Streamlined Integration of GR(1) Synthesis and Reinforcement Learning for Optimizing Critical Cyber-Physical Systems

1st Eric Wete
Institute for Data Science
Leibniz University Hannover
Hannover, Germany
eric.roslin.wete.poaka@stud.uni-hannover.de

2nd Joel Greenyer
Department of Software Engineering
Fachhochschule für die Wirtschaft Hannover
Hannover, Germany
joel.greenyer@fhdw.de

3rd Tom Yaacov
Department of Computer Science
Ben-Gurion University of the Negev
Be'er Sheva, Israel
tomya@post.bgu.ac.il

4th Daniel Kudenko
L3S Research Center
Leibniz University Hannover
Hannover, Germany
kudenko@l3s.de or

5th Wolfgang Nejdl
L3S Research Center
Leibniz University Hannover
Hannover, Germany
nejdl@l3s.de

***Abstract*—Cyber-physical systems (CPSs) must meet safety-critical requirements and optimization goals. Reactive controller synthesis (RCS) from temporal logic (TL) specifications can automatically construct agent behaviors that ensure safety and liveness. However, RCS struggles with time and probability aspects. Reinforcement Learning (RL) can optimize system behaviors, but cannot guarantee correctness. RL and RCS can be combined by shielding an agent with a synthesized controller, ensuring that the agent behavior satisfies the TL specification. However, this requires aligning different models: the TL specification and the often hand-crafted RL training environment code. We propose a three-step method, centered around code generation and manual refinement, for streamlining this alignment: (1) Formalize discrete environment assumptions and guarantees into a Spectra TL specification. (2) From it, automatically generate Gymnasium-compatible training environment code. For the target code, we employ BPpy, which enables a unique one-to-one mapping of the Spectra properties to individual code modules. This allows users to refine time and probability aspects as well as optimization goals without misaligning the models. (3) Synthesize a permissive controller from the TL specification and integrate it as a shield for the agent. In contrast to previous work, our approach (a) guarantees not only safety but also liveness; (b) even if the system is too complex for RCS, we can leverage the generated BPpy code as a model-at-runtime that shields the agent from unsafe actions.**

***Index Terms*—reactive synthesis, behavioral programming, reinforcement learning, model-at-runtime**

I. Introduction

Designing cyber-physical systems (CPS) is challenging [1]. Especially, robotic systems must meet safety-critical requirements, such as avoiding collisions and not harming humans. Moreover, CPS must satisfy optimization goals regarding resource-, energy-, and time consumption.

Automatic synthesis of reactive system controllers from temporal logic (TL) specification (i.e., LTL: linear temporal logic, *Reactive Controller Synthesis*, *RCS*) [2] can automatically produce controllers from formalized requirements (correct-by-construction). For GR(1), a fragment of LTL, efficient symbolic algorithms for controller synthesis exist [3], [4]. A GR(1) specification comprises environment assumptions and system guarantees that can contain initial properties, forbidden constraints (*safety* properties), and conditions that must occur infinitely often (*justice* properties). This is sufficient for specifying a broad range of systems. Tools like Spectra [5] and `slugs` [6] can perform controller synthesis and offer useful analysis features like *realizability checking* (checking the existence of a correct controller) or localize properties that cause inconsistencies.

The optimization of CPS, however, typically needs to account for the timing of processes, such as movement times or task durations, as well as the probabilities of events, such as the likelihood of failures or varying inter-arrival times of items in a production system. RCS is less suited for ensuring optimization goals in a timed and probabilistic setting. This is because tools like Spectra do not support these properties at all, or because including them makes the synthesis problem significantly more complex [7]–[9].

Reinforcement Learning (RL), on the other hand, can successfully train agents that optimize their behavior in timed and probabilistic settings, but cannot guarantee correctness.

Related work combines synthesis with planning [10] to exploit the advantages of RCS, but also pursue optimization goals. RCS can be combined with RL via an approach called *shielding*. Shielding employs a synthesized controller to keep the RL agent from taking unsafe actions [11]–[15].

Shielding approaches, however, critically depend on maintaining the alignment between two different models at different abstraction levels: the TL specification and the code of the agent's training environment. Developing a suitable training environment that accurately captures the CPS' environment dynamics, including accurate time and probability assumptions, can be a complex task on its own. Ensuring the alignment with a TL specification requires additional validation, which can be time-consuming, thus complicating the successful application of shielding.

In this paper, we propose a streamlined approach that aids in aligning the RCS and RL models by directly translating GR(1) specifications, specifically those written in the Spectra language [5], into executable Python code that realizes a Gymnasium-compatible training environment for RL. The translation that we propose ensures that individual Spectra properties are mapped one-to-one to separate code modules. This allows training environment engineers to refine the code with timing and probability aspects without risking misalignment with the Spectra specification.

The approach consists of three steps, illustrated in Fig. 1:

(1) *Formalize the CPS system requirements into a TL specification and synthesize a permissive controller that can be used for shielding an agent.* In this step, engineers create a Spectra specification of the discrete CPS behavior, abstracting from timing and probability aspects. The Spectra tools help, for example, to detect and remove unnecessary assumptions [16], which can reduce the complexity of the synthesis task. Realizability checking and counter-strategy-based debugging [17] can help engineers understand and eliminate specification inconsistencies. In this step, engineers maintain an affordable abstraction level to capture all critical requirements, without over-specifying.

(2) *Map the TL specification to executable training environment code and refine that code with timing and probability aspects as well as optimization goals.* We compile the Spectra specification to Python code based on BPpy [18], a behavioral programming (BP) [19] framework for Python. BPpy offers a unique way to provide an operational semantics to Spectra: it allows us to encode each Spectra property in a BPpy *b-thread*. These b-threads are executed concurrently in lock-step, ensuring that all agent-vs-training environment executions satisfy the assumptions as well as the guarantees of the Spectra specification.

The Spectra-to-BPpy mapping is realized by defining mapping rules for central Spectra idioms and concepts like the Dwyer patterns [20] that are supported by Spectra [21].

Engineers can extend the b-threads to add timing and probability aspects without fearing misalignment with the Spectra specification. Moreover, engineers can express optimization goals by adding reward signals to b-threads. For example, rewards could be dispensed each time that a robot delivers an item.

While this approach prevents misalignment of the specification and training environment code, engineers are not prevented from all wrongdoings. There is a risk of introducing inconsistencies if, for example, there are two Spectra properties constraining the same phenomenon and engineers introduce different deadlines to the corresponding b-threads. With awareness, such issues can mostly be avoided, but some validation remains necessary.

(3) *Train and deploy the agent shielded with the synthesized controller.* We train the agent in the training environment based on a maskable RL algorithm (maskable PPO [22]) that constrains the actions that the algorithm can explore in a state. We define a masking function that queries the synthesized controller for all permitted actions in the current state. To achieve this, we developed a web service that wraps the synthesized controller and a Java-based controller execution algorithm supplied by Spectra tools. This service can be called from the masking function. Overall, this architecture allows the agent to explore and optimize behaviors that satisfy the specified guarantees in any environment that satisfies the refined assumptions.

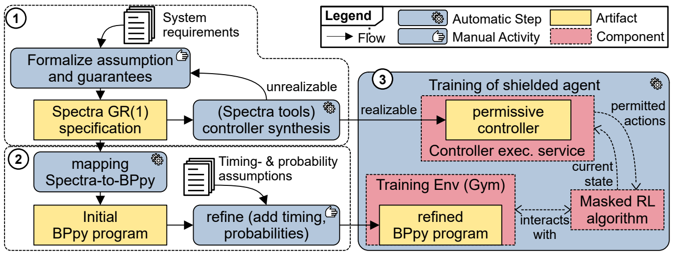

Fig. 1. Overview of streamlined RCS/RL integration

We evaluate the approach using a pick-and-place robot cell with varying parameters. In a setting where items arrive more frequently in one location, the robot learns to visit this location more frequently and ahead of time. We also observe that controller-guided agents learn faster than unguided agents.

