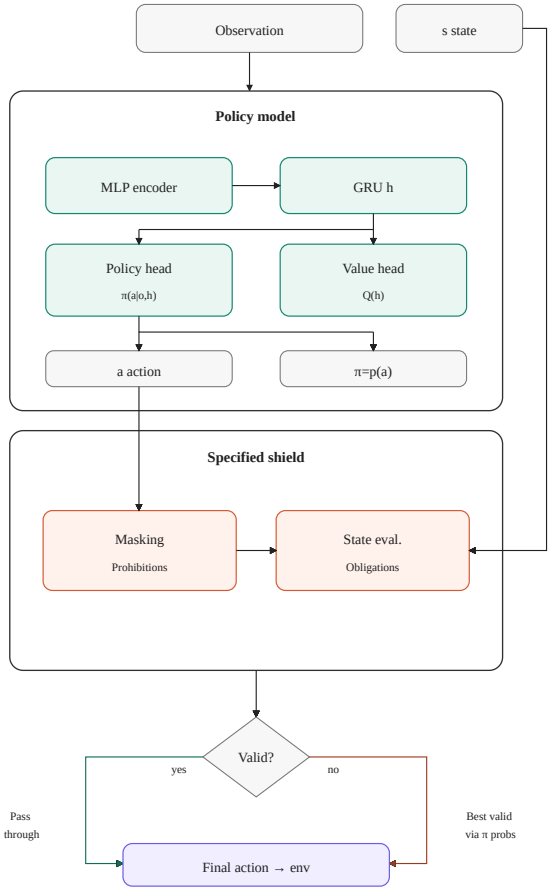

Figure 5: Composite decision model consisting of specified shield and trainable policy model. During each update batch of PPO and at final deployment, the composite models' internal state consists of the functions π, Q, p, and M computing values $(s, a) \in S \times A$ at various stages of the inference pipeline.

function to label it an action a. The constructed pairs (o, a) are batched to pre-train the policy model via supervised gradient descent as demonstration. We pre-train with 2,000 labeled data points $(o, a) \in O \times A$.

3.2.2 Shield-guided PPO. A pre-trained policy can propose contract-violating actions during PPO's stochastic exploration. To prevent this, we construct a shield, a deterministic post-processing that maps A to the set of contract-compliant actions U_s at state s. Encoding of the contract reduces task complexity for the policy model and allows our pipeline to avoid the difficulties of explicit reward-shaping. The construction of the specified shield is a one-step process ending with a function we query to determine the contract-compliance of an action. We collect the output constraints from the RL-based controller's contract and combine them into a single conjunction, $C : A \to \mathbb{B}$. This conjunction checks actions

$a \in A$ as truth assignments against the policy for contract compliance. The set of all contract-compliant actions is $U = \{a \in A | C(a)\}$, and the set of all non-compliant actions is $V = A \setminus U$. For state-dependent `#NeuralRequirement` attributes, the compliant set is state-dependent $U = U_s$. Actions that are non-compliant in every state—i.e., $\bigcap_{s \in S} V_s$—are *dead actions*, permanently masked before training begins. Our external shield program constructs a *mask* function $M(a, p(A), s)$ mapping $(a, p(A), s)$ to compliant actions via

$$M(a, p(A), s) = \begin{cases} a^* \in \text{argmax}_{a' \in U} p(a') & \text{if } a \in V_s \\ a & \text{else} \end{cases}.$$

Phase 2 is a temporal learning regime of 2,000 PPO episodes implemented via `PyTorch`, buffered into 100 episode batches for each update step. Phase 2 learns temporal properties and efficient strategy, such as coordinating pump timing to avoid overshoot. Since the policy model $\pi(a|o, h)$ outputs a ranked probit list, it learns the effective action ranking given an observable o safety-verified by M. The final decision of the combined controller is $a_{\text{final}} = M(a, p(A), s)$ and the environment rewards the policy model for this decision, which trains the policy model to optimize rewards via contract-compliant training feedback. We use a penalized sparse reward function

$$R(s, a, s') = \begin{cases} 1.0 & \text{if done} = \text{TRUE and } a \in U \\ -0.01 & \text{else} \end{cases}$$

to incentivize the completion of episodes. Every 100 episodes the composite model is evaluated on 100 episodes and the model with the least evaluation safety violations is saved, or the model with highest accuracy in case of ties. The controller's policy model is both trained and deployed with the same specified shield.

