Universal Safety Controllers with Learned Prophecies*

Bernd Finkbeiner[1], **Niklas Metzger**[1], **Satya Prakash Nayak**[2], **Anne-Kathrin Schmuck**[2]
[1]CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
[2]Max Planck Institute for Software Systems, Kaiserslautern, Germany
{finkbeiner, niklas.metzger}@cispa.de, {sanayak, akschmuck}@mpi-sws.org

Abstract

Universal Safety Controllers (USCs) are a promising logical control framework that guarantees the satisfaction of a given temporal safety specification when applied to any realizable plant model. Unlike traditional methods, which synthesize one logical controller over a given detailed plant model, USC synthesis constructs a *generic controller* whose outputs are conditioned by plant behavior, called *prophecies*. Thereby, USCs offer strong generalization and scalability benefits over classical logical controllers. However, the exact computation and verification of prophecies remain computationally challenging. In this paper, we introduce an approximation algorithm for USC synthesis that addresses these limitations via learning. Instead of computing exact prophecies, which reason about sets of trees via automata, we only compute under- and over-approximations from (small) example plants and infer computation tree logic (CTL) formulas as representations of prophecies. The resulting USC generalizes to unseen plants via a verification step and offers improved efficiency and explainability through small and concise CTL prophecies, which remain human-readable and interpretable. Experimental results demonstrate that our learned prophecies remain generalizable, yet are significantly more compact and interpretable than their exact tree automata representations.

1 Introduction

The automatic construction of correct-by-design systems is central to both *reactive synthesis*, which derives correct implementations from temporal logic specifications, and *supervisory control*, which derives correct restrictions of existing open systems (plants). In practice, control tasks often combine both perspectives: behavioral goals are specified in logic, while the physical system, i.e., robots, sensors, or environments, is modeled as a plant. This has led to a large variety of synthesis approaches that combine specification automata with plant models and solve the resulting ω-regular game to derive a correct-by-design controller (see books and surveys by Tabuada (2009); Belta, Yordanov, and Gol (2017); Yin, Gao, and Yu (2024)). These approaches are, however, known to face major scalability challenges: exploring all plant behaviors in a model can lead to an intractable explosion in the state space during synthesis. Even more importantly, when the plant is only known at runtime, or evolves over time, then the usual state exploration becomes impractical very quickly.

These severe limitations have motivated the introduction of *Universal Safety Controllers (USCs)* by Finkbeiner et al. (2025a), which shift the focus from plant-specific synthesis to the computation of a *universal controller* derived from the specification alone, whose decisions are conditioned by so-called *prophecies*. As a result, USCs promise two major advantages: (1) generalization: USCs provide a correct solution for all realizable plant models, and (2) computational efficiency: by focusing on the specification, USC synthesis promises to reduce the computational burden of utilizing a complete plant model. The latter computational benefit, however, is only achieved if prophecies are small and concise. Finkbeiner et al. (2025a) use tree automata as prophecies which encode all possible future branching structure of the plant potentially relevant for the specification. While preserving correctness and completeness, this technique renders both prophecy synthesis and prophecy verification computationally very intense.

This paper addresses this limitation of the automata-based approach by introducing the first *learning-based* algorithm for prophecy construction. Concretely, we obtain prophecies in computation tree logic (CTL) (Clarke and Emerson 1981), which we learn from a small set of representative (nominal) plant models drawn from the application domain. The resulting USC generalizes to unseen plants via a verification step and offers improved efficiency and explainability through small and concise CTL prophecies, which remain generalizable and human-readable.

1.1 Motivating Example

As a motivating example, we consider the problem of synthesizing a simple load-balancing scheduling controller, which assigns incoming tasks to two processing units (CPUs). The load-balancing controller is specified with the following temporal formula:

$$\varphi := \square(\mathit{task} \rightarrow \bigcirc(\mathit{asgn}_1 \vee \mathit{asgn}_2)) \wedge \square\neg \mathit{overload}, \quad (1)$$

where task models the arrival of a new task and is controlled by the *environment*, asgn_i assigns an incoming task to either cpu_1 or cpu_2 and is controlled by the controller, and

[*] Authors are ordered alphabetically.

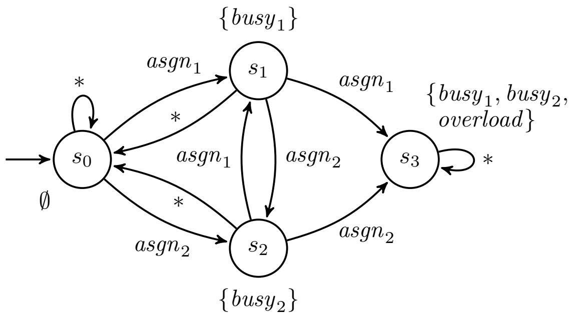

Figure 1: A plant that models the occupancy of processors in the load balancer example. We use $*$ to represent edges that are taken whenever no other edge guard is satisfied.

$overload$ means that a task was assigned to a busy processor, which is controlled by the *plant*. The specification states that there should never be an $overload$, i.e., the controller should never assign a task to a processor that is already busy with another task.

Ultimately, we are interested in applying the scheduler to a concrete system implementation, such as the system described by the plant model in Fig. 1. In this particular system implementation, each of the processors is busy for exactly one time-step once assigned a task. An example of a controller that satisfies our specification for this plant, and would typically be found by a standard synthesis algorithm, is the round-robin controller, which alternates between assigning new tasks to the first and second CPU. The disadvantage of this controller, however, is that it *only* works correctly for our *specific* plant. By contrast, a universal controller is immediately applicable to an entire set of plants.

Previous work: automata-based USC synthesis. The previous approach to universal controller synthesis, UNICON by Finkbeiner et al. (2025a), takes as input only a specification: the USC is computed independently of any specific plant. For each possible output of the controller, the synthesis algorithm computes the exact condition on the plant under which this output is correct. In our example, both $asgn_1$ and $asgn_2$ might be correct outputs, depending on whether the respective CPU is available, and also depending on the *future* availability of the CPU. For example, if, in a hypothetical plant, cpu_1 would become permanently busy if used in the first step, it would only be correct to start with $asgn_2$. UNICON computes a tree automaton that recognizes *exactly those* plants where $asgn_1$ (or $asgn_2$, respectively) does not lead to an immediate $overload$ and where, additionally, a control strategy exists for the state reached by executing $asgn_1$ (or $asgn_2$, respectively). The tree automaton is nontrivial and difficult to interpret; we omit depicting it here.

New approach: learning-based USC synthesis. Our new learning-based approach, UCLEARN, takes as input both the specification and a set of *nominal* plants which are representative of the plants the controller will be applied to. Similar to UNICON, the prophecies express conditions on the plants for which a certain output should be applied. However, instead of *characterizing* precisely the set of plants for which a particular output is correct, our learned prophecies *separate* plants for which a control output is correct from those where the output is incorrect. This relaxation introduces a freedom of choice in the selection of the separating condition. The precise condition computed by UNICON is also one of the separating conditions, but typically, there are separating conditions that are much simpler, easier to verify, and easier to understand. We find such a condition, expressed as a temporal formula, using a learning algorithm for CTL.

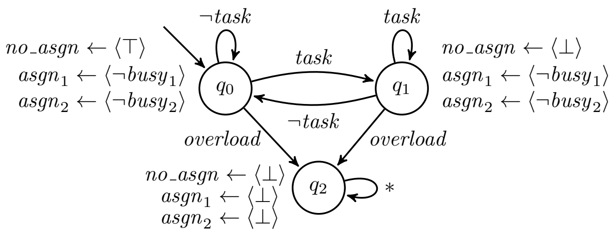
$\neg task$
$task$
$no\_asgn \leftarrow \langle\top\rangle$
$asgn_1 \leftarrow \langle\neg busy_1\rangle$
$asgn_2 \leftarrow \langle\neg busy_2\rangle$
q_0
$task$
q_1
$no\_asgn \leftarrow \langle\bot\rangle$
$asgn_1 \leftarrow \langle\neg busy_1\rangle$
$asgn_2 \leftarrow \langle\neg busy_2\rangle$
$\neg task$
$overload$
$overload$
$no\_asgn \leftarrow \langle\bot\rangle$
$asgn_1 \leftarrow \langle\bot\rangle$
$asgn_2 \leftarrow \langle\bot\rangle$
q_2
$*$

Figure 2: An approximate USC with CTL prophecies.