One critique of RCS-based approaches is that synthesis may be infeasible for complex systems. However, our approach even has its merits in this case. Instead of shielding the agent by a synthesized controller, the generated BPpy code can be used as a model-at-runtime: since the code also executes b-threads directly derived from TL specification guarantees, they can block the agent from taking forbidden actions—*thus shielding the agent even without the synthesized controller.* While this enforces safety guarantees, the liveness guarantees (properties that occur infinitely often) can no longer be formally ensured as they depend on the synthesized controller. Instead, smart look-ahead execution (smart play-out [23]) could still ensure liveness with a high probability.

Structure: Section II and Sect. III introduce a motivating use case and the background. We describe the formal specification approach in Sect. IV. Section V explains the Spectra-to-BPpy mapping and how to refine time and probability aspects. Section VI describes the integration of the BPpy program with RL algorithms. We discuss evaluation results in Sect. VII, related work in Sect. VIII, and conclude in Sect. IX.

II. Example Use Case

We consider a pick-and-place robot cell in which items arrive at source locations, and a robot must pick up these items and deliver them to target locations, see Fig. 2.

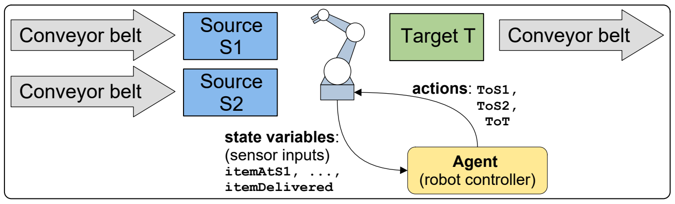

Fig. 2. A pick-and-place robot system

A software controller (agent) can control the robot by three actions that send it to one of the three locations (`ToS1`, `ToS2`, `ToT`). The agent has several input variables read from sensors:

- location (`loc`): The robot can be at one of the three locations (`AtS1`, `AtS2`, `AtT`)
- `itemAtS1`: Is there an item at location S1?
- `itemAtS2`: Is there an item at location S2?
- `itemPickedUp`: Does the robot carry an item?
- `itemDelivered`: True transiently in a state where the robot drops an item onto the target location.

The system must satisfy the following simple **guarantee**: *Periodically, the robot must deliver items to the target location*, i.e., `itemDelivered` must be true repeatedly.

The **assumptions** are:

1) *Initially, the robot is at the target location.*
2) *If the agent orders the robot to go to a location, the robot will arrive at that location (after some movement time), unless the agent orders going to another location.*
3) *Items periodically arrive at the source locations (sources have individual mean arrival times).*
4) *Items remain at source until the robot arrives to pick up.*
5) *An item can only be delivered if the robot is at the target.*
6) *If the robot arrives at the target location, it drops off the item (the item will no longer be picked up).*
7) *A picked-up item stays picked up until delivered.*

The assumptions include specific movement times of the robot arm to get from one location to another. Moreover, we assume probabilistic inter-arrival times of items at S1 and S2. The key *optimization goal* of the system is to increase the item delivery speed. If items arrive much faster in one location than the other, a beneficial strategy for the robot is to move to that location ahead of time in order to increase the delivery speed.

III. Preliminaries

A. Linear Temporal Logic

Linear Temporal Logic (LTL) [24] is a logic for specifying properties of infinite sequences of states evolving over time. LTL is widely used to specify the behaviors of systems and software components. LTL [24] is specified over a finite set of atomic propositions AP, which are Boolean statements that assume a truth value `true` ($\top$) or `false` ($\bot$) in every state. LTL uses the fundamental logical operators $\vee$ (`or`) and $\neg$ (`not`), as well as the temporal modal operators `X` (`next`) and `U` (`until`) [3].

An LTL formula is constructed from atomic propositions using the LTL syntax defined recursively in Eq. (1):

$$\phi \;:=\; p \,|\, \neg\,\phi \,|\, \phi \vee \phi \,|\, \mathtt{X}\,\phi \,|\, \phi\,\mathtt{U}\,\phi \;, \tag{1}$$

where p denotes an atomic proposition.

An LTL formula is interpreted over an infinite sequence of states $s = s_0, s_1, \ldots$, also called a *trace* or *execution*, where a *state* s_i defines the set of atomic propositions that hold in the state at the i-th position of the execution s. That is, $s_i \in \mathscr{P}$ and $s = s_0 s_1 \ldots \in \mathscr{P}^\omega$ for $\mathscr{P} = 2^{AP}$.

Let ϕ a formula, then $s,i \models \phi$ denotes that ϕ holds in state s_i of execution s. The semantics of the basic operators is defined as follows:

- $\top = p \vee \neg\, p$
- $\bot = \neg\,\top$
- $s,i \models p$ iff $p \in s_i$
- $s,i \models \neg\,\phi$ iff $s,i \not\models \phi$
- $s,i \models \phi_1 \vee \phi_2$ iff $s,i \models \phi_1$ or $s,i \models \phi_2$
- $s,i \models \mathtt{X}\,\phi$ iff $s,i+1 \models \phi$
- $s,i \models \phi_1\,\mathtt{U}\,\phi_2$ iff $\exists j \geq i : s,j \models \phi_2$ and $\forall k, i \leq k < j : u,k \models \phi_1$

The formula ϕ *satisfies* a computation s, formally specified as $s \models \phi$, if and only if $s,0 \models \phi$.

LTL defines additional future temporal operators `F` (finally) with $\mathtt{F}\,\phi \equiv \top\,\mathtt{U}\,\phi$ and `G` (globally) with $\mathtt{G}\,\phi \equiv \neg\mathtt{F}\,\neg\phi$.

PastLTL [25] introduces past operators `H` (historically) and `S` (since), where $\mathtt{H}\,\phi \equiv \neg(\top\,\mathtt{S}\,\neg\phi)$ and the semantics of `S` is:

- $s,i \models \phi_1\,\mathtt{S}\,\phi_2$ iff $\exists j \leq i : s,j \models \phi_2$ and $\forall k, j < k \leq i : s,k \models \phi_1$

(ϕ_2 held at some point in the past and ϕ_1 has held since then.)

B. Büchi Automaton

A Büchi automaton (BA) [26] is a tuple $B = (S, \Sigma, I, T, F)$, where S is a finite set of states, Σ is an alphabet of symbols, $I \in S$ is a set of initial states, $T \subseteq S \times \Sigma \times S$ is a set of transitions, and $F \subseteq S$ is a set of accepting states. A BA accepts an infinite sequence of symbols Σ^ω if there exists a corresponding path (matching the transition labels) from an initial state that visits accepting states infinitely often. If we set $\Sigma = 2^{AP}$, then BAs can accept or reject executions as defined above; hence, BAs can specify linear temporal properties over execution. Spectra uses BAs to encode Dwyer patterns [21], and our approach relies on the same principle for encoding patterns in BPpy.

C. Reactive Systems, GR(1), and Spectra

In reactive system specifications, we assume that a system is described by variables that are partitioned into *system variables* (controlled by the system) and *environment variables* (uncontrolled by the system, i.e., controlled by the environment). Atomic propositions are conditions over these variables.

An *execution* of the system is now a sequence of states where the system can assign values to system variables and the environment can assign values to environment variables.

Reactive controller synthesis (RCS) is the problem of finding a controller that can, for each state, always assign system variables in such a way that the resulting execution satisfies a given LTL specification, regardless of how the environment assigns environment variables. A specification is *realizable* iff such a controller exists.

The complexity of RCS for LTL is double-exponential in the specification size [2]. Due to this impractical complexity, an LTL subset was introduced that offers sufficient expressive power for many cases, but for which synthesis is more efficient: *Generalized reactivity of rank 1* (GR(1)) [3], [27].