3.3 Runtime Monitoring

At deployment, the shield compiled in §3.2.2 serves as the runtime monitor: every action the trained policy proposes passes through M before reaching the environment. This non-temporal check—evaluating the contract against the current state and proposed action, with no lookahead—is sufficient because design verification (§3.1) has already established that the system's safety requirements hold for *all* contract-compliant controller behaviors. Each `#Prohibition` and `#Obligation` is verified as an `INVARSPEC` over a transition system whose controller outputs are nondeterministic subject to the `TRANS` constraint derived from the annotation of `#NeuralRequirement`. The shield guarantees $a_{\text{final}} \in U_s$ at every step, so the deployed controller never exits this verified envelope.

4 Evaluation

We evaluate CLARITY on three CPS benchmarks of increasing complexity, organized around six research questions:

- **RQ1** *How much manual effort does CLARITY eliminate?*
- **RQ2** *Can CLARITY verify safety requirements extracted from SysML v2 designs?*
- **RQ3** *Does the synthesized RL pipeline produce effective, safe controllers?*
- **RQ4** *Is the generated runtime monitor effective and efficient?*
- **RQ5** *What are the resource costs of running the pipeline?*

Table 1: Benchmark systems. *Dead actions* are structurally infeasible action combinations identified at compile time by the shield (3.2.2).

System	Safety Reqs	Action Space	Dead Actions	Scan Phases
Thermostat	3	4	1 (25%)	1
Cruise Control	5	4	1 (25%)	1
Chemical Mixing	4	16	7 (44%)	14
Total	12			

Table 2: Automation metrics. All artifacts are auto-extracted from a single SysML v2 model. The shared pipeline is reused across all benchmarks without modification.

System	SysML LOC (engineer-written)	nuXmv LOC (auto-extracted)	Ratio
Thermostat	269	106	39%
Cruise Control	305	126	41%
Chemical Mixing	577	254	44%
Shared pipeline infrastructure: 2,361 LOC (model-independent)			

- **RQ6** *Does the model verified by nuXmv behave the same as the model used in the RL pipeline?*

4.1 Benchmarks

Table 1 summarizes the three systems. The *thermostat* switches a heater and an air conditioner to maintain a room at a target setpoint. The *cruise controller* regulates vehicle speed and following distance through throttle and brake actuation, requiring simultaneous satisfaction of speed and gap safety constraints. The *chemical mixing skid*, introduced in §2, orchestrates two pumps and two valves across a 14-phase PLC-style scan cycle to transfer fluid volumes between ingredient tanks and a mixing tank. This system is modeled after an evaluation skid at Idaho National Laboratory used for cyber-physical systems security research, preserving its PLC-style scan cycle, Modbus communication protocol, and physical safety constraints.

These systems span a range of complexity: actuator count grows from 2 to 4, the action space from 4 to 16, scan-cycle depth from 1 to 14 phases, and the number of safety requirements from 2 to 5. The dead-action ratio (§3.2.2) ranges from 25% to 44%, reflecting increasingly constrained safety specifications.

4.2 Automation (RQ1)

The central claim of CLARITY is that a single SysML v2 model is sufficient to drive verification, controller synthesis, and runtime monitoring with no manual translation at any boundary. Table 2 quantifies this automation at two levels.

From each SysML v2 design, CLARITY generates five artifacts: a nuXmv formal model, a Python simulation environment with specification-derived dynamics, a compiled safety shield, a contract-compliant labeling function for behavioral cloning, and a runtime monitor. The nuXmv model is 39–44% the size of its SysML v2

source, a reduction that reflects the non-trivial translation described in Figure 4. The engineer writes only the SysML v2 model; all downstream artifacts are derived automatically. The same 2,361 lines of shared infrastructure handle any supported SysML v2 model; adding a new benchmark requires only a new specification and no pipeline modification.