Fig. 2 shows the controller synthesized by UCLEARN (Nayak et al. 2025) for the specification of the load-balancing controller together with the plant from Fig. 1 as the (single) nominal plant. The initial state q_0 represents the case that no task has been received yet. Here, no_asgn is a correct decision independently of the plant. The outputs $asgn_1$ and $asgn_2$ are allowed if the respective CPU is not busy. State q_1 represents that a task has been received and needs to be assigned to a CPU. The difference to q_0 is that no_asgn is no longer an option. Finally, q_2 represents an overload. This happens with the plant from Fig. 1, although it might happen on plants which the synthesis algorithm has not seen. In q_2, no controller output is correct, because the specification has already been violated.

The synthesized controller is not only correct for the given plant, it also generalizes to other plants that have sufficient CPU availability and use $busy$ like in our example. UCLEARN thus produces controllers that are more general than the plant-specific controllers computed by classic algorithms from reactive synthesis and supervisory control, and simpler than the universal controllers produced by UNICON.

1.2 Overview

Fig. 3 gives an overview of UCLEARN. The general idea is that an approximate USC is initially obtained from the specification and then continuously refined based on the plants provided to the algorithm. An approximate USC provides for each state and controller output an over- and an under-approximation of the prophecy: the under-approximation $\underline{\kappa}$ characterizes plants for which the output is guaranteed to be correct, the over-approximation $\overline{\kappa}$ characterizes plants for which the output is not guaranteed to be incorrect. As shown in the upper part of Fig. 3, we initialize the process by translating the safety LTL formula to an automaton, and setting $\underline{\kappa} = \emptyset$, i.e., the controller output is guaranteed to be correct for the empty set, and $\overline{\kappa} = \mathbb{P}$, i.e., the output is not guaranteed to be incorrect for all plants.

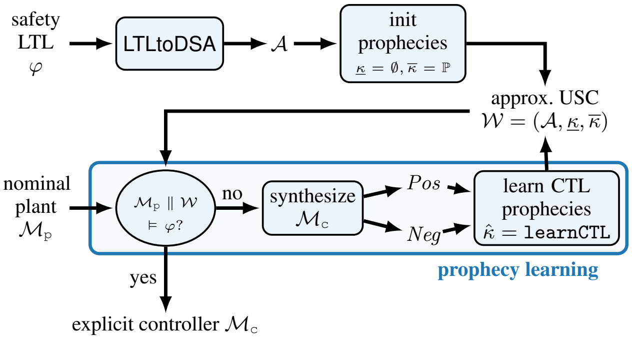
safety
LTL
φ
LTLtoDSA
$\mathcal{A}$
init
prophecies
$\underline{\kappa} = \emptyset, \overline{\kappa} = \mathbb{P}$
approx. USC
$\mathcal{W} = (\mathcal{A}, \underline{\kappa}, \overline{\kappa})$
nominal
plant
$\mathcal{M}_{\mathtt{p}}$
$\mathcal{M}_{\mathtt{p}} \parallel \mathcal{W} \vDash \varphi?$
no
synthesize
$\mathcal{M}_{\mathtt{c}}$
Pos
Neg
learn CTL
prophecies
$\hat{\kappa} = \mathtt{learnCTL}$
prophecy learning
yes
explicit controller $\mathcal{M}_{\mathtt{c}}$

Figure 3: An overview of UCLearn.

The main refinement loop, framed in red in Fig. 3, applies the current approximation $\mathcal{W}$ to a new nominal plant and uses the result to improve $\mathcal{W}$. We first concretize the universal controller to an explicit controller for the new plant by choosing the outputs on which the prophecy conditions are satisfied. We then verify if the explicit controller is correct for the plant: if yes, $\mathcal{W}$ already covers the plant and there is no need to refine. If the controller is incorrect, we synthesize a correct controller and obtain positive and negative samples based on the states in the plant where the controller acts correctly and incorrectly, respectively. A separate CTL learning algorithm (`learnCTL`) then finds a formula that separates the new sets of positive/negative samples.

1.3 Related Work

The generalization and adaptability of the universal controller stem from its *permissiveness*, i.e., its ability to represent a set of control strategies rather than a single one. Permissiveness has been widely studied in supervisory control (Cassandras and Lafortune 2021) and reactive synthesis (Bernet, Janin, and Walukiewicz 2002; Bouyer et al. 2011; Fremont and Seshia 2018; Klein, Baier, and Klüppelholz 2015; Anand, Nayak, and Schmuck 2023), though typically under a fixed plant model with limited adaptability. Other works consider strategies correct for sets of plants, e.g., dominant (Damm and Finkbeiner 2014; Finkbeiner and Passing 2022) and admissible (Berwanger 2007; Basset, Raskin, and Sankur 2017) strategies, or restricted plant classes (Anand et al. 2023; Brenguier, Raskin, and Sankur 2017)—but they yield a single strategy correct for all plants. In contrast, our method synthesizes a controller that adapts its strategy to the given plant model.

Temporal logic specification learning has been well studied (Neider and Gavran 2018; Raha et al. 2023; Valizadeh, Fijalkow, and Berger 2024; Pommellet, Stan, and Scatton 2024), but these works focus on inferring specifications from samples, not using them for control. Learning has been applied to LTL synthesis without plant models (Kretínský et al. 2025; Balachander, Filiot, and Raskin 2023), whereas we learn CTL formulas that characterize plant behavior. We also leverage prophecy variables (Abadi and Lamport 1991; Beutner and Finkbeiner 2022) to anticipate future plant actions during universal controller synthesis, following the framework introduced in (Finkbeiner et al. 2025a).

2 Preliminaries

System Architectures. For every set X, we write X^* and X^ω to denote the sets of finite and respectively infinite, sequences of elements of X, and let $X^\infty = X^* \cup X^\omega$. For $\rho \in X^\infty$, we denote $|\rho| \in \mathbb{N} \cup \{\infty\}$ the length of ρ, and define $dom(\rho) := \{0, 1, \ldots, |\rho| - 1\}$. For $\rho = x_0 x_1 \cdots \in X^\infty$ and $i \in dom(\rho)$, we write $\rho[i]$ to denote its i-th element, i.e., x_i. For $\rho \in X^\infty$ and some set Y, we write $\rho \downarrow_Y$ to denote the sequence ρ' with $\rho'[i] = \rho[i] \cap Y$. We focus on systems with three processes, $\mathrm{Proc} = \{\mathtt{c}, \mathtt{p}, \mathtt{e}\}$, i.e., a controller, a plant, and an environment and $\mathtt{AP} = O_{\mathtt{e}} \uplus O_{\mathtt{c}} \uplus O_{\mathtt{p}}$ partitioned into the output propositions of the processes. This yields an architecture $\mathtt{Arc} = (I_{\mathtt{e}}, O_{\mathtt{e}}, I_{\mathtt{c}}, O_{\mathtt{c}}, I_{\mathtt{p}}, O_{\mathtt{p}})$ over $\mathtt{AP}$ where the input propositions of a process $i \in \mathrm{Proc}$ are output propositions of all other processes, i.e., $I_i = \bigcup_{j \neq i} O_j$.