A GR(1) specification takes the form in Eq. (2) [5], [27]:

$$\varphi^{sr} = (\theta^e \rightarrow \theta^s) \wedge (\theta^e \rightarrow \mathtt{G}((\mathtt{H}\rho^e) \rightarrow \rho^s)) \wedge \\ \left(\theta^e \wedge \mathtt{G}\rho^e \rightarrow \left(\bigwedge_{i=1}^{n} \mathtt{GF}\,\mathscr{J}_i^e \rightarrow \bigwedge_{j=1}^{m} \mathtt{GF}\,\mathscr{J}_j^s\right)\right) , \quad (2)$$

where:

- θ^e = an assertion over the initial environment states
- θ^s = an assertion over the initial system states
- ρ^e = the transition relation of the environment
- ρ^s = the transition relation of the system
- $\mathscr{J}_i^e$ = a justice assumption, i.e., an assertion over the specification variables to hold infinitely often for the environment
- $\mathscr{J}_j^s$ = a justice guarantee, i.e., an assertion over the specification variables to hold infinitely often for the system

SPECTRA [5] is a high-level specification language for reactive systems with the expressive power of GR(1). A SPECTRA specification comprises assumptions and guarantees over controllable or uncontrollable variables. Assumptions specify the environment behavior, whereas guarantees are properties that the system must satisfy. Both assumptions and guarantees can contain initial conditions, conditions that must hold in all states (*safety* or *invariant* properties), and conditions that must occur infinitely often (*justice* properties).

Synthesis and realizability checking require solving infinite two-player games on a game graph. The run-time complexity for this is polynomial in the size of the game graph [3], [28], which is exponential in the number of variables. There exist tools like the SPECTRA tools[1] that implement symbolic algorithms (based on BDDs: Binary Decision Diagrams) and avoid explicit construction of the game graph. This makes realizability checking efficient. Even symbolic representations of controllers can be synthesized, so-called *just-in-time* controllers [4], which often make below-exponential run-times possible for synthesis.

The SPECTRA language supports specifications over different variables with finite domains (boolean, enumerations, bounded integers, fixed-length arrays). Initial conditions are defined with the keyword `ini`, safety properties with the keyword `alw` (always), and justice properties with `alwEv` (always eventually); they are prefixed with `gar` for guarantees and `asm` for assumptions.

D. Behavioral Programming

Behavioral Programming (BP) [19] is a modern paradigm for modeling reactive systems from a set of modules that capture separate behavioral aspects of the system. BP is suitable for the specification of reactive systems that define the environment and system behaviors. A BP program (*b-program*) comprises independent modules called *b-threads* that run concurrently. A b-thread defines a behavior with some sequential control flow that may repeatedly reach *synchronization points*, where the b-thread specifies three types of *event sets*: *requested*, *blocked*, and *waited-for* events.

During execution, all b-threads execute their logic until they reach the next synchronization point. Once all b-threads have reached that synchronization point, a central event selection mechanism selects a single event that is requested by at least one b-thread and not blocked by any b-thread. Once an event is selected, all b-threads requesting or waiting for the event are notified and resume their execution. This process repeats until no more events are requested and not blocked. It is possible for the b-program to deadlock if all requested events are blocked.

E. BPPY: BP in Python

BPPY is a Python implementation of the BP paradigm [18], where b-threads function as co-routines [29], [30]. BPPY supports Satisfiability Modulo Theories (SMT)-based events [31], where events are defined as assignments over SMT variables. SMT expressions thus define variable value assignments that the b-threads may request, block, or wait for. The event selection strategy employs a solver (Z3) that selects a variable value assignment that satisfies at least one requested expression and none of the blocking expressions.

Listing 1 shows b-threads that implement a partial behavior of a robot cell example. A b-thread is a Python generator with the decorator `thread` preceding its declaration. The `yield sync()` calls are the synchronization points that can declare the three BP event sets (`request`, `block`, `waitFor`), where each event set is described by an SMT expression. The first b-thread (`init`) requests a variable assignment in which an item is picked up but not delivered, the robot is at location S1, and the selected action is sending the robot to the target location. The second b-thread requests that the location is `atT` after the selected robot action is `toT`. The b-thread does this by first waiting for an event where `act == toT`.

The execution of this b-program starts all b-threads. Since the second b-thread waits for the condition requested by the first b-thread, both b-threads advance on this synchronization point simultaneously. While the first b-thread now terminates, the second b-thread requests `loc == atT` next. The last b-thread never advances and continuously forbids `itemPickedUp` when the robot is at the target.

IV. FORMAL REACTIVE SYSTEM SPECIFICATION

Step (1) of our approach (see Fig. 1) is to formalize the system specification using SPECTRA. First, engineers identify

[1]https://github.com/SpectraSynthesizer

```
Action, (toS1, toS2, toT) \
    = EnumSort('Action', ('ToS1', 'ToS2', 'ToT'))
act = Const('act', Action)
itemPickedUp = Bool('itemPickedUp')
...
@thread
def init():
    yield sync(request=And(itemPickedUp,
    Not(itemDelivered), loc == atS1, act == toT))
@thread
def on_to_target_move_to_target():
    while True:
        yield sync(waitFor=(act == toT))
        yield sync(request=(loc == atT))
@thread
def at_target_implies_not_item_picked_up():
    yield sync(block=And(loc==atT, itemPickedUp))
```

Listing 1. A set of b-threads in BPPY

the system/environment boundary and determine the (sensor) inputs of the system as well as the (actuator) outputs. These are modeled as environment resp. system variables. Additional variables may be introduced if required to model additional state aspects. Then engineers specify the behavioral properties.

Listing 2 presents an excerpt of the specification[2] (see full specification in [32]) that formalizes the informal guarantees and assumptions given in Sect. II. Line 1 imports a pattern

```
1  import "SupplementalPatterns.spectra"
2
3  spec TwoSourcesRobot
4
5  sys {ToS1, ToS2, ToT} act;
6  env {AtS1, AtS2, AtT} loc;
7  env boolean itemAtS1;
8  env boolean itemAtS2;
9  env boolean itemPickedUp;
10 env boolean itemDelivered;
11
12 gar alwEv itemDelivered;
13
14 asm robot_is_initially_at_target:
15   ini loc=AtT;
16 asm items_periodically_arrive_at_S1:
17   alwEv itemAtS1;
18 ...
19 asm robot_arrives_at_S1_unless_ToS1_cancelled:
20   S_responds_to_P_unless_R_globally(loc=AtS1,act=ToS1,
       act!=ToS1);
21 ...
22 asm alw_items_stay_at_S1_until_robot_arrives:
23   Globally_P_implies_Q_Weak_Until_R(
24     itemAtS1 and !itemPickedUp,
25     loc!=AtS1 and itemAtS1,
26     loc=AtS1 and !itemAtS1 and itemPickedUp);
27 ...
28 asm itemPickedUp_means_not_AtT:
29   alw itemPickedUp implies loc!=AtT;
30 asm itemDelivered_only_atT:
31   alw itemDelivered implies loc=AtT;
32 asm itemPickedUp_stays_true_until_itemDelivered:
33   alw itemPickedUp implies
34     next(itemPickedUp) or next(itemDelivered);
```

Listing 2. SPECTRA specification of a pick-and-place robot cell

library. Lines 5-10 define the controllable system variables and the uncontrollable environment variables. SPECTRA supports the modeling of LTL specification patterns, especially the Dwyer patterns (Collection of property specification patterns) [21]. We apply two custom patterns: The first pattern, `S_responds_to_P_unless_R_globally`, is used to specify assumption (2) in lines 19-20. The pattern states that whenever condition p holds (second parameter), condition s (first parameter) must hold eventually, unless a third condition r holds (third parameter). In LTL, this can be written as $G(p \rightarrow F(s \vee r))$[3]. This pattern applies to all locations (S1, S2, and T). The second pattern, `Globally_P_implies_Q_Weak_Until_R` means that if p occurs, then q must hold until r holds, but r is not required to occur. In LTL, this can be written as $G(p \rightarrow (q\,W\,r))$[4]. This pattern is used to formalize that when an item arrives at a source location, it remains there until the robot arrives to pick it up (assumption (4), lines 22-26). This pattern applies to both source locations (S1 and S2).