To ground these metrics in practice: prior to CLARITY, the authors manually implemented the mixing system's RL pipeline in approximately 8–12 engineer-hours, the majority spent redesigning reward functions after the controller repeatedly learned to take no action to avoid safety penalties. CLARITY completes the equivalent pipeline in under 9 minutes from the SysML v2 specification alone.

Answer to RQ1. An engineer contributes only the SysML v2 specification (269–577 LOC). CLARITY produces five artifacts that would otherwise require independent implementation—a process that previously required 8–12 hours for the mixing system alone.

4.3 Design Verification (RQ2)

CLARITY verified all 12 safety requirements across the three benchmarks using nuXmv [6]. The thermostat and cruise controller requirements are directly k-inductive and verify without strengthening. The chemical mixing system's 14-phase scan cycle creates intermediate states in which actuators are partially updated, breaking the inductive hypothesis for both the No Dead Heading and No Dry Running requirements. For this system, CLARITY automatically generates strengthening invariants that restore k-inductivity, requiring no manual intervention. Verification completes in under 2 seconds per system (Table 4).

Combined with the `#NeuralRequirement` contract (3.2.2) runtime enforcement of the shield, these results complete the soundness argument: the verified design guarantees safety under the contract assumption, and the shield ensures this holds at every step.

Answer to RQ2. All 12 safety requirements verify, including the mixing system's properties which depend on automatically generated strengthening invariants.

4.4 Controller Synthesis (RQ3)

All three benchmarks use identical training configurations (§3.2): 2,000 labeling function samples, 100 cloning epochs, 2,000 PPO episodes, batch size 100 with gradient checkpoints. Because the shield, labeling function, and reward are all derived from the specification, no per-system hyperparameter tuning is required.

Table 3 reports end-to-end results at the best evaluation checkpoint, selected over 100 greedy episodes by highest success rate with ties broken by reward. Standard deviation for intervals reports for 30 training seeds for each model. All three systems achieve 100% task completion with zero safety violations. The shield (3.2.2) guarantees safety: no action violating the `#NeuralRequirement` contract reaches the environment during training or deployment. The shield's intervention rate is low across all systems (0.2–12%), indicating that the learned policies produce predominantly compliant actions on their own.

The cruise controller and mixing system achieve average episode lengths within 1% of the theoretical optimum computed from the specification, indicating near-optimal learned behavior. All three systems converge under identical hyperparameters despite spanning action-space sizes from 4 to 16 and scan cycles from 1 to 14 phases—a consequence of the specification-derived shield, labeling function, and reward removing the need for per-system tuning.

Table 3: End-to-end results at the best evaluation checkpoint, selected over 100 greedy episodes per system. Results given with 95% confidence intervals over randomness of 30 training seeds. All systems achieve 100% task completion and incur zero safety violations.

System	Reqs Veri.	Avg Steps	Shield Override	Monitor Mean (μs)	Monitor P99 (μs)
Therm.	3/3	515±9.5	0.18±0.02%	11.8±0.1	14.5±0.2
Cruise	5/5	42.8±0.4	23±0.12%	16.5±0.19	62.2±1.3
Chem.	4/4	33.2±0.4	13±1.9%	18.8±1.2	68.4±1.7

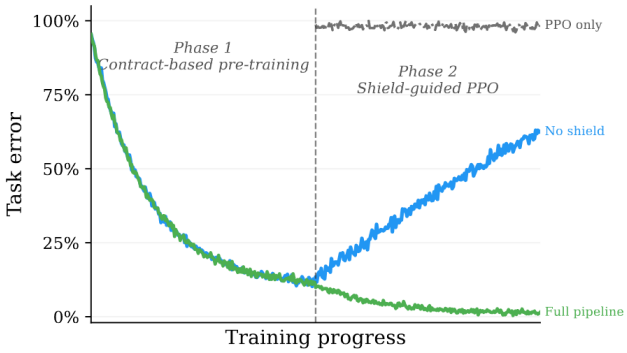

Figure 6: Task error across our Phase 1 and Phase 2 over training. Performance degradation in our controller can be seen in Phase 1 + Phase 2 PPO no Shield (blue) compared to Phase 1 + Phase 2 with Shield (green). The gray line shows the lack of progress when only Phase 2 PPO is used.