Traces and Specifications. We fix alphabet $\Sigma = 2^{\mathtt{AP}}$ to the power set of the atomic propositions $\mathtt{AP}$. A *trace* over Σ is a sequence $\gamma \in \Sigma^\omega$. A *specification* φ specifies some restrictions on the traces. We write $\gamma \vDash \varphi$ to denote that the trace γ satisfies the specification φ. The language $\mathcal{L}(\varphi)$ of a specification φ represents the set of traces that satisfy φ. Linear-time temporal logic (LTL) (Pnueli 1977) specifications over atomic propositions $\mathtt{AP}$ are defined by $\varphi, \psi ::= \alpha \in \mathtt{AP} \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \bigcirc \varphi \mid \varphi \, U \, \psi \mid \Diamond \varphi \mid \Box \varphi$. Computation tree logic (CTL) (Clarke and Emerson 1981) specifies temporal logics over trees by differentiating between path formulas and state formulas. Paths are quantified with $\forall \varphi$ (φ holds on all paths) and $\exists \varphi$ (φ holds on some path). Additionally, every temporal modality must be preceded by a path quantifier. A system is said to satisfy an LTL or CTL specification φ as defined by the usual semantics; we refer to (Baier and Katoen 2008) for more details.

Process Strategies. We model a strategy for process i as a Moore machine $\mathcal{M} = (S, s_0, \tau, o)$ over inputs I_i and outputs O_i, consisting of a finite set of states S, an initial state $s_0 \in S$, a transition function $\tau : S \times 2^{I_i} \to S$ over inputs, and an output labeling function $o : S \to 2^{O_i}$. For a finite input sequence $\gamma = \alpha_0 \alpha_1 \cdots \alpha_{k-1} \in (2^{I_i})^*$, $\mathcal{M}$ produces a finite path $s_0 s_1 \cdots s_k$ and an output sequence $o(s_0) o(s_1) \cdots o(s_k) \in (2^{O_i})^*$ such that $\tau(s_j, \alpha_j) = s_{j+1}$. We write $out(\mathcal{M}, \gamma)$ to denote $o(s_k)$. Similarly, for an infinite input sequence $\gamma \in (2^{I_i})^\omega$, $\mathcal{M}$ produces an infinite path $s_0 s_1 \cdots$ and an infinite output sequence $o(s_1) o(s_1) \cdots \in (2^{O_i})^\omega$. $Traces(\mathcal{M})$ denotes the set of all traces $\gamma \in \Sigma^\omega$ such that for input sequence $\gamma \downarrow_{I_i}$, $\mathcal{M}$ produces the output sequence $\gamma \downarrow_{O_i}$. We say that $\mathcal{M}$ satisfies a specification φ, denoted $\mathcal{M} \vDash \varphi$, if $\gamma \vDash \varphi$ holds for all trace $\gamma \in Traces(\mathcal{M})$. For a strategy $\mathcal{M} = (S, s_0, \tau, o)$, we write $\mathcal{M}(s)$ to denote the strategy (S, s, τ, o) with the same transition and output labeling function but with initial state s. The set of all plant strategies and controller strategies are $\mathbb{P}$ and $\mathbb{C}$, respectively. The parallel composition $\mathcal{M}_i \parallel \mathcal{M}_j$ of two strategies $\mathcal{M}_i = (S_i, s_0^i, \tau_i, o_i)$, $\mathcal{M}_j = (S_j, s_0^j, \tau_j, o_j)$ of processes $i, j \in \mathrm{Proc}$ is a strategy, i.e., a Moore machine, (S, s_0, τ, o) over inputs $(I_i \cup I_j) \setminus (O_i \cup O_j)$ and outputs $O_i \cup O_j$ with $S = S_i \times S_j$, $s_0 = (s_0^i, s_0^j)$, $\tau((s, s'), \sigma) = (\tau_i(s, (\sigma \cup o_j(s')) \cap I_i), \tau_j(s', (\sigma \cup o_i(s)) \cap I_j))$, and $o((s, s')) = o_i(s) \cup o_j(s')$.

ω-Automata. An *ω-automaton* $\mathcal{A}$ over alphabet Σ is a tuple (Q, q_0, δ, Ω) consisting of a finite set of states Q, an initial state $q_0 \in Q$, a transition function $\delta\colon Q \times \Sigma \to Q$, and an acceptance condition $\Omega \subseteq Q^\omega$. The unique *run* of $\mathcal{A}$ from state q on some trace $\gamma \in \Sigma^\infty$, denoted by $run(\mathcal{A}, q, \gamma)$, is a sequence of states $\rho \in Q^\infty$ with $|\rho| = |\gamma| + 1$, $\rho[0] = q$, and $\delta(\rho[i], \gamma[i]) = \rho[i + 1]$ for all $i \in dom(\gamma)$. A run ρ is *accepting* if $\rho \in \Omega$. The language $\mathcal{L}(\mathcal{A})$ is the set of all traces γ for which the unique run $run(\mathcal{A}, q_0, \gamma)$ is accepting. Furthermore, we write $\mathcal{A}(q) = (Q, q, \delta, \Omega)$ to denote the automaton with the same transition function and acceptance condition but with initial state q, and $\texttt{reachable}(\mathcal{A})$ to denote the set of states that are reachable from the initial state q_0 in $\mathcal{A}$, i.e., there exists a run ρ of $\mathcal{A}$ with $\rho[0] = q_0$ and $\rho[k] = q$ for some $k \geq 0$.

In *safety* automata, the acceptance $\Omega = \mathit{safe}(F)$ is given by a set $F \subseteq Q$ of safe states, where the automaton accepts the set of runs which only visits states in F. An LTL specification φ is said to be *safety* if it can be expressed as a safety automaton $\mathcal{A}$. A standard result in the literature is that every safety LTL specification can be translated into an equivalent *deterministic* safety automaton with a double exponential blow-up (Latvala 2003). We denote this procedure by $\mathsf{LTLtoDSA}(\varphi)$.

Let $\mathcal{A} = (Q, q_0, \delta, \Omega)$ be an automaton, and $\mathcal{M}$ be a Moore machine over inputs I and outputs O with $2^{I \cup O} \subseteq \Sigma$. We define the composition $\mathcal{A} \times \mathcal{M} = (Q \times S, (q_0, s_0), \delta', \Omega')$ as the product automaton with acceptance condition Ω' given by the set of runs whose first component is accepting by Ω, and a partial transition function $\delta' : (Q \times S) \times \Sigma \to (Q \times S)$ such that $\delta'((q, s), \sigma)$ is defined if and only if $\sigma \cap O = o(s)$ and in that case $\delta'((q, s), \sigma) = (\delta(q, \sigma), \tau(s, \sigma))$. We write $Runs(\mathcal{A}, q, \mathcal{M})$ to denote the set of runs ρ of $\mathcal{A}$ for which there exists a corresponding run ρ' of $\mathcal{A} \times \mathcal{M}$ with $\rho'[i] = (\rho[i], s_i)$ for each $i \geq 0$. We write $\texttt{reachable}(\mathcal{A} \times \mathcal{M})$ to denote the set of reachable states (q, s) from the initial state (q_0, s_0) in $\mathcal{A} \times \mathcal{M}$, i.e., there exists a run ρ of $\mathcal{A} \times \mathcal{M}$ with $\rho[0] = (q_0, s_0)$ and $\rho[k] = (q, s)$ for some $k \geq 0$.

Logical Controller Synthesis. Logical controller synthesis addresses the problem of, given a specification and a plant, synthesizing a controller that satisfies the specification if executed with the plant. For an LTL specification φ over $\texttt{AP}$, and a fixed plant strategy $\mathcal{M}_\mathrm{p}$, the goal is to find a controller strategy $\mathcal{M}_\mathrm{c}$ such that $\mathcal{M}_\mathrm{c} \parallel \mathcal{M}_\mathrm{p} \vDash \varphi$. A plant strategy is *admissible* if such a controller exists.

3 Universal Controller Synthesis

In this section, we recall the notion of universal controllers and their synthesis from (Finkbeiner et al. 2025a). Traditional controller synthesis requires exploring the state space of a given plant. In contrast, we consider a synthesis framework that abstracts away from explicit plants by reasoning over sets of possible plants, called *prophecies*. A prophecy captures assumptions about the future plant behavior. Instead of solving the synthesis problem for each plant individually, we synthesize a controller that conditions its behavior on verified prophecies for the given plant. This leads to the notion of a *prophecy-annotated controller*[1] (*prophecy controller* for short), which conditions controller behavior with prophecies.