Lines 16-17 leverage justice properties (`alwEv`) to specify that items arrive periodically at the source locations (here S1) see assumption (3). Lines 28-31 apply safety properties (`alw`) to formalize assumptions (5) and (6). Lines 32-34 formalize assumption (7); this property uses the `next`-state variable reference to constrain the transition relation. Lines 14-15 show an initial condition (`ini`) that models assumption (1). Finally, line 12 shows the guarantee defined in Sect. II.

Patterns can be specified by translating their Büchi automata (BA) representation to a SPECTRA property. Figure 3 shows the BA for $G(p \rightarrow (q\,W\,r))$, and Fig. 4 shows the BA for $G(p \rightarrow F(s \vee r))$.

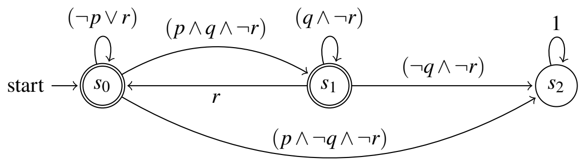

Fig. 3. The Büchi automaton for $G(p \rightarrow (q\,W\,r))$

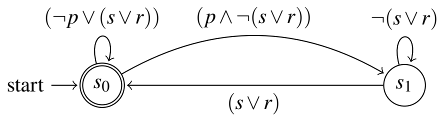

Fig. 4. The Büchi automaton for $G(p \rightarrow F(s \vee r))$

Lines 2-13 in Lst. 3 show the representation in SPECTRA of the BA for $G(p \rightarrow (q\,W\,r))$. The pattern defines state variables and an initial state via an `ini` condition. The transition relation is encoded via an invariant (`alw`) property, and the acceptance condition by an `alwEv` condition. Lines 16-24 similarly define the pattern of the BA for $G(p \rightarrow F(s \vee r))$.

The specification shown above is *realizable*: the interaction of the assumptions about the robot movement, the robot picking up the item, and that picked-up items are delivered when the robot reaches the target, force `itemDelivered` to occur periodically if the agent selects the appropriate actions.

[2]Artifacts available at https://doi.org/10.5281/zenodo.16338649.

[3]The pattern $G(p \rightarrow F(s \vee r))$ is equivalent to the *response* pattern by Dwyer, $G(p \rightarrow Fq)$, when $q = s \vee r$. We introduce this variant because s and r are handled differently in the timed refinement later, cf. Sect. V-C

[4]$\phi_1\,W\,\phi_2 \equiv (\phi_1\,U\,\phi_2) \vee G\,\phi_1$

```
// LTL: G(p -> (q W r))
pattern Globally_P_implies_Q_Weak_Until_R(p, q, r) {
  var { S0, S1, S2 } state;
  ini state=S0;
  alw ( (state=S0 & (!p | r) & next(state=S0)) |
        (state=S0 & (p & q & !r) & next(state=S1)) |
        (state=S0 & (p & !q & !r) & next(state=S2)) |
        (state=S1 & (r) & next(state=S0)) |
        (state=S1 & (q & !r) & next(state=S1)) |
        (state=S1 & (!q & !r) & next(state=S2)) |
        (state=S2 & (true) & next(state=S2))  );
  alwEv (state=S0 | state=S1);
}

// LTL: G(p -> F(s | r))
pattern S_responds_to_P_unless_R_globally(s, p, r) {
  var { S0, S1 } state;
  ini state=S0;
  alw ( (state=S0 & ((!p) | (p & (s | r))) & next(state=
     S0)) |
        (state=S0 & (p & !(s | r)) & next(state=S1)) |
        (state=S1 & (s | r) & next(state=S0)) |
        (state=S1 & (!(s | r)) & next(state=S1)));
  alwEv (state=S0);
}
```

Listing 3. SPECTRA patterns $G(p \to (qWr))$ and $G(p \to F(s \lor r))$

V. Mapping Spectra to BPpy

Step (2) of our approach is to map the SPECTRA specification to BPPY code and refine it with timing properties, probabilities, as well as optimization goals in the form of reward signals. We define a mapping for the central SPECTRA constructs, including Dwyer patterns [21] and other user-defined patterns. Patterns are mapped to b-threads that encode the corresponding BA, which can be constructed automatically from its LTL formulation [33].

A. Variables and Logical Operators

SPECTRA variables of type Boolean, enumeration, and bounded integer are mapped to Z3 types over which an SMT-based BPPY program can specify expressions, see Lst. 1 lines 1-4. Bounded integer variables are mapped to Z3 integer variables along with a b-thread that blocks values outside of the specified bounds. SPECTRA logical operators are translated to Z3 expressions: `not`, `and`, `or`, `implies` in SPECTRA correspond to `Not`, `And`, `Or`, `Implies` in Z3, respectively.

B. Mapping Assumptions

1) Initial Properties: The initial properties (`ini`) map to b-threads that request the initial condition in the first step and block its negation. See lines 1-4 in Lst. 4 corresponding to lines 14-15 in Lst. 2.

2) Invariant Properties: The invariant (safety) properties (`alw`) are mapped to b-threads that forever block the negation of the required property (cf. lines 5-8 in Lst. 4 mapped to lines 28-29 in Lst. 2). The invariant properties with `next`-state variable reference constrain the allowed transitions in SPECTRA. In BPPY, we relate the last-state variable values to the next-state variable values. Repeatedly, we first take the last-state values of all variables outside of the `next`-state references and then use them to block the negation of the invariant property. As an example, lines 9-16 in Lst. 4 show the mapping of the invariant property in lines 32-34 of Lst. 2.

```
1  @thread
2  def robot_is_initially_at_target():
3      cond = And(act == toT, wait_t == 0.0)
4      yield sync(request=cond, block=Not(cond))
5  @thread
6  def itemPickedUp_means_not_AtT():
7      cond = Implies(itemPickedUp, loc != atT)
8      yield sync(block=Not(cond))
9  @thread
10 def itemPickedUp_stays_true_until_itemDelivered():
11     e = yield sync(waitFor=true)
12     while True:
13         prev_itemPickedUp = e.eval(itemPickedUp)
14         e = yield sync(block=Not(Implies(
    prev_itemPickedUp,
15                  Or(itemPickedUp, itemDelivered))),
16             waitFor=true)
```

Listing 4. Mapping initial and safety properties

3) Justice Properties: A justice assumption requires a condition to hold repeatedly, but does not specify the time delay or regularity. To execute a timed/probabilistic environment simulation, the user refines this non-determinism by specifying either a constant delay or a probability distribution for the delay.

Lines 1-4 in Lst. 5 show the mapping for the justice property in lines 16-17 in Lst. 2. The property maps to a b-thread that calls the function `repeatedly_p`, which is a higher-order function that takes a desired condition to hold repeatedly as the parameter p. As the second parameter, the function takes an `interval` function that specifies the time delay between periods where p is true. Specifying the interval function is a *timed, probabilistic refinement* of the specification. Here, `exponential(2.0)` is a function that samples values from a negative exponential distribution with an average expected time of $\beta = 2.0$, i.e., the user has refined that items arrive at location S1 randomly, with a mean time of 2 seconds (assuming the unit is seconds) after the previous item was removed. The intervals can also be specified using uniform, normal, or other distributions, depending on the nature of the phenomenon.

The function `repeatedly_p` (lines 6-9 in Lst. 5) calls the function `p_true_after_time_unless_r`, forwarding the desired condition p as well as time delay values from invoking the `interval()` function.