Figure 6 confirms that each pipeline component is necessary. Without pre-training, shielded PPO alone makes no progress (gray line): the policy cannot discover task-completing trajectories from random exploration within the contract-compliant action space, converging instead to a degenerate no-op—the same reward avoidance failure encountered during manual development (§4.2). With pre-training but without the shield (blue line), the policy initially benefits from behavioral cloning but degrades during PPO as stochastic exploration proposes contract-violating actions, achieving 0% success on all benchmarks after 2,000 episodes. Only the full pipeline (green line) achieves stable convergence, with task error reaching near zero. Without PPO, behavioral cloning handles single-step decisions but fails on temporal dynamics, producing end-of-episode failures on tasks requiring multi-step coordination.

Answer to RQ3. The synthesized pipeline produces controllers achieving 100% task completion with zero safety violations on all three benchmarks, using identical hyperparameters and no system-specific tuning. Each pipeline component—shield, labeling function, and PPO—is necessary and removing any one causes training to fail.

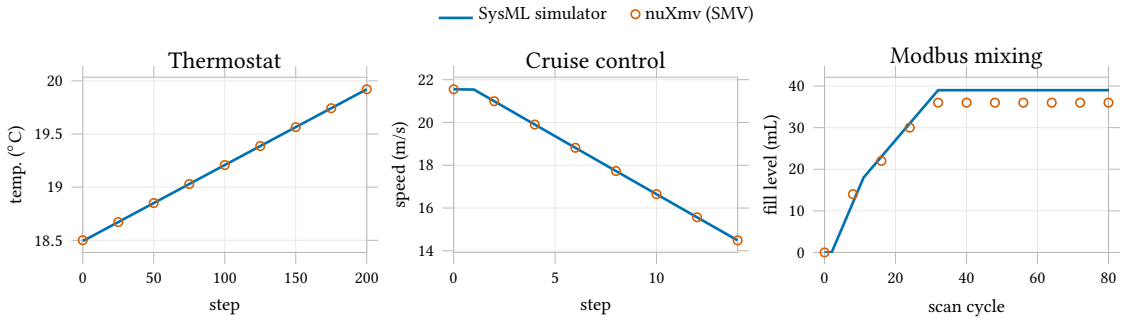

Figure 7: Semantic conformance of the extracted SMV model and our SysML simulator used during RL. Specifically, this figure shows the plant trajectory produced by the SysML simulator (line) and by nuXmv's synchronous SMV semantics (markers) under the same reference policy and inputs. Note the mixing plant agrees within a single Modbus scan cycle's transfer (≤ 3 mL).

Table 4: Pipeline execution cost per system. All experiments were conducted with 95% confidence intervals on a system with an AMD Ryzen 9 9950X, an NVIDIA RTX 5060Ti GPU, and Ubuntu 25.04.

System	Extract (s)	Verify (s)	Train (min)	Total (min)	Peak RSS (MB)
Thermostat	<5	<30	21.5±0.25	~30	240.0 ±2.7
Cruise Control	<5	<5	2.4±0.25	~5	92.2 ±0.2
Chemical Mixing	<5	5	2.05±0.02	~5	96.5 ±0.3

4.5 Runtime Monitoring (RQ4)

The runtime monitor is generated from the same contract specifications used for shield compilation and design verification (4.3). It confirms zero violations across all monitored deployment steps (Table 3), with per-check latencies of 25–30 µs on average and under 50 µs at P99. For a CPS operating at dt=0.1 s (100 ms control cycle), the monitor consumes less than 0.03% of the available budget. The monitor completes CLARITY's assurance chain: the verifier proves safety at design time, the shield enforces it during training and deployment, and the monitor confirms compliance at every step.