Definition 1 (Prophecy-annotated controller)**.** *A* prophecy *$\theta \subseteq \mathbb{P}$ is a set of plants. A* prophecy-annotated controller *$\mathcal{U}$ over an alphabet Σ and a set of prophecies $\mathbb{F}$ is a tuple (S, s_0, τ, κ), where S is a finite set of states, $s_0 \in S$ is the initial state, $\tau : S \times \Sigma \to S$ is the transition function, $\kappa : S \times 2^{O_c} \to \mathbb{F}$ is the prophecy annotation.*

Note that, similar to ω-automata, the transition function τ is defined over all possible alphabets, and hence, $\mathcal{U} \times \mathcal{M}$ is well-defined for any strategy $\mathcal{M}$. Furthermore, each controller output from a state is associated with a prophecy that must hold for that output to be valid. For an explicit plant, we compute a consistent controller by verifying the prophecies at each state.

Definition 2 (Consistency)**.** *Given a prophecy controller $\mathcal{U} = (S, s_0, \tau, \kappa)$ and a plant $\mathcal{M}_p$, a controller $\mathcal{M}_c = (S^c, s_0^c, \tau^c, o^c)$ is said to be consistent with $\mathcal{U}$ w.r.t. $\mathcal{M}_p$, denoted by $\mathcal{M}_c \vDash \mathcal{U} \parallel \mathcal{M}_p$, if for all $(s, s^p, s^c) \in \texttt{reachable}(\mathcal{U} \times (\mathcal{M}_c \parallel \mathcal{M}_p))$, it holds that $\mathcal{M}_p(s^p) \in \kappa(s, o^c(s^c))$.*

We are interested in prophecy controllers that are correct when combined with a consistent controller for a plant.

Definition 3 (Correctness)**.** *Given a specification φ, a prophecy controller $\mathcal{U}$ is* correct *for a plant strategy $\mathcal{M}_p$, if it holds that $\mathcal{M}_c \vDash \mathcal{U} \parallel \mathcal{M}_p$ implies $\mathcal{M}_p \parallel \mathcal{M}_c \vDash \varphi$.*

It is possible that there does not exist a controller $\mathcal{M}_\mathrm{c}$ that is consistent with the prophecy controller $\mathcal{U}$ for a given plant $\mathcal{M}_\mathrm{p}$. In that case, we say $\mathcal{U}$ is *incompatible* with $\mathcal{M}_\mathrm{p}$, denoted by $\mathcal{U} \parallel \mathcal{M}_\mathrm{p} = \emptyset$. If all correct controllers for an (admissible) plant are consistent with the universal controller, we call such a universal controller *most permissive*.

Definition 4 (Permissiveness)**.** *Given a specification φ, a prophecy controller $\mathcal{U}$ is* most permissive *for a plant strategy $\mathcal{M}_p$, if it holds that $\mathcal{M}_p \parallel \mathcal{M}_c \vDash \varphi$ implies $\mathcal{M}_c \vDash \mathcal{U} \parallel \mathcal{M}_p$.*

The goal of universal controller synthesis is to construct a prophecy controller that is correct and most permissive for every plant. This implies it would produce correct controllers for all admissible plants and would be incompatible for all inadmissible plants. However, note that a trivial solution to this problem is a prophecy controller that includes all plants in the prophecies from safe states and excludes all plants in the prophecies from unsafe states. This is due to the fact that every plant (including inadmissible ones) would satisfy the prophecies as long as the composition is in the safe region, and would become incompatible once the composition reaches an unsafe state. To avoid such trivial solutions, we require that if a plant satisfies a prophecy from a state, it must be compatible with the controller from that state. This can be formalized as follows: a prophecy controller $\mathcal{U} = (S, s_0, \tau, \kappa)$ is said to be *forward-complete* if for every plant $\mathcal{M}_\mathrm{p}$, if $\mathcal{M}_\mathrm{p} \in \kappa(s, \alpha)$ for some $s \in S$ and

[1] Note that (Finkbeiner et al. 2025a) refers to prophecy-annotated controllers as *universal controllers*. We call a prophecy-annotated controller *universal* iff it is correct and most permissive.

$\alpha \in 2^{O_\mathrm{c}}$, then $\mathcal{U}(s) \parallel \mathcal{M}_\mathrm{p} \neq \emptyset$. With this definition, we can now define a universal controller.

Definition 5 (Universal Controller)**.** *A universal controller $\mathcal{U}$ for a specification φ is a forward-complete prophecy controller that is correct and most permissive for every plant.*

In (Finkbeiner et al. 2025a), the authors present a procedure to synthesize a universal safety controller (USC), i.e., a universal controller for safety LTL specifications. The procedure is based on the safety automaton of the specification and shows that the correct safe prophecies can be formulated as tree automata (Grädel, Thomas, and Wilke 2002). Furthermore, it guarantees that the USC uses the same transition structure as its safety automaton (Finkbeiner et al. 2025a). Hence, from now on, we only consider and refer to USCs (or any prophecy controller) as its automaton with prophecy annotations, i.e., $\mathcal{U} = (\mathcal{A}, \kappa)$.

4 Approximations of Universal Controllers

The construction of a USC in (Finkbeiner et al. 2025a) computes the exact set of correct prophecies, ensuring correctness and most permissiveness for *all* plants. In practice, however, we typically work with a restricted class of plants, and building the (often complex) USC is not practical. In this paper, we consider approximations of the USC that maintain both under- and over-approximations of the prophecies. These approximations can be used to generate positive and negative examples for learning the prophecies.

4.1 Approximations

Our goal is to construct prophecy controllers that provide both sound under- and over-approximations of the USC's prophecies. We formalize this notion below.

Definition 6 (Approximations)**.** *Given a USC $\mathcal{U} = (\mathcal{A}, \kappa)$ for a safety LTL specification, an* approximation *is a tuple $\mathcal{W} = (\mathcal{A}, \underline{\kappa}, \overline{\kappa})$, where $\underline{\kappa}$ and $\overline{\kappa}$ are prophecy annotations such that: $\underline{\kappa}(q, \alpha) \subseteq \kappa(q, \alpha) \subseteq \overline{\kappa}(q, \alpha)$ for all states q and propositions $\alpha \subseteq O_c$.*

Here, $\underline{\kappa}$ is an under-approximation and $\overline{\kappa}$ is an over-approximation of the prophecy annotations κ of the USC $\mathcal{U}$. We refer to under-approximation and over-approximation as $\underline{\mathcal{W}} = (\mathcal{A}, \underline{\kappa})$ and $\overline{\mathcal{W}} = (\mathcal{A}, \overline{\kappa})$, respectively. Note that the above definition does not require the approximations to be correct or most permissive for all plants. Nevertheless, it is straightforward to see that the under-approximation remains correct for every plant, while the over-approximation is most permissive for every plant.

Lemma 1. *For an approximation $\mathcal{W} = (\mathcal{A}, \underline{\kappa}, \overline{\kappa})$, its under-approximation $\underline{\mathcal{W}} = (\mathcal{A}, \underline{\kappa})$ is correct and its over-approximation $\overline{\mathcal{W}} = (\mathcal{A}, \overline{\kappa})$ is most permissive for all plants.*

4.2 Refinements

While Lemma 1 guarantees that any under-approximation is correct and any over-approximation is most permissive, such approximations are not immediately useful in practice. For example, a trivial approximation sets $\underline{\kappa}$ to the empty set and $\overline{\kappa}$ to the set of all plants, which yields no useful insight about the actual plants. In practical scenarios, approximations become valuable when the synthesis procedure targets a specific class of plants: For every new plant for which we synthesize a solution, we can refine the approximation $\mathcal{W}$. This incrementally increases the precision, targeting an approximation that is correct and permissive for this exact class of plants. In this scenario, we assume that the general structure of the synthesis problem remains consistent, while specific aspects, such as the arrangement of obstacles, vary between plants. For example, a refinement of the approximate controller in our running example (see Fig. 2) could be that some processor becomes unavailable under some circumstances. Once this plant is observed, the approximations are refined to check exactly for this behavior. While the controller strategies remain largely consistent across different plants, the associated prophecies may vary. To address this, we apply *refinements* to update an existing approximation so that it remains correct and most permissive for the current plant. To guarantee that each refinement increases the precision of the approximation, we formalize the following necessary conditions.