The function `p_true_after_time_unless_r` ensures that the given condition p occurs exactly after the given time t unless a release condition r occurs before. It requests the condition p to occur exactly after the given deadline, but also allows non-p events to occur before. If a non-$(p \lor r)$-event happens (line 19), it recalculates the remaining time for the p-deadline (line 20) and requests p again.

When calling `p_true_after_time_unless_r` from `repeatedly_p`, there is no release condition r, so the default applies (`r=False`).

C. Mapping Patterns

SPECTRA patterns are mapped by translating their BA representations to BPPY code. Patterns can specify safety properties, which manifest in the BA with a non-accepting sink state, like state $s2$ for the pattern $G(p \to (q\,W\,r))$

```python
@thread
def items_periodically_arrive_at_S1():
    p = itemAtS1
    yield from repeatedly_p(p, exponential(2.0))

def repeatedly_p(p, interval):
    while True:
        yield sync(waitFor=Not(p))
        yield from p_true_after_time_unless_r(p,
     interval())

def p_true_after_time_unless_r(p, t, r=False):
    rem_t = t
    while True:
        e = yield sync(
            request=And(wait_t == rem_t, p),
            block=Or(wait_t < 0.0, wait_t > rem_t,
                And(wait_t < rem_t, p)),
            waitFor=True)
        if e.eval(Or(p,r)): break # p or r occurred
        rem_t -= z3_float_as_float(e.eval(wait_t))
    return e
```

Listing 5. Mapping of a justice assumption

shown in Fig. 3. Patterns can also specify liveness properties, like `S_responds_to_P_unless_R_globally`, $G(p \rightarrow F(s \vee r))$. The liveness component $(F...)$ leads to a non-accepting state in the BA on p, from which an accepting state can only be reached again if $s \vee r$ occurs, see the BA in Fig. 4.

The BAs are mapped to BPpy using the following rules:

- Transitions to non-accepting sink states are mapped to `block` conditions
- Transitions from non-accepting states are mapped to calls of the `p_true_after_time_unless_r` function introduced above (Lst. 5). If the transition is labeled with a disjunction of subconditions, like $(s \vee r)$, these can be handled differently depending on the *modality* with respect to time:
  - *a condition that must occur exactly after the given deadline*: is mapped to parameter p of the function `p_true_after_time_unless_r`.
  - *a "release" condition that may occur before the given deadline*: is mapped to parameter r of the function `p_true_after_time_unless_r`.
- All other transitions are mapped to `waitFor` conditions.

Listing 6 shows the mapping of the property in line 22 of Lst. 2 leveraging the pattern $G(p \rightarrow (q\,W\,r))$ (cf. BA in Fig. 3). Listing 7 shows the mapping for the property in line 19 of Lst. 2 based on the pattern $G(p \rightarrow F(s \vee r))$. s and r have a different modality—r is the release condition.

D. Mapping Guarantees

The guarantee initial conditions and invariants are mapped to BPpy in the same way as assumptions (Sect. V-A). Mapping the invariants is not necessary when the agent is shielded by a synthesized controller because it guarantees to satisfy these invariants. However, if the BPpy program is executed without a shielding controller, the invariant guarantee-b-threads can forbid the agent from selecting forbidden actions. (We outlined in the introduction how this could be useful for shielding the agent even if no controller can be synthesized.)

Justice properties represent desirable outcomes that should occur periodically and are used to specify rewards that model

```python
@thread
def alw_items_stay_at_S1_until_robot_arrives():
    p = And(itemAtS1, Not(itemPickedUp))
    q = And(loc!=atS1, itemAtS1)
    r = And(loc==atS1, Not(itemAtS1), itemPickedUp)
    yield from globally_p_implies_q_weak_until_r(p, q, r
    )

def globally_p_implies_q_weak_until_r(p, q, r):
    s0_s0 = Or(Not(p), r)
    s0_s1 = And(p, q, Not(r))
    s0_s2 = And(p, Not(q), Not(r))
    s0_wait_cond = Or(s0_s0, s0_s1)
    s1_s0 = r
    s1_s1 = And(q, Not(r))
    s1_s2 = And(Not(q), Not(r))
    s1_wait_cond = Or(s1_s0, s1_s1)

    class State(Enum):
        S0 = 0
        S1 = 1

    s = State.S0
    while True:
        if s == State.S0:
           e=yield sync(block=s0_s2,waitFor=s0_wait_cond
    )
           if e.eval(s0_s1): s = State.S1
        elif s == State.S1:
           e=yield sync(block=s1_s2,waitFor=s1_wait_cond
    )
           if e.eval(s1_s0): s = State.S0
```

Listing 6. Implementation of the pattern $G(p \rightarrow (qWr))$

```python
@thread
def robot_arrives_at_S1_unless_ToS1_cancelled():
    s = loc==atS1
    r = act!=toS1
    p = (act==toS1)
    yield from s_responds_to_p_globally_unless_r(s, p, r
    , normal(2.0,0.1))

def s_responds_to_p_globally_unless_r(s, p, r, interval)
    :
    s0_s0 = Or(Not(p), Or(s, p))
    s0_s1 = And(p, Not(Or(s, p)))
    s0_wait_cond = Or(s0_s0, s0_s1)
    s1_s0_request = s
    s1_s0_release = r

    class State(Enum):
        S0 = 0
        S1 = 1

    s = State.S0
    while True:
        if s == State.S0:
            e = yield sync(waitFor=s0_wait_cond)
            if e.eval(s0_s1): s = State.S1
        elif s == State.S1:
            yield from p_true_after_time_unless_r(
                s1_s0_request, interval(), s1_s0_release
    )
            s = State.S0
```

Listing 7. Implementation of the pattern $G(p \rightarrow F(s \vee r))$

the agent's optimization goals. Listing 8 illustrates how the justice guarantee in line 12 in Lst. 2 is mapped to a b-thread that calls the function `alw_reward_on_p`. This function computes and dispenses rewards (using BPpy idiom `localReward` for RL [18]) based on the occurrences of the desired property p and elapsed time. The parameter values for the function call are case-specific constraints. Here, `itemDelivered` is rewarded with 10 points, but passing time is penalized with reward -1 per time unit.

```
@thread
def items_delivered_periodically():
    yield from alw_reward_on_p(itemDelivered, 10.0,
     -1.0)

def alw_reward_on_p(p, p_reward, t_reward=0.0):
    reward = 0.0
    while True:
        e = yield sync(request=true, localReward=reward)
        reward = p_reward if is_true(e.eval(p)) else 0.0
        reward += t_reward * z3_float_as_float(e.eval(
     wait_t))
```

Listing 8. Mapping justice guarantees specifying rewards

E. Mapping non-deterministic choices

In Spectra, it is common to under-specify the environment behavior. Constraints on the transition relation (using `alw`) are usually encoded so that they leave non-deterministic choices for the environment. Also in justice properties or when using patterns, non-determinism is common.

However, for the operational behavior in BPpy, this non-determinism must be refined to deterministic or probabilistic choices. (Where non-determinism remains unrefined, the SMT solver produces choices unpredictably.)

In a hypothetical extension of our pick-and-place robot, we might specify that the robot may non-deterministically break items and fail to deliver them. In the BPpy program, we need to indicate that breaking items occurs with some (usually small) probability. To achieve this, the user has two options:

- *Add extra b-threads*: Users can refine these choice probabilities by implementing additional b-threads that block different choices with certain probabilities.
- *Specify pattern variants:* Users can specify pattern variants, like `S1_or_S2_responds_to_P_globally`, which accepts two conditions $s1$ and $s2$ as parameters. This variant can then be mapped to a BPpy function that takes a probability parameter to determine the likelihood of choosing $s1$ over $s2$ in response to p.

In our example, there is non-determinism, e.g., because the Spectra specification in Lst. 2 allows `itemPickedUp` to be true spontaneously even if no item has arrived. However, in the BPpy program, we do not assume that this variable becomes true spontaneously. Instead, we leverage the SMT solver to select the previous value unless specified otherwise. If a different behavior is required, the user must implement an additional b-thread to enforce specific constraints.