Answer to RQ4. The auto-generated monitor confirms zero violations with sub-40 µs per-check latency at negligible overhead. The verifier, shield, and monitor all derive from the same specification, providing a fully traceable assurance argument from design through deployment.

4.6 Pipeline Resource Costs (RQ5)

Table 4 reports wall-clock time and peak memory for each pipeline stage. Extraction completes in under a second for all systems via a single-pass AST traversal. Verification also terminates in seconds; the mixing system takes longest due to its larger state space. Training time is dominated by episode length rather than action-space size: the thermostat's slow thermal dynamics produce long episodes (~515 steps), requiring 25 minutes of training time despite having the smallest action space, while the cruise controller's and mixing system's shorter episodes (~43 and ~34 steps) complete training in under 2 minutes.

Answer to RQ5. On CPU, total times are 5–40 minutes with peak RSS ~250 MB. These costs support iterative use: an engineer can modify a SysML v2 model, re-run the pipeline, and evaluate the result within a single development session.

4.7 Semantic Conformance (RQ6)

The safety guarantees of §4.3 hold for the nuXmv model, yet the controller is trained and evaluated against the Python simulator (§3.2). Both are extracted from the same SysML v2 source, but through independent code paths, so a soundness gap remains unless the two implement the same semantics. We test this directly: for each benchmark we sample 50 scenarios (randomized setpoints, initial states, and inputs) and replay each in both engines under an identical reference policy, then compare the resulting trajectories step by step (Figure 7). The two engines stage signals differently: the simulator implements sense–decide–actuate as one step via sequential constraint solving, whereas nuXmv's synchronous `next(...)` semantics advances each register by one step.

Answer to RQ6. The verified model and the RL model implement the same behavior. Across all 50 scenarios per system the two engines agree on controller decisions in 94–100% of steps and on simulation time exactly; the thermostat and cruise plant trajectories coincide to machine precision ($\leq 2.1\times10^{-14}$), and the mixing plant agrees within a single scan cycle's transfer (≤ 3 mL). The residual differences are due to the staging artifact of a one-step register latency and the SMV's zero-initialized sensor registers on the first step, not semantic divergence. The safety properties proved by nuXmv (§4.3) therefore apply to the model the controller is actually trained and deployed against.

4.8 Threats to Validity

Scope. We evaluate on three benchmarks with discrete actuators, deterministic dynamics, and discrete action spaces. Industrial CPSs with continuous action spaces, stochastic disturbances, or deeper part hierarchies would stress different aspects of the pipeline. The translational semantics and shield construction generalize in principle, but continuous action spaces would require a solution for system constraints that we save for future work.

Table 5: Comparison with related approaches. ● = automated from the design; ◐ = partial; ○ = unsupported; M = manual.

	Design lang.	Action sem.	Design verif.	RL pipe. gen.	Shield gen.	Runtime monitor
CLARITY	●	●	●	●	●	●
JSC [8]	○	○	●	○	M	M
Gamma/Imandra [17, 23]	●	○	◐	○	○	○
Shielded RL [4, 5, 33]	○	○	○	○	◐	○
Aegis [30]	○	○	○	○	●	●
BBS/Barriers [16, 29, 34]	○	○	○	○	○	M
A/G [3, 15]	○	○	●	○	○	○

Metrics. Task completion and episode length are surrogate metrics for controller quality; a production deployment would impose additional criteria such as settling time or overshoot tolerance. These can be encoded as additional `#Prohibition` or `#Obligation` requirements, at which point the pipeline incorporates them automatically into shield enforcement, verification, and reward shaping.

5 Related Work

Table 5 summarizes the relationship between CLARITY and prior work.