Algorithm 1: Refinement of an Approximation

```
Input: An approximation $\mathcal{W} = (\mathcal{A}, \underline{\kappa}, \overline{\kappa})$ with
    $\mathcal{A} = (Q, q_0, \Sigma, \delta, \Omega)$ and a plant $\mathcal{M}_\mathrm{p} = (S^\mathrm{p}, s_0^\mathrm{p}, \tau^\mathrm{p}, o^\mathrm{p})$.
Output: A refined approximation $\mathcal{W}' = (\mathcal{A}, \underline{\kappa}', \overline{\kappa}')$
let refine$(\mathcal{W}, \mathcal{M}_\mathrm{p})$ :=
    let $G = (Q \times S^\mathrm{p}, (q_0, s_0), \delta', \Omega') \leftarrow \mathcal{A} \times \mathcal{M}_\mathrm{p}$ // Compose
    let $Win \leftarrow \mathsf{solve}(G)$ // Winning states in the game
    for each $q \in Q$, $s^\mathrm{p} \in S^\mathrm{p}$, and $\alpha \subseteq O_\mathrm{c}$ do // All outputs
        if $\delta'((q, s^\mathrm{p}), \alpha \cup \beta) \in Win$ for every $\beta \in 2^{O_\mathrm{e}}$
            then $\underline{\kappa}(q, \alpha) \leftarrow \underline{\kappa}(q, \alpha) \cup \{\mathcal{M}_\mathrm{p}(s^\mathrm{p})\}$ // Add plant
            else $\overline{\kappa}(q, \alpha) \leftarrow \overline{\kappa}(q, \alpha) \setminus \{\mathcal{M}_\mathrm{p}(s^\mathrm{p})\}$// Remove plant
    return $(\mathcal{A}, \underline{\kappa}, \overline{\kappa})$
```

Definition 7 (Refinement)**.** *Given an approximation $\mathcal{W} = (\mathcal{A}, \underline{\kappa}, \overline{\kappa})$ and a plant $\mathcal{M}_p$, a* refinement *is a different approximation $\mathcal{W}' = (\mathcal{A}, \underline{\kappa}', \overline{\kappa}')$ such that:*

(i) $\underline{\kappa}(q, \alpha) \subseteq \underline{\kappa}'(q, \alpha)$ and $\overline{\kappa}'(q, \alpha) \subseteq \overline{\kappa}(q, \alpha)$ for all states q and propositions $\alpha \subseteq O_c$,

(ii) $\underline{\mathcal{W}}'$ and $\overline{\mathcal{W}}'$ are correct and most permissive for $\mathcal{M}_p$.

The definition of refinement implies that the approximation $\mathcal{W}$ is correct or permissive for at least one more plant. We claim that, in practice, every refinement steps captures a multitude of new plants.

4.3 Computing Refinements

We continue by providing algorithms for the computation of refinements. The goal of the refinement is to incorporate the information of the plant $\mathcal{M}_\mathrm{p}$ into the prophecy approximations. We do so by extracting all the sub-plants of the given plant and updating each prophecy annotation by adding or removing them based on their correctness.

The overall algorithm for refining an approximation is presented in Algo. 1. The algorithm takes an approximation $\mathcal{W} = (\mathcal{A}, \underline{\kappa}, \overline{\kappa})$ and a plant $\mathcal{M}_\mathrm{p}$ as input and returns a refined approximation $\mathcal{W}' = (\mathcal{A}, \underline{\kappa}', \overline{\kappa}')$. The first step is the

Algorithm 2: Learning Approximations

```
Input: An approximation $\mathcal{W} = (\mathcal{A}, \underline{\kappa}, \overline{\kappa})$ with
    $\mathcal{A} = (Q, q_0, \Sigma, \delta, \Omega)$ and a set of plants $Set \subseteq \mathbb{P}$.
Output: A prophecy controller $\mathcal{U}$.
let learnApprox$(\mathcal{W}, Set)$ :=
  for each $\mathcal{M}_{\mathrm{p}} \in Set$ do // For each plant in the set
    $\mathcal{W} \leftarrow$ refine$(\mathcal{W}, \mathcal{M}_{\mathrm{p}})$ // Refine the approximation
  for each $q \in Q, \alpha \subseteq O_{\mathrm{c}}$ do
    let $Pos \leftarrow \underline{\kappa}(q, \alpha)$ // Positive samples
    let $Neg \leftarrow \mathbb{P} \setminus \overline{\kappa}(q, \alpha)$ // Negative samples
    let $\phi \leftarrow$ learnCTL$(Pos, Neg)$ // Learn a CTL formula
    $\kappa(q, \alpha) \leftarrow \mathcal{L}(\phi)$ // Set the prophecy annotation
  return $(\mathcal{A}, \kappa)$
```

construction of the game G that ranges over the composition of the specification automaton $\mathcal{A}$ and the plant $\mathcal{M}_{\mathrm{p}}$. We solve this game to obtain the winning region and the states of the winning region Win. Next, based on the winning states, it iterates over all states $q \in Q$ and propositions $\alpha \subseteq O_{\mathrm{c}}$ to update the prophecy annotations $\underline{\kappa}'$ and $\overline{\kappa}'$. If a sub-plant $\mathcal{M}_{\mathrm{p}}(s^{\mathrm{p}})$, i.e., the strategy of the plant at state s^{p}, is a correct choice for the output α from state q, then it is added to the under-approximation $\underline{\kappa}'(q, \alpha)$. Otherwise, if it is not a correct choice, it is removed from the over-approximation $\overline{\kappa}'(q, \alpha)$. This ensures that the refined approximation $\mathcal{W}'$ is more precise than the original approximation $\mathcal{W}$, and it is correct and most permissive for the plant $\mathcal{M}_{\mathrm{p}}$. This is formalized in the following theorem and its proof can be found in the extended version (Finkbeiner et al. 2025b).

Theorem 1. *Given an approximation $\mathcal{W}$ and a plant $\mathcal{M}_p$, the procedure* `refine`$(\mathcal{W}, \mathcal{M}_p)$ *returns a refinement.*

One can start with a trivial approximation (e.g., $\underline{\kappa} = \emptyset$ and $\overline{\kappa} = \mathbb{P}$) and iteratively refine it with the information from the plants encountered in practice. Due to the monotonicity of the refinement, this also ensures that, at each step, both the under- and over-approximation preserve the solutions of previous computation steps.

5 Synthesis with Learned Prophecies

We now turn approximating controllers and the algorithms of the previous section into practice by learning representations of prophecies for under- and over-approximations. The target formalism for approximations are formulas in computation tree logic (CTL).

5.1 Learning CTL Formulas for Approximations

Prophecies of universal controllers, and their over- and under-approximation, are sets of plants. The algorithm presented in (Finkbeiner et al. 2025a), therefore, uses tree automata for prophecies that accept plants iff the considered output is correct for this plant. While tree automata represent prophecies precisely, and thereby preserve correctness and most permissiveness, they are computationally hard to compute, solve, and especially hard to understand. We choose a more lightweight formalism for categorizing trees: CTL formulas. While CTL formulas are not as expressive as tree automata (Baier and Katoen 2008), they are concise, human-readable, and easy to verify.