When refining the BPpy program, it is possible for users to introduce inconsistencies when different time constraints or probabilities are specified for the same phenomenon in different b-threads. This could result in inaccurate behavior or deadlocks. Hence, validation of the refinements is required, like any RL training environment implementation should be tested for its accurate reflection of the environment behavior.

VI. RL Integration with Gymnasium

BPpy programs can be integrated with the Gymnasium environment [34] (cf. [18]). The observation space comprises the variables constituting the state space. The action space is the set of all actions the system can take: all possible system variable assignments. If multiple system variables are involved, the action space expands to the cross-product of all possible value assignments for each variable. The execution of the controller synthesized from the Spectra specification is based on a controller executor component supplied by Spectra tools. This executor is implemented in Java and wrapped in an HTTP service to integrate with the RL framework (see Fig. 1).

The shielding of the agent is achieved by the process illustrated in Fig. 5. The agent learns based on a maskable RL algorithm, here, MaskablePPO [22], which interacts with a masking function.

Initialization: the agent calls `reset()` on the RL environment. This initializes a B-Program with an initial state (s_0) that is returned to the agent.

Step-cycle: Based on the returned state, the agent invokes the masking function (1) that calls the controller service (2) to retrieve the valid actions in the given state (3), from which the action mask is computed (4). Next, the agent selects an action and calls `step(action)` on the environment (Gymnasium) (5). The RL environment advances the B-Program based on the action (6) and returns the next state and reward signal (s_{t+1}, r_{t+1}) to the agent (7,8). The process is repeated until the B-Program terminates or the maximum step count for a training episode is reached. The interaction might also continue indefinitely if the agent is eventually deployed.

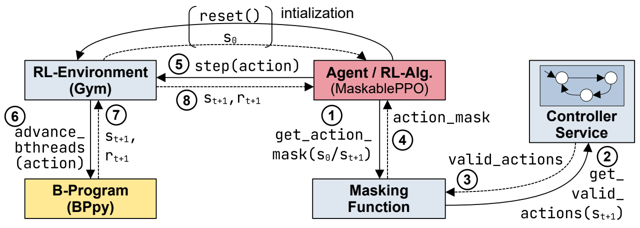

Fig. 5. Computing safe actions for the controller-guided agent

In an environment that satisfies the specified assumptions, the agent always follows the synthesized controller, which guarantees that the resulting behavior is correct w.r.t. the Spectra specification. The masking function may fail to map an environment state to a controller state if the environment does not satisfy the assumptions. This may be either because engineers introduced an error during the refinement, or because a real environment, in which the agent is deployed, does not behave as assumed. In this case, the strict formal interpretation of the Spectra specification would allow the agent to choose any action, because any behavior would satisfy the specification once the assumption (premise) is violated. However, in practice, it is advisable to implement an application-specific minimum-risk behavior; our pick-and-place would stop the movement of the robot.

We defined a reward function and applied its computed value in BPpy `sync` statements. For pick-and-place use cases, the reward function is defined as $r = c + \alpha \times t$, where: r is the reward when the goal is reached, i.e., an item is delivered, c is a constant that represents the initial reward, i.e., the maximal reward if the goal is achieved without elapsed time, α is the penalty per time unit, and t is the elapsed time from the last item delivery.

VII. Evaluation Results

We apply our approach to an industrial pick-and-place robot cell. The item inter-arrival times at `S1` and `S2` are $25s$ and $5s$. The robot movement times between locations are specified from uniform distributions in $[2.5, 3.5]s$, $[3, 3.5]s$, and $[3, 3.5]s$ for `S1`, `S2`, and `Target`, respectively. We apply different RL algorithms [35], [36] to train our RL agent: Advantage Actor-Critic (A2C), Deep Q-Network (DQN), Proximal Policy Optimization (PPO), and MaskablePPO that extends PPO with our function mask (see Fig. 5). The models were trained in 10000 steps. The step reward is $10 \times \beta + \alpha \times t$, where $\beta = 1$ if an item is delivered or 0 otherwise, $\alpha = -1$ is the penalty per time unit, and t is the elapsed time between steps. The learning rate is 1×10^{-4}, and the exploration rate decayed linearly from 1 to 0.05 with 0.1 decay fraction per step. See [32].

Table I shows the total reward, average value, and standard deviation for each RL algorithm over 10000 steps. It also shows the total number of items delivered and the delivery speed of the items per time unit.

TABLE I
Experiment results: reward and item delivery

	Reward			**Item delivery**	
Algorithm	**Total** ($\times 10^3$)	**Avg.**	**Std.**	**Total** ($\times 10^3$)	**Speed**
DQN	−12.8	−206.9	109.5	1.5	0.06
A2C	−5.1	−81.9	73.4	1.6	0.08
PPO	−1.1	−18.3	44.1	1.7	0.10
Mask.PPO	1.5	23.9	12.6	2.1	0.14

RQ1: *Does controller-guided RL speed up learning? Is the training of a controller-guided agent faster than that of an unguided agent?*

The controller-guided MaskablePPO algorithm outperforms the three other unguided algorithms; see the average rewards in Fig. 6.

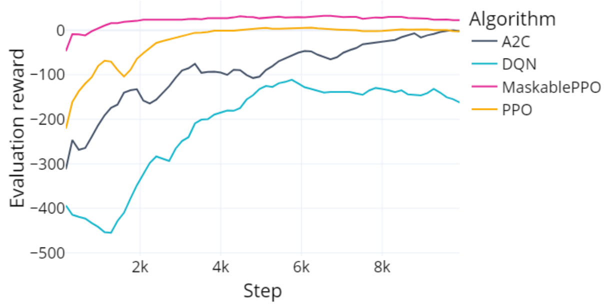

Fig. 6. RL model evaluation rewards

These results confirm our expectations. The agent learns to anticipate where items arrive more frequently. Additionally, the controller-guided agent has an advantage, as the controller prevents incorrect actions that an unguided agent explores. The controller speeds up learning and reduces the agent's exploration space. Unguided algorithms take longer to train and have higher variance.

RQ2: *How does RCS scale? What are the boundaries in which controller-guided RL is feasible?*

To evaluate the synthesis scalability, we measured synthesis times for settings with 2-5 robots, up to 20 source locations, and up to 3 target locations. For some cases, synthesis times can reach 5 minutes due to state-space explosion. (See Tab. II). Other evaluations of Spectra performance showed promising results [4], [5], [16]. One of the boundaries in applying controller-guided RL is thus the computational complexity inherent to RCS. We also discussed earlier that real-world environments may also violate specified assumptions. With our technique, assumption violations can be detected; we suggest to add case-specific minimum-risk behavior for such cases.

TABLE II
Reactive synthesis times under various problem settings

Robots	Sources	Destinations	Synthesis Time (s)
2	2	1	0.3
2	5	3	1.1
2	20	3	229.0
3	2	1	1.0
3	5	3	85.0
4	3	1	7.1
4	5	3	290.9
5	2	1	9.5

While the industrial pick-and-place example demonstrates the feasibility of our approach, it may not fully reflect the range of complexities present in broader real-world systems. However, we contend that it is reasonably representative and plan to study more examples in future work.

VIII. Related Work

There exist approaches that consider timed and probabilistic properties during controller synthesis. UPPAAL Stratego [37] can synthesize strategies in stochastic priced timed games. Ehlers et al. [6], [7] consider synthesis with timing and optimality aspects. Dräger et al. [41] investigate controller synthesis for systems that are probabilistic, real-time, and partially observable. PRISM can be used for the automated synthesis of optimal controllers for autonomous agents [42].