Verification of system-level designs. Formal verification of system-level designs has a long history in CPS engineering. With the adoption of SysML v2, several tools have been integrated with the language. Molnár et al. [18] survey existing approaches to SysML v2 formal verification, including Gamma [17], Imandra [23], and constraint-propagation approaches. These tools verify statechart transitions and structural properties but do not support `action` bodies—the constructs CPS engineers use to model plant dynamics and sequential control logic. CLARITY addresses this with translational semantics covering the action fragment of SysML v2 (Figure 3), including sequential assignment, branching, message passing, and sub-action invocation.

Contract-based reasoning for neural components. When a system contains a black-box controller, model checking requires an abstraction that stands in for the controller's behavior. Bak and Chaki [3] combine software model checking with hybrid systems reachability to verify such compositions, while Duong et al. [15] apply assume-guarantee reasoning to compositional neural network verification. In both cases, the contract is written manually and maintained independently of the training pipeline. CLARITY instead embeds the contract directly in the SysML v2 design via `#Neural` and `#NeuralRequirement` annotations, ensuring the same contract constrains verification, training, and monitoring.

Safe and shielded reinforcement learning. Shielded RL restricts an agent to safe actions at each step [4, 5, 33], but existing approaches assume a manually provided shield or temporal logic formula and do not derive the shield from a verified system design. Temporal logic reward shaping methods [1, 12, 22] automate reward generation but assume a pre-existing environment while CLARITY generates the environment from the SysML v2 design. The most closely related work is Fulton and Platzer's Justified Speculative Control algorithm [8], which combines offline verification in differential dynamic logic with ModelPlex runtime monitors to constrain an RL agent. However, JSC operates on manually written hybrid programs—the shield, reward function, and runtime monitor must each be constructed by hand. CLARITY derives all three mechanically from a single SysML v2 specification.

Runtime assurance for neural controllers. Runtime assurance has been addressed via barrier certificates [2, 29], simplex architectures [16, 24], programmatic shields [20, 30], and certificate repair [34]. These approaches require engineers to re-encode safety properties in a formalism disconnected from both the system design and the training process. In CLARITY, the same `#NeuralRequirement` drives verification, shield compilation, and runtime monitoring—eliminating this re-encoding entirely.

6 Conclusion

We presented CLARITY, a system that connects design verification, reinforcement learning, and runtime monitoring through a single SysML v2 specification, eliminating the blind spots that arise when these concerns are addressed in isolation. Evaluation on three CPS benchmarks confirms that the approach verifies all 12 safety requirements, synthesizes controllers achieving 100% task completion with zero safety violations, and provides traceable runtime assurance—all from a single source of truth. Future work includes extending to continuous action spaces—requiring constraint-based shield compilation in place of truth-table enumeration—and integrating specification-derived dense reward functions for control objectives beyond our current sparse reward.

Data Availability

The data and artifacts supporting the results of this study are publicly available in a repository on Zenodo at https://doi.org/10.5281/zenodo.21126107 [21].

References

[1] Rajeev Alur, Suguman Bansal, Osbert Bastani, and Kishor Jothimurugan. 2022. *A Framework for Transforming Specifications in Reinforcement Learning.* Springer Nature Switzerland, 604–624. doi:10.1007/978-3-031-22337-2_29

[2] Mahathi Anand, Vishnu Murali, Ashutosh Trivedi, and Majid Zamani. 2022. k-Inductive Barrier Certificates for Stochastic Systems. In *25th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '22).* ACM, 1–11. doi:10.1145/3501710.3519532

[3] Stanley Bak and Sagar Chaki. 2016. Verifying cyber-physical systems by combining software model checking with hybrid systems reachability. In *Proceedings of the 13th International Conference on Embedded Software (ESWEEK '16).* ACM, 1–10. doi:10.1145/2968478.2968490

[4] Osbert Bastani and Shuo Li. 2021. Safe Reinforcement Learning via Statistical Model Predictive Shielding. In *Robotics: Science and Systems XVII (RSS2021).* Robotics: Science and Systems Foundation. doi:10.15607/rss.2021.xvii.026