Algorithm 3: On-the-fly Composition

```
Input: A prophecy controller $\mathcal{U} = (\mathcal{A}, \kappa)$ with
    $\mathcal{A} = (Q, q_0, \Sigma, \delta, \Omega)$ and a plant $\mathcal{M}_{\mathrm{p}} = (S, s_0, \tau, o)$.
Output: An explicit controller $\mathcal{M}_{\mathrm{c}}$.
let compose$(\mathcal{U}, \mathcal{M}_{\mathrm{p}})$ :=
  let $s'_0 \leftarrow (q_0, s_0)$; $S' \leftarrow \{s'_0\}$; $\tau' \leftarrow []$; $o' \leftarrow []$ // Initialize
  let $queue \leftarrow Queue(s'_0)$ // Initialize the queue
  while $(q, s) \leftarrow queue.pop()$ do
    for each $\alpha \subseteq O_{\mathrm{c}}$ do // For each output
      if $\mathcal{M}_{\mathrm{p}}(s) \in \kappa(q, \alpha)$ then // If prophecy is satisfied
        let $o'(q, s) \leftarrow \alpha$ // Set the label
        for each $\beta \subseteq I_{\mathrm{c}}$ do // For each input, add transition
          $q'' \leftarrow \delta(q, \alpha \cup \beta)$; $s'' \leftarrow \tau(s, (\alpha \cup \beta) \cap I_{\mathrm{p}})$
          $\tau'((q, s), \beta) \leftarrow (q'', s'')$
          if $(q'', s'') \notin S'$ then // Add new states
            $S' \leftarrow S' \cup \{(q'', s'')\}$; $queue.push((q'', s''))$
        break // break the loop as we found a valid output
  return $(S', s'_0, \tau', o')$
```

Our algorithmic solution to constructing CTL formulas for prophecies is learning. The key idea is to use the set of plants in the under-approximation as positive samples and the complement of the set of plants in the over-approximation as negative samples. From these samples, we use standard CTL learning techniques (Bordais, Neider, and Roy 2024) to learn a CTL formula that accepts the positive samples and rejects the negative samples. This learned formula can then be used as the prophecy annotation – the over-approximation is the set of all plants that satisfy the formula, and the under-approximation is the set of all plants that do not satisfy the formula. The learning process is summarized in Algo. 2.

As the algorithm returns a prophecy annotation that classifies the plants in the under-approximation as positive samples and the plants not in the over-approximation as negative samples, by Thm. 1 and property (ii) of refinements, the resulting prophecy controller is guaranteed to be correct and most permissive for every plant in the set. We formalize this result in the following.

Corollary 1. *Given an approximation $\mathcal{W} = (\mathcal{A}, \underline{\kappa}, \overline{\kappa})$ and a plant set $Set \subseteq \mathbb{P}$, the procedure* `learnApprox`$(\mathcal{W}, Set)$ *returns a prophecy controller $\mathcal{U} = (\mathcal{A}, \kappa)$ such that: $\underline{\kappa}(q, \alpha) \subseteq \kappa(q, \alpha) \subseteq \overline{\kappa}(q, \alpha)$ for all states q and propositions $\alpha \subseteq O_c$. Furthermore, $\mathcal{U}$ is correct and most permissive for every plant in Set.*

5.2 Composition with Prophecy Controllers

The goal of Algo. 2 is to obtain a prophecy controller that correctly approximates the USC restricted to the class of plants similar to the ones in the set Set, i.e., the prophecy controller that is correct and most permissive for this class of plants. Once the prophecy controller is learned, we can compose the learned prophecy controller with a concrete plant to obtain an explicit controller for the given plant. This is possible if the given plant is contained in the learned prophecies. The algorithm for the composition of prophecy controller and plant is presented in Algo. 3.

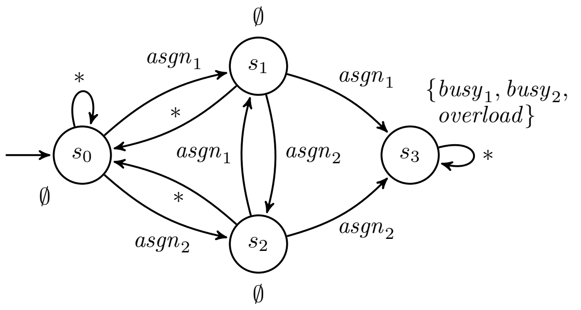

Figure 4: A plant for the load balancer example that only signals $busy$ when processors are overloaded. We use $*$ to represent edges that are taken whenever no other edge guard is satisfied.

The algorithm implements an on-the-fly exploration of the state space of the composition of prophecy controller and plant, and only adds transitions that satisfy the prophecy annotations. For every visited state and every possible output, it checks whether the prophecy annotation is satisfied by the plant. If satisfied, the output is added as a label to the current state, and transitions to the next states are added for each possible input of the controller. Once all states have been explored, the algorithm returns the explicit controller.

Note that checking whether the prophecy is satisfied by a plant is done by model checking the corresponding CTL formula against the plant, which can be done in polynomial time (Clarke, Emerson, and Sistla 1986). This allows us to efficiently check the prophecies and compose the controller with the plant on-the-fly.

5.3 Synthesis via Learning and Refinement

Although Algo. 3 can be used to synthesize an explicit controller for a given plant using a learned prophecy controller, its correctness is not immediately guaranteed: If the plant is not contained in the learned prophecies, the constructed controller might not be correct. To overcome this limitation, we add an additional model-checking step to verify whether the synthesized controller is correct for the given plant. If it is, we return the synthesized controller, otherwise, we refine the approximation using this plant and repeat the learning process to obtain a new prophecy controller that includes this plant. This overall synthesis procedure is summarized in Algo. 4 and its correctness is shown in Thm. 1 and Cor. 1.

Algorithm 4: Synthesis via Learning and Refinement

```
Input: An approximation $\mathcal{W}$, its learned prophecy controller
      $\mathcal{U} = (\mathcal{A}, \kappa)$ with $\mathcal{A} = (Q, q_0, \delta, \Omega)$, and a plant $\mathcal{M}_\mathrm{p}$.
Output: A correct controller $\mathcal{M}_\mathrm{c}$ for $\mathcal{M}_\mathrm{p}$.
let synthesize($\mathcal{W}$,$\mathcal{U}$, $\mathcal{M}_\mathrm{p}$) :=
  $\mathcal{M}_\mathrm{c} \leftarrow$ compose($\mathcal{U}, \mathcal{M}_\mathrm{p}$) // Compose controller
  if $Runs(\mathcal{A}, q_0, \mathcal{M}_\mathrm{p} \parallel \mathcal{M}_\mathrm{c}) \subseteq \Omega$ then // Check correctness
    return $\mathcal{M}_\mathrm{c}$ // Return correct controller
  else
    $\mathcal{U} \leftarrow$ learnApprox($\mathcal{W}, \{\mathcal{M}_\mathrm{p}\}$) // Refine and re–learn
    return compose($\mathcal{U}, \mathcal{M}_\mathrm{p}$) // Return new controller
```

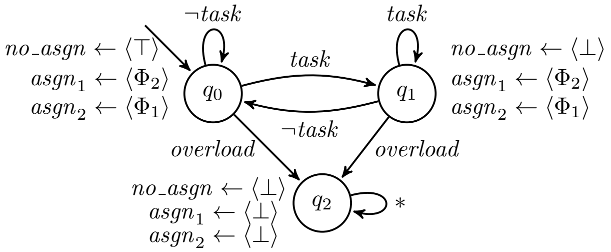
$\neg task$
$task$
$no\_asgn \leftarrow \langle\top\rangle$
$asgn_1 \leftarrow \langle\Phi_2\rangle$
$asgn_2 \leftarrow \langle\Phi_1\rangle$
q_0
$task$
q_1
$no\_asgn \leftarrow \langle\bot\rangle$
$asgn_1 \leftarrow \langle\Phi_2\rangle$
$asgn_2 \leftarrow \langle\Phi_1\rangle$
$\neg task$
$overload$
$overload$
$no\_asgn \leftarrow \langle\bot\rangle$
$asgn_1 \leftarrow \langle\bot\rangle$
$asgn_2 \leftarrow \langle\bot\rangle$
q_2
$*$

Figure 5: A refined prophecy controller for the load balancer example, where the CTL prophecy Φ_i is given by $\forall\bigcirc(overload \Rightarrow asgn_i)$ for $i \in \{1, 2\}$.