Since controller synthesis under consideration of timed and stochastic properties is hard, the idea is to combine discrete controller synthesis with RL via shielding [43]. Shielded RL is introduced by Alshiekh [11]. Koenighofer et al. [44] propose synthesizing shields from *safety* properties and handcrafted MDP abstraction of the environment. See also Odriozola et al. [45] for a review of shielded RL approaches. Koenighofer et al. [46] described a method for computing shields at runtime.

Our work considers discrete shields only and targets, in addition to safety guarantees, also liveness (justice) guarantees as well as the modeling of safety and liveness *assumptions*,

TABLE III
COMPARISON OF RELATED WORK ACROSS KEY CRITERIA

Paper	**Supported properties**	**Correctness Guarantee**	**RL**	**Timing Property Handling**	**Probability Handling**	**Controller synthesized from**
Ehlers et al. 2013 (Shortcut through an evil door) [7]	GR(1) (non-timed, non-stoch.)	Hard	No	Explicit delay cost in discrete time	No	LTL specification, GR(1)
David et al. 2015 (Uppaal Stratego) [37]	TCTL subset on stoch./priced Timed Game Automata	Hard	No	stochastic and timed games	stochastic and timed games	TGA + TCLT specification
Könighofer et al. 2020 (Shield Synthesis for RL) [38]	Safety (via LTL, PLTL, or Timed Automata)	Hard	Yes	Timed automata for timing guarantees	PLTL for probabilistic guarantees	Safety spec + handcrafted env. abstraction (MDP)
Hasanbeig 2020 (Cautious RL with logical constraints) [39]	LTL (via LDBA, non-timed, non-stoch.)	Soft (via rewards)	Yes	Not supported	During RL	no synthesis, online-tracking of LDBA
Wete et al. 2021 (MCTS and GR(1) synthesis) [10]	GR(1) (non-timed, non-stoch.)	Hard	No	MCTS optimization on timed env.-sim. (but non-timed spec)	MCTS optimization on env.-sim. (but non-prob. spec)	LTL Specification, GR(1) (Spectra)
Wete et al. 2024 (Controller-based safe RL) [40]	PCTL (checked on PRISM model)	Hard	Yes	Timed properties via PRISM	Stochastic properties via PRISM	given as manually modeled (verified) PRISM model
Brorholt et al. 2024 (Shielded RL for Hybrid Systems) [15]	Predicates over hybrid state variables	Soft (statistical)	Yes	stochastic hybrid automata	stochastic hybrid automata	finite-state hybrid abstraction (sampled) + safety predicates
This paper	GR(1) (non-timed, non-stoch.)	Hard	Yes	Env.-sim. refinement (derived from non-timed spec)	Env.-sim. refinement (derived from non-stoch. spec)	LTL specification, GR(1) (Spectra)

which makes it possible to capture the environment dynamics declaratively. Most importantly, our paper addresses a key challenge in the aforementioned work, which requires handcrafting an MDP to specify the environment. However, creating such an abstraction is a key challenge in practice. Our approach reverses this process by starting at a higher abstraction level. We suggest first specifying the critical discrete requirements and assumptions (safety and liveness properties), in a temporal logic-based SPECTRA specification. By ensuring that we can synthesize a controller (shield) from this specification, we ensure first and upfront that the agent's discrete design is realizable. From there, we offer a streamlined approach for refining, and eventually optimizing, timed and stochastic properties.

Brorholt et al. [15] propose a method to construct approximate safety shields for hybrid systems, using state predicates and sample-based abstractions of hybrid MDPs. Related is also the work of Hasanbeig et al. [39]. Instead of shielding, they translate an LTL specification into a Büchi automaton. During learning, the agent tracks its progress in the BA, receiving rewards that guide it toward accepting states.

Wete et al. [10], [47] consider the safe and optimized execution of a multi-robot system by executing controllers synthesized from SPECTRA GR(1) specifications with MCTS-based planning. Following work [40] compares optimization via planning vs. learning, where the agent is shielded by a controller that is designed manually (instead of being synthesized, as our approach) and for which probabilistic safety and liveness properties are verified by statistical model checking (via PRISM [48]).

RL-based execution methods for BP were also introduced by Yaacov [49] and Ashrov and Katz [50].

Selected related works are compared in Tab. III.

IX. CONCLUSION

This paper presents a novel approach for synthesizing correct and optimal controllers for systems in a timed and probabilistic environment. By combining SPECTRA's GR(1) specification and controller synthesis with RL, our work not only shields the agent from taking unsafe actions, but also guarantees that the agent satisfies liveness/justice guarantees.

In contrast to previous shielding approaches, our approach addresses the challenge of maintaining the alignment of models at different levels of abstraction for controller synthesis on the one hand and RL on the other.

A *core technical contribution* of our work is using BPPY for the training environment code, which allows for an encoding where each SPECTRA constraint is mapped into independent b-threads. Specifically, *we conceived a way to encode timing and probability in SMT-based behavioral programming*. This allows engineers to refine timing and probabilistic details while minimizing the risk of misaligning the involved models.

Our evaluation demonstrates the feasibility of this approach and even shows that controller-guided agents learn optimized behaviors faster while maintaining correctness guarantees.

In future work, we plan to expand the patterns and translation rules to cover other rich features of SPECTRA, such as triggers [51]. We are also investigating how to extend SPECTRA, so that time and probability annotations can be added already at the specification level.

References

[1] E. A. Lee, "Cyber Physical Systems: Design Challenges," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2008, pp. 363–369.

[2] A. Pnueli and R. Rosner, "On the Synthesis of a Reactive Module," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 179–190. [Online]. Available: https://doi.org/10.1145/75277.75293

[3] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) Designs," in *Verification, Model Checking, and Abstract Interpretation*, E. A. Emerson and K. S. Namjoshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–380.

[4] S. Maoz and I. Shevrin, "Just-in-time reactive synthesis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 635–646. [Online]. Available: https://doi.org/10.1145/3324884.3416557

[5] S. Maoz and J. O. Ringert, "Spectra: a specification language for reactive systems," *Software and Systems Modeling*, Apr 2021. [Online]. Available: https://doi.org/10.1007/s10270-021-00868-z

[6] R. Ehlers and V. Raman, "Slugs: Extensible GR(1) synthesis," in *International Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science (LNCS), vol. 9780. Springer, 2016, pp. 333–339.

[7] G. Jing, R. Ehlers, and H. Kress-Gazit, "Shortcut through an evil door: Optimality of correct-by-construction controllers in adversarial environments," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 4796–4802.

[8] K. W. Wong, R. Ehlers, and H. Kress-Gazit, "Resilient, provably-correct, and high-level robot behaviors," *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 936–952, 2018.

[9] C. Belta and S. Sadraddini, "Formal methods for control synthesis: An optimization perspective," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 2, pp. 115–140, 2019, first published as a Review in Advance on December 10, 2018. [Online]. Available: https://doi.org/10.1146/annurev-control-053018-023717

[10] E. Wete, J. Greenyer, A. Wortmann, O. Flegel, and M. Klein, "Monte Carlo Tree Search and GR(1) Synthesis for Robot Tasks Planning in Automotive Production Lines," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Oct 2021, pp. 320–330.

[11] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, "Safe reinforcement learning via shielding," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[12] N. Jansen, B. Könighofer, S. Junges, A. Serban, and R. Bloem, "Safe Reinforcement Learning Using Probabilistic Shields (Invited Paper)," in *31st International Conference on Concurrency Theory (CONCUR 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), I. Konnov and L. Kovács, Eds., vol. 171. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 3:1–3:16. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12815

[13] J. Riley, R. Calinescu, C. Paterson, D. Kudenko, and A. Banks, "Assured deep multi-agent reinforcement learning for safe robotic systems," in *Agents and Artificial Intelligence: 13th International Conference, ICAART 2021, Virtual Event, February 4–6, 2021, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 158–180. [Online]. Available: https://doi.org/10.1007/978-3-031-10161-8_8

[14] S. Bharadwaj, R. Bloem, R. Dimitrova, B. Konighofer, and U. Topcu, "Synthesis of minimum-cost shields for multi-agent systems," in *2019 American Control Conference (ACC)*, 2019, pp. 1048–1055.