[5] Asger Horn Brorholt, Peter Gjøl Jensen, Kim Guldstrand Larsen, Florian Lorber, and Christian Schilling. 2024. Shielded Reinforcement Learning for Hybrid Systems. In *Bridging the Gap Between AI and Reality*, Bernhard Steffen (Ed.). Springer Nature Switzerland, Cham, 33–54. doi:10.1007/978-3-031-46002-9_3

[6] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. *The nuXmv Symbolic Model Checker.* Springer International Publishing, 334–342. doi:10.1007/978-3-319-08867-9_22

[7] Kirill Djebko, Tom Baumann, Erik Dilger, Frank Puppe, and Sergio Montenegro. 2026. LeLaR: The First In-Orbit Demonstration of an AI-Based Satellite Attitude Controller. *IEEE Access* 14 (2026), 49348–49367. doi:10.1109/access.2026.3678816

[8] Nathan Fulton and André Platzer. 2018. Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 32, 1 (April 2018). doi:10.1609/aaai.v32i1.12107

[9] Kristin Giammarco and Kathleen Giles. 2017. *Verification and Validation of Behavior Models Using Lightweight Formal Methods.* Springer International Publishing, 431–447. doi:10.1007/978-3-319-62217-0_31

[10] Vinicius G. Goecks, Gregory M. Gremillion, Vernon J. Lawhern, John Valasek, and Nicholas R. Waytowich. 2020. Integrating Behavior Cloning and Reinforcement Learning for Improved Performance in Dense and Sparse Reward Environments. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1 (AAMAS '04)*. IEEE Computer Society, 465–473. doi:10.65109/knri7638

[11] Aicheng Gong, Yangkun Chen, Junjie Zhang, and Xiu Li. 2024. Possibilities of reinforcement learning for nuclear power plants: Evidence on current applications and beyond. *Nuclear Engineering and Technology* 56, 6 (2024), 1959–1974. doi:10.1016/j.net.2024.01.003

[12] M. Hasanbeig, Y. Kantaros, A. Abate, D. Kroening, G. J. Pappas, and I. Lee. 2019. Reinforcement Learning for Temporal Logic Control Synthesis with Probabilistic Satisfaction Guarantees. In *2019 IEEE 58th Conference on Decision and Control (CDC)*. IEEE, 5338–5343. doi:10.1109/cdc40024.2019.9028919

[13] IEC. 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems.

[14] ISO. 2018. Road vehicles – Functional safety.

[15] Xiaoyan Liu. 2024. Compositional Verification Using Geodesic Distance via Assume-Guarantee Reasoning. *IEEE Access* 12 (2024), 92612–92621. doi:10.1109/access.2024.3421239

[16] Usama Mehmood, Sanaz Sheikhi, Stanley Bak, Scott A. Smolka, and Scott D. Stoller. 2022. *The Black-Box Simplex Architecture for Runtime Assurance of Autonomous CPS*. Springer International Publishing, 231–250. doi:10.1007/978-3-031-06773-0_12

[17] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. 2018. The Gamma Statechart Composition Framework. In *40th International Conference on Software Engineering (ICSE)*. ACM, Gothenburg, Sweden, 113–116. doi:10.1145/3183440.3183489

[18] Vince Molnár, Bence Graics, András Vörös, Stefano Tonetta, Luca Cristoforetti, Greg Kimberly, Pamela Dyer, Kristin Giammarco, Manfred Koethe, John Hester, Jamie Smith, and Christoph Grimm. 2024. Towards the Formal Verification of SysML v2 Models. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24)*. ACM, 1086–1095. doi:10.1145/3652620.3687820

[19] Object Management Group. 2025. *OMG Systems Modeling Language (SysML), Version 2.0*. Technical Report. Object Management Group. https://www.omg.org/spec/SysML/2.0/Language/PDF formal/2025-09-xx.