Corollary 2. *Given an approximation $\mathcal{W}$ and its learned prophecy controller $\mathcal{U}$ for a safety LTL specification φ, and a plant $\mathcal{M}_p$, the procedure synthesize($\mathcal{W}$,$\mathcal{U}$, $\mathcal{M}_p$) returns a controller $\mathcal{M}_c$ such that $\mathcal{M}_p \parallel \mathcal{M}_c \vDash \varphi$.*

Example 1. *To illustrate the overall synthesis procedure, we revisit the example in Sec. 1.1. We consider the specification φ in* (1) *and th nominal plant $\mathcal{M}_p$ in Fig. 1. By using the procedure* learnApprox($\mathcal{W}, \{\mathcal{M}_p\}$) *on this plant with an initial (coarse) approximation, we obtain the prophecy controller $\mathcal{U}$ in Fig. 2, where prophecies are represented as CTL formulas. As discussed in Sec. 1.1, this prophecy controller is not only correct for the plant $\mathcal{M}_p$ but also generalizes to larger plants that have sufficient CPU availability and signal $busy_i$ whenever cpu_i is busy.*

Suppose we obtain a new plant that does not signal the busy *status in the same way, e.g., a plant that only signals busy once the CPUs are overloaded. For instance, consider the plant $\mathcal{M}_p'$ as in Fig. 4. If we use the procedure* compose($\mathcal{U}, \mathcal{M}_p'$) *to synthesize a controller for this plant, we obtain an explicit controller that is not correct for this plant, as the prophecies are based on the assumption that the plant signals busy whenever a CPU is busy. Hence, we need to refine the approximation using this plant and re-learn the prophecy controller by calling* learnApprox($\mathcal{W}, \{\mathcal{M}_p'\}$).

Fig. 5 shows the refined prophecy controller after learning the new plant. Here, at states q_0 and q_1, the prophecy annotations for outputs $asgn_1$ and $asgn_2$ are represented by the CTL formulas $\Phi_i := \forall\bigcirc(overload \Rightarrow asgn_i)$, respectively, stating that if there is an overload in the next step, then the controller must have assigned the task to cpu_i. The controller can only assign a task to a CPU if it is guaranteed that the overload in the next step can only be caused by the other CPU. Therefore, the refined prophecy controller is correct for the new plant $\mathcal{M}_p'$, and can be used to synthesize controllers for plants that implement similar behavior.

6 Experimental Evaluation

We implemented the algorithms presented in Secs. 4 and 5 in an F#-based prototype tool called UCLEARN (Nayak et al. 2025) to assess the effectiveness of our method. The prototype utilizes SPOT (Duret-Lutz et al. 2022) for LTL and automata operations, OINK (van Dijk 2018) for game solving,

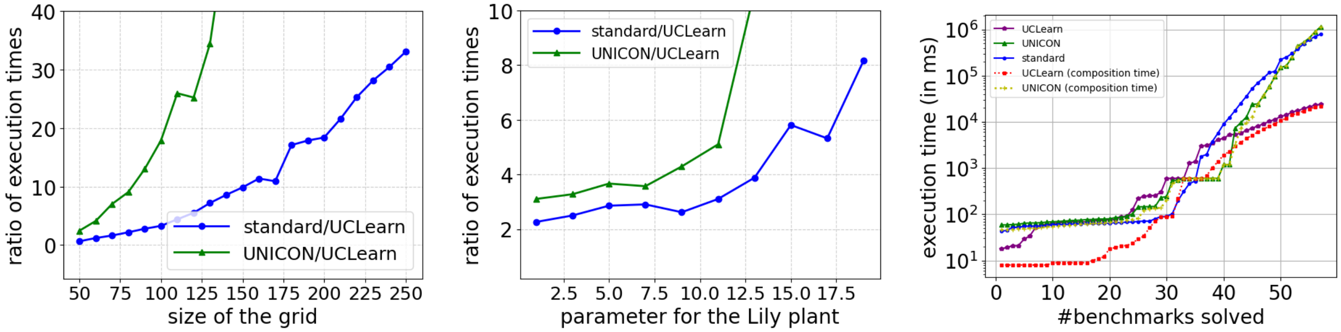

Figure 6: Experimental results showing scalability on the grid world (left) and lily (middle), and overall comparison (right).

and (Bordais, Neider, and Roy 2024) for CTL learning. Additionally, we use UNICON (Finkbeiner et al. 2025a) to compare our approach with the USC synthesis algorithm and the standard reactive synthesis algorithm.

As a baseline, we tested the running example from Sec. 1.1 and Ex. 1 using UCLearn, and the resulting prophecy controller with CTL prophecies are shown in Figs. 2 and 5. This example illustrates that the learned prophecies are both simple and interpretable.

We further evaluated UCLearn using a diverse set of benchmarks, including standard reactive synthesis benchmarks from SYNTCOMP (Jacobs et al. 2022) and a set of benchmarks based on a robot grid world, which are detailed below. All experiments were performed on an Apple M1 Pro 8-core CPU and 16GB of RAM. The presented execution times are averages over three runs.

Scalability in Grid World. In this benchmark, the goal is to control a robot navigating an $n \times n$ grid. The plant encodes the grid structures with obstacles and the robot's position. The robot can move in four directions (up, down, left, right), and the controller's task is to ensure that the robot can move in the grid while avoiding obstacles, encoded as a simple LTL formula: $\square \neg \mathit{collision}$. We evaluate the performance of each approach on increasing grid sizes, and show the ratio of execution times between the existing approaches and UCLearn in Fig. 6 (left). The results indicate that our approach is much more scalable and over an order of magnitude faster than the other approaches. This is not surprising as UNICON requires capturing all realizations of the plant, while UCLearn only needs to capture the realizations that are similar to a given set of plants. Furthermore, it is worth noting that the approximation was learned from a single plant with grid size 2 and consists of CTL formulas of size at most 2. This demonstrates that our approach is capable of learning a suitable approximation from a small benchmark and that the resulting formulas are human-readable.

Scalability in Lily. This benchmark, taken from SYNTCOMP (Jacobs et al. 2022), requires the controller to grant or cancel a request from a user within 3 time steps and ensure that no two requests are granted consecutively. This is encoded as the LTL formula: $\square(\mathit{request} \rightarrow \bigcirc(\mathit{grant} \vee \mathit{cancel} \vee \bigcirc(\mathit{grant} \vee \mathit{cancel} \vee \bigcirc(\mathit{grant} \vee \mathit{cancel})))) \wedge \square(\mathit{grant} \rightarrow \bigcirc \neg \mathit{grant})$. We evaluate all approaches on this benchmark with plants encoding increasing complexity of requests, and the results are shown in Fig. 6 (middle). As with the grid world benchmark, the approximation is learned from a single plant with parameter 2, and the learning approach is up to 8 times faster than the standard approach.

Adaptability in SYNTCOMP. In addition to demonstrating the scalability of our approach, we also evaluate its adaptability to different plants. We use the benchmarks from the previous evaluation and a set of safety benchmarks from SYNTCOMP, each consisting of an assumption and a guarantee. For such benchmarks, we obtain a plant satisfying the assumption and, afterward, learn an approximation for the guarantee. This learned approximation is then used to obtain a controller for a plant of similar size that satisfies the assumption. The results of this evaluation are shown in Fig. 6 (right). The plot also compares the time required to compose the (already learned/constructed) prophecy controller with the plant, labeled as the composition time, for both UCLearn and UNICON. These results indicate that UCLearn adapts to changes in the plant model much faster than both the standard approach and UNICON. This is due to the fact that the CTL formulas learned from the plant are concise and small (with a size of at most 4), making them easily reusable for similar plants.

7 Conclusion

In this paper, we introduced a new method for universal controller synthesis, where the controllers are initially created independently of specific plants and are then refined for a given set of nominal plants in a subsequent step. Universal controllers offer multiple advantages over standard controller synthesis approaches, including scalability, adaptability, and explainability. We have proposed approximate universal controllers, which use prophecies tailored to a particular class of plants. The prophecy approximations are represented as CTL formulas and learned from positive and negative examples that are gathered during the explicit construction of the controller. Our experiments demonstrate that our approach outperforms existing universal controller synthesis methods whenever the learned prophecies are adequate for building an explicit solution.