[15] A. H. Brorholt, P. G. Jensen, K. G. Larsen, F. Lorber, and C. Schilling, "Shielded reinforcement learning for hybrid systems," in *Bridging the Gap Between AI and Reality*, B. Steffen, Ed. Cham: Springer Nature Switzerland, 2024, pp. 33–54.

[16] R. Shalom and S. Maoz, "Which of My Assumptions are Unnecessary for Realizability and Why Should I Care?" in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, May 2023, pp. 221–232.

[17] S. Maoz and Y. Sa'ar, "Counter play-out: executing unrealizable scenario-based specifications," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 242–251.

[18] T. Yaacov, G. Weiss, A. Ashrov, G. Katz, and J. Zisser, "Exploring and evaluating interplays of bppy with deep reinforcement learning and formal methods," in *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering*. SciTePress, 2025, pp. 27–40.

[19] D. Harel, A. Marron, and G. Weiss, "Behavioral programming," *Commun. ACM*, vol. 55, no. 7, p. 90–100, jul 2012. [Online]. Available: https://doi.org/10.1145/2209249.2209270

[20] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 411–420. [Online]. Available: https://doi.org/10.1145/302405.302672

[21] S. Maoz and J. O. Ringert, "Gr(1) synthesis for ltl specification patterns," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 96–106. [Online]. Available: https://doi.org/10.1145/2786805.2786824

[22] S. Huang and S. Ontañón, "A closer look at invalid action masking in policy gradient algorithms," *The International FLAIRS Conference Proceedings*, vol. 35, May 2022. [Online]. Available: https://journals.flvc.org/FLAIRS/article/view/130584

[23] D. Harel, H. Kugler, R. Marelly, and A. Pnueli, "Smart play-out," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 68–69. [Online]. Available: https://doi.org/10.1145/949344.949353

[24] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57.

[25] D. M. Gabbay, "The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems," in *Temporal Logic in Specification*. Berlin, Heidelberg: Springer-Verlag, 1987, p. 409–448.

[26] J. R. Büchi, *On a Decision Method in Restricted Second Order Arithmetic*. New York, NY: Springer New York, 1990, pp. 425–435. [Online]. Available: https://doi.org/10.1007/978-1-4613-8928-6_23

[27] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911 – 938, 2012, in Commemoration of Amir Pnueli. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000011000869

[28] K. Chatterjee, W. Dvorák, M. Henzinger, and V. Loitzenbauer, "Conditionally Optimal Algorithms for Generalized Büchi Games," in *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Faliszewski, A. Muscholl, and R. Niedermeier, Eds., vol. 58. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, pp. 25:1–25:15. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.MFCS.2016.25

[29] M. E. Conway, "Design of a separable transition-diagram compiler," *Commun. ACM*, vol. 6, no. 7, p. 396–408, jul 1963. [Online]. Available: https://doi.org/10.1145/366663.366704

[30] C. D. Marlin, *Coroutines: A Programming Methodology, a Language Design and an Implementation*, ser. Lecture Notes in Computer Science. Springer, 1980, vol. 95. [Online]. Available: https://doi.org/10.1007/3-540-10256-6

[31] D. Harel, G. Katz, A. Marron, A. Sadon, and G. Weiss, "Executing Scenario-Based Specification with Dynamic Generation of Rich Events," in *Model-Driven Engineering and Software Development*, S. Hammoudi, L. F. Pires, and B. Selić, Eds. Cham: Springer International Publishing, 2020, pp. 246–274.

[32] E. R. Wete Poaka, J. Greenyer, T. Yaacov, D. Kudenko, and W. Nejdl, "Streamlined Integration of GR(1) Synthesis and Reinforcement Learning for Optimizing Critical Cyber-Physical Systems ," Jul. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16338649

[33] P. Gastin and D. Oddoux, "Fast ltl to büchi automata translation," in *Computer Aided Verification*, G. Berry, H. Comon, and A. Finkel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 53–65.

[34] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel,

R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," Mar. 2023. [Online]. Available: https://zenodo.org/record/8127025

[35] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable Baselines3," https://github.com/DLR-RM/stable-baselines3, 2019.

[36] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[37] A. David, P. G. Jensen, K. G. Larsen, M. Mikučionis, and J. H. Taankvist, "Uppaal stratego," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 206–211.

[38] B. Könighofer, F. Lorber, N. Jansen, and R. Bloem, "Shield Synthesis for Reinforcement Learning," in *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2020, pp. 290–306.

[39] M. Hasanbeig, A. Abate, and D. Kroening, "Cautious reinforcement learning with logical constraints," in *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '20. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2020, p. 483–491.

[40] E. Wete, J. Greenyer, D. Kudenko, and W. Nejdl, "Multi-Robot Motion and Task Planning in Automotive Production Using Controller-based Safe Reinforcement Learning," in *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2024, Auckland, New Zealand, May 6-10, 2024*, M. Dastani, J. S. Sichman, N. Alechina, and V. Dignum, Eds. ACM, 2024, pp. 1928–1937. [Online]. Available: https://dl.acm.org/doi/10.5555/3635637.3663056

[41] K. Dräger, V. Forejt, M. Kwiatkowska, D. Parker, and M. Ujma, "Permissive controller synthesis for probabilistic systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 531–546.

[42] R. Giaquinta, R. Hoffmann, M. Ireland, A. Miller, and G. Norman, "Strategy synthesis for autonomous agents using prism," in *NASA Formal Methods*, A. Dutle, C. Muñoz, and A. Narkawicz, Eds. Cham: Springer International Publishing, 2018, pp. 220–236.

[43] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang, "Shield synthesis:," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 533–548.

[44] B. Könighofer, M. Alshiekh, R. Bloem, L. Humphrey, R. Könighofer, U. Topcu, and C. Wang, "Shield synthesis," *Formal Methods in System Design*, vol. 51, no. 2, pp. 332–361, Nov 2017. [Online]. Available: https://doi.org/10.1007/s10703-017-0276-9

[45] H. Odriozola-Olalde, M. Zamalloa, and N. Arana-Arexolaleiba, "Shielded reinforcement learning: A review of reactive methods for safe learning," in *2023 IEEE/SICE International Symposium on System Integration (SII)*, 2023, pp. 1–8.

[46] B. Könighofer, J. Rudolf, A. Palmisano, M. Tappler, and R. Bloem, "Online shielding for reinforcement learning," *Innovations in Systems and Software Engineering*, vol. 19, no. 4, pp. 379–394, December 2023. [Online]. Available: https://doi.org/10.1007/s11334-022-00480-4

[47] E. Wete, J. Greenyer, D. Kudenko, W. Nejdl, O. Flegel, and D. Eisner, "A Tool for the Automation of Efficient Multi-Robot Choreography Planning and Execution," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 37–41. [Online]. Available: https://doi.org/10.1145/3550356.3559090

[48] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[49] T. Yaacov, A. Elyasaf, and G. Weiss, "Keeping behavioral programs alive: Specifying and executing liveness requirements," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, 2024, pp. 91–102.

[50] A. Ashrov and G. Katz, "Enhancing deep learning with scenario-based override rules: A case study," in *Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering, MODELSWARD 2023, Lisbon, Portugal, February 19-21, 2023*, F. J. D. Mayo, L. F. Pires, and E. Seidewitz, Eds. SCITEPRESS, 2023, pp. 253–268. [Online]. Available: https://doi.org/10.5220/0011796600003402

[51] G. Amram, D. Ma'ayan, S. Maoz, O. Pistiner, and J. O. Ringert, "Triggers for Reactive Synthesis Specifications," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE Press, May 2023, pp. 729–741.