[20] Haritz Odriozola-Olalde, Maider Zamalloa, and Nestor Arana-Arexolaleiba. 2023. Shielded Reinforcement Learning: A review of reactive methods for safe learning. In *2023 IEEE/SICE International Symposium on System Integration (SII)*. 1–8. doi:10.1109/SII55687.2023.10039301

[21] Austin O'Quinn, Conor Snedeker, and Max Taylor. 2026. *CLARITY – Artifact*. doi:10.5281/zenodo.21126107

[22] Austin O'Quinn and Max Taylor. 2025. Automated Synthesis of Verified Neural Network Controllers from Linear Temporal Logic Specifications. In *Proceedings of the 2025 Workshop on Re-design Industrial Control Systems with Security (RICSS '25)*. ACM, 26–34. doi:10.1145/3733823.3764517

[23] Grant Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. 2020. *The Imandra Automated Reasoning System (System Description)*. Springer International Publishing, 464–471. doi:10.1007/978-3-030-51054-1_30

[24] Dung T Phan, Radu Grosu, Nils Jansen, Nicola Paoletti, Scott A Smolka, and Scott D Stoller. 2020. Neural simplex architecture. In *NASA Formal Methods Symposium*. Springer, 97–114. doi:10.48550/arXiv.1908.00528

[25] Cheng Qian and Hongliang Ren. 2025. *Deep reinforcement learning in surgical robotics: Enhancing the automation level*. Elsevier, 89–102. doi:10.1016/b978-0-443-13271-1.00055-8

[26] Majdi I. Radaideh, Leo Tunkle, Dean Price, Kamal Abdulraheem, Linyu Lin, and Moutaz Elias. 2025. Multistep Criticality Search and Power Shaping in Nuclear Microreactors with Deep Reinforcement Learning. *Nuclear Science and Engineering* 200, sup1 (Feb. 2025). doi:10.1080/00295639.2024.2447012

[27] Aidan Rigby, Mike Wagner, Daniel Mikkelson, and Ben Lindley. 2025. A Reinforcement Learning Approach to Augment Conventional PID Control in Nuclear Power Plant Transient Operation. *Nuclear Technology* 212, 2 (May 2025), 427–445. doi:10.1080/00295450.2025.2466137

[28] Paul Seurin and Koroush Shirvan. 2026. Surpassing legacy approaches to pwr core reload optimization with single-objective reinforcement learning. *Nuclear Science and Engineering* 200, 3 (2026), 574–605. doi:10.1080/00295639.2025.2488702

[29] Meng Sha, Xin Chen, Yuzhe Ji, Qingye Zhao, Zhengfeng Yang, Wang Lin, Enyi Tang, Qiguang Chen, and Xuandong Li. 2021. Synthesizing Barrier Certificates of Neural Network Controlled Continuous Systems via Approximations. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 631–636. doi:10.1109/dac18074.2021.9586327

[30] Jieke Shi, Junda He, Zhou Yang, Đorđe Žikelić, and David Lo. 2026. Synthesizing Efficient and Permissive Programmatic Runtime Shields for Neural Policies. *ACM Transactions on Software Engineering and Methodology* 35, 7 (2026), 1–38. doi:10.1145/3773034

[31] Faraz Torabi, Garrett Warnell, and Peter Stone. 2018. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954* (2018). doi:10.48550/arXiv.1805.01954

[32] Leo Tunkle, Kamal K. Abdulraheem, Linyu Lin, and Majdi I. Radaideh. 2025. Nuclear Microreactor Control with Deep Reinforcement Learning. (2025). doi:10.2139/ssrn.5203632

[33] Wen-Chi Yang, Giuseppe Marra, Gavin Rens, and Luc De Raedt. 2023. Safe Reinforcement Learning via Probabilistic Logic Shields. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (IJCAI-2023)*. International Joint Conferences on Artificial Intelligence Organization, 5739–5749. doi:10.24963/ijcai.2023/637

[34] Emily Yu, Đorđe Žikelić, and Thomas A. Henzinger. 2025. Neural Control and Certificate Repair via Runtime Monitoring. *Proceedings of the AAAI Conference on Artificial Intelligence* 39, 25 (April 2025), 26409–26417. doi:10.1609/aaai.v39i25.34840

Received 2026-03-26; accepted 2026-06-18