Acknowledgments

This work is funded by the DFG grant 389792660 as part of TRR 248 - CPEC, by the European Union with ERC Grant HYPER (No. 101055412), and by the Emmy Noether Grant SCHM 3541/1-1. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

Abadi, M.; and Lamport, L. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.*, 82(2): 253–284.

Anand, A.; Mallik, K.; Nayak, S. P.; and Schmuck, A. 2023. Computing Adequately Permissive Assumptions for Synthesis. In Sankaranarayanan, S.; and Sharygina, N., eds., *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, 211–228. Springer.

Anand, A.; Nayak, S. P.; and Schmuck, A. 2023. Synthesizing Permissive Winning Strategy Templates for Parity Games. In Enea, C.; and Lal, A., eds., *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I*, volume 13964 of *Lecture Notes in Computer Science*, 436–458. Springer.

Baier, C.; and Katoen, J.-P. 2008. *Principles of model checking*. MIT press.

Balachander, M.; Filiot, E.; and Raskin, J. 2023. LTL Reactive Synthesis with a Few Hints. In *TACAS (2)*, volume 13994 of *Lecture Notes in Computer Science*, 309–328. Springer.

Basset, N.; Raskin, J.; and Sankur, O. 2017. Admissible Strategies in Timed Games. In Aceto, L.; Bacci, G.; Bacci, G.; Ingólfsdóttir, A.; Legay, A.; and Mardare, R., eds., *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, volume 10460 of *Lecture Notes in Computer Science*, 403–425. Springer.

Belta, C.; Yordanov, B.; and Gol, E. A. 2017. *Formal methods for discrete-time dynamical systems*, volume 89. Springer.

Bernet, J.; Janin, D.; and Walukiewicz, I. 2002. Permissive strategies: from parity games to safety games. *RAIRO Theor. Informatics Appl.*, 36(3): 261–275.

Berwanger, D. 2007. Admissibility in Infinite Games. In Thomas, W.; and Weil, P., eds., *STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, February 22-24, 2007, Proceedings*, volume 4393 of *Lecture Notes in Computer Science*, 188–199. Springer.

Beutner, R.; and Finkbeiner, B. 2022. Prophecy Variables for Hyperproperty Verification. In *35th IEEE Computer Security Foundations Symposium, CSF 2022, Haifa, Israel, August 7-10, 2022*, 471–485. IEEE.

Bordais, B.; Neider, D.; and Roy, R. 2024. Learning Branching-Time Properties in CTL and ATL via Constraint Solving. In *FM (1)*, volume 14933 of *Lecture Notes in Computer Science*, 304–323. Springer.

Bouyer, P.; Markey, N.; Olschewski, J.; and Ummels, M. 2011. Measuring Permissiveness in Parity Games: Mean-Payoff Parity Games Revisited. In Bultan, T.; and Hsiung, P., eds., *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*, 135–149. Springer.

Brenguier, R.; Raskin, J.; and Sankur, O. 2017. Assume-admissible synthesis. *Acta Informatica*, 54(1): 41–83.

Cassandras, C. G.; and Lafortune, S. 2021. *Introduction to Discrete Event Systems*. Springer Nature.

Clarke, E. M.; and Emerson, E. A. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In Kozen, D., ed., *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, 52–71. Springer.

Clarke, E. M.; Emerson, E. A.; and Sistla, A. P. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2): 244–263.

Damm, W.; and Finkbeiner, B. 2014. Automatic Compositional Synthesis of Distributed Systems. In Jones, C. B.; Pihlajasaari, P.; and Sun, J., eds., *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, 179–193. Springer.

Duret-Lutz, A.; Renault, E.; Colange, M.; Renkin, F.; Aisse, A. G.; Schlehuber-Caissier, P.; Medioni, T.; Martin, A.; Dubois, J.; Gillard, C.; and Lauko, H. 2022. From Spot 2.0 to Spot 2.10: What's New? In *International Conference on Computer Aided Verification, CAV 2022*, volume 13372 of *Lecture Notes in Computer Science*. Springer.

Finkbeiner, B.; Metzger, N.; Nayak, S. P.; and Schmuck, A. 2025a. Synthesis of Universal Safety Controllers. In Gurfinkel, A.; and Heule, M., eds., *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II*, volume 15697 of *Lecture Notes in Computer Science*, 177–197. Springer.

Finkbeiner, B.; Metzger, N.; Nayak, S. P.; and Schmuck, A.-K. 2025b. Universal Safety Controllers with Learned Prophecies. arXiv:2511.11390.

Finkbeiner, B.; and Passing, N. 2022. Synthesizing Dominant Strategies for Liveness. In Dawar, A.; and Guruswami, V., eds., *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2022, December 18-20, 2022, IIT Madras,* *Chennai, India*, volume 250 of *LIPIcs*, 37:1–37:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Fremont, D. J.; and Seshia, S. A. 2018. Reactive Control Improvisation. In Chockler, H.; and Weissenbacher, G., eds., *Computer Aided Verification*, 307–326. Cham: Springer International Publishing. ISBN 978-3-319-96145-3.

Grädel, E.; Thomas, W.; and Wilke, T., eds. 2002. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer. ISBN 3-540-00388-6.

Jacobs, S.; Pérez, G. A.; Abraham, R.; Bruyère, V.; Cadilhac, M.; Colange, M.; Delfosse, C.; van Dijk, T.; Duret-Lutz, A.; Faymonville, P.; Finkbeiner, B.; Khalimov, A.; Klein, F.; Luttenberger, M.; Meyer, K. J.; Michaud, T.; Pommellet, A.; Renkin, F.; Schlehuber-Caissier, P.; Sakr, M.; Sickert, S.; Staquet, G.; Tamines, C.; Tentrup, L.; and Walker, A. 2022. The Reactive Synthesis Competition (SYNTCOMP): 2018-2021. *CoRR*, abs/2206.00251.

Klein, J.; Baier, C.; and Klüppelholz, S. 2015. Compositional construction of most general controllers. *Acta Informatica*, 52(4-5): 443–482.

Kretínský, J.; Meggendorfer, T.; Prokop, M.; and Zarkhah, A. 2025. SemML: Enhancing Automata-Theoretic LTL Synthesis with Machine Learning. In *TACAS (1)*, volume 15696 of *Lecture Notes in Computer Science*, 233–253. Springer.

Latvala, T. 2003. Efficient Model Checking of Safety Properties. In Ball, T.; and Rajamani, S. K., eds., *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, volume 2648 of *Lecture Notes in Computer Science*, 74–88. Springer.

Nayak, S. P.; Metzger, N.; Schmuck, A.-K.; and Finkbeiner, B. 2025. Artifact for "Universal Safety Controllers with Learned Prophecies" at AAAI 2026.

Neider, D.; and Gavran, I. 2018. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, 1–10. IEEE.

Pnueli, A. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, 46–57. IEEE Computer Society.

Pommellet, A.; Stan, D.; and Scatton, S. 2024. Sat-based learning of computation tree logic. In *International Joint Conference on Automated Reasoning*, 366–385. Springer.

Raha, R.; Roy, R.; Fijalkow, N.; Neider, D.; and Pérez, G. A. 2023. Synthesizing efficiently monitorable formulas in metric temporal logic. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 264–288. Springer.

Tabuada, P. 2009. *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media.

Valizadeh, M.; Fijalkow, N.; and Berger, M. 2024. Ltl learning on gpus. In *International Conference on Computer Aided Verification*, 209–231. Springer.

van Dijk, T. 2018. Oink: An Implementation and Evaluation of Modern Parity Game Solvers. In Beyer, D.; and Huisman, M., eds., *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, 291–308. Springer.

Yin, X.; Gao, B.; and Yu, X. 2024. Formal synthesis of controllers for safety-critical autonomous systems: Developments and challenges. *Annual Reviews in Control*, 57: 100940.