A Runtime Monitoring Framework to Enforce Invariants on Reinforcement Learning Agents Exploring Complex Environments

Piergiuseppe Mallozzi
Chalmers | University of Gothenburg, Sweden
mallozzi@chalmers.se

Ezequiel Castellano
National Institute of Informatics, Japan
The Graduate University for Advanced Studies, Japan
ecastellano@nii.ac.jp

Patrizio Pelliccione
Chalmers | University of Gothenburg, Sweden
Università degli Studi dell'Aquila, Italy
patrizio.pelliccione@gu.se

Gerardo Schneider
Chalmers | University of Gothenburg, Sweden
gerardo@cse.gu.se

Kenji Tei
Waseda University, Japan
ktei@aoni.waseda.jp

***Abstract*—Without prior knowledge of the environment, a software agent can learn to achieve a goal using machine learning. Model-free Reinforcement Learning (RL) can be used to make the agent explore the environment and learn to achieve its goal by trial and error. Discovering effective policies to achieve the goal in a complex environment is a major challenge for RL. Furthermore, in safety-critical applications, such as robotics, an unsafe action may cause catastrophic consequences in the agent or in the environment. In this paper, we present an approach that uses runtime monitoring to prevent the reinforcement learning agent to perform "wrong" actions and to exploit prior knowledge to smartly explore the environment. Each monitor is defined by a *property* that we want to enforce to the agent and a *context*. The monitors are orchestrated by a meta-monitor that activates and deactivates them dynamically according to the *context* in which the agent is learning. We have evaluated our approach by training the agent in randomly generated learning environments. Our results show that our approach blocks the agent from performing dangerous and safety-critical actions in all the generated environments. Besides, our approach helps the agent to achieve its goal faster by providing feedback and shaping its reward during learning.**

I. Introduction

Artificial intelligence is increasingly being used to solve problems in many different domains, such as robotics, where a software agent is trained to act autonomously in an, often, unknown environment. Reinforcement Learning (RL) [1] algorithms can be used to train a software agent: the agent learns a policy that maximizes a final reward by trying different actions on the environment (*trial and error*) and collecting rewards.

During training, at *learning time*, the agent can perform actions that are potentially dangerous to the environment or to itself. At *execution time* we cannot be sure that the agent will always act correctly since it uses probabilistic models to make decisions. By *safe exploration* [2] we refer to the problem of guaranteeing that an agent has to act safely both at learning and execution time. For example, a cleaning robot should learn to clean the dirt without breaking other elements in a room, or without harming the agent itself.

Runtime verification techniques can prevent the agent to perform catastrophic actions. Safety-critical requirements can be encoded in one or more monitors and enforced at learning and execution time when the monitor detects that the agent is about to violate them. On one hand we have to *convey goals* to the RL agent through the reward function, on the other hand we want the agent to respect some important properties that include safety-critical requirements, which we call *invariants*, at all time. The work in [3] goes in the direction of conveying the goals by building more structured reward functions, by modelling and verifying them at design-time. In this paper, we address the problem of preserving the invariants of a RL agent at learning and execution time with an approach called WiseML. In [4] we have presented a preliminary idea of WiseML, a method that combines model-free RL with runtime monitoring and enforcement, without providing a concrete solution. In this paper, we further develop the idea, provide a concrete solution, and largely validate it.

WiseML is agnostic with respect to the RL algorithms used to train the agent. We refer to WiseML as a *safety envelope*: it wraps the RL agent and prevents the execution of actions that would violate its invariants. The WiseML enhanced RL agent, which we call WiseML agent, analyses every action that the RL agent proposes and those that do not violate the specified invariants are sent to execution.

We express the invariants via the use of *specification patterns*, which based on the work in [5], [6], we use 4 patterns: *absence*, *universally*, *precedence*, and *response*. Once invariants are expressed in terms of patterns, then they can be automatically translated in Linear-time Temporal Logic (LTL) [7] or other logics thanks to the mappings provided in [5], [6]. We implement each invariant the RL agent has to obey with a monitor. We introduce also a concept of *context* that is similar to the concept of *scope* present in the specification patterns proposed by Dwyer et al. [5]. Each pattern has a scope, which defines in which part of the execution the pattern must hold. In WISEML we have a *meta-monitor* that dynamically activates and deactivates the individual monitors according to their context.

Blocking unsafe actions will prevent the agent from damaging the environment or itself. However, the agent can also *learn* what caused the violation in order to improve its policy in the future. The safety envelope includes a *reward shaping* component that influences the rewards received by the agent at runtime. This component penalizes the agent for attempting to violate specified invariants since the RL agent performs its decisions based on probabilistic models we can never be sure that the agent will never perform harmful actions during execution time. For this reason the *safety-envelope* is active also during execution time. However, in the current version, we have used WISEML only at learning time. In the future, we will also investigate and experiment during execution time.

The approach has been evaluated extending the *gym-minigrid* platform [8] with our environments and the WISEML *safety envelope*. We have evaluated the RL agent in *150 randomly generated* environments of different sizes, each executed 10 times with the presence of WISEML and 10 times without. Our results show that WISEML correctly enforces the invariants in all the simulated scenarios. Furthermore, thanks to the reward shaping feature, the agent learns much faster with the presence of WISEML. In fact, our experiments show that WISEML blocked violations 100% of the times while helping the agent converging up to 55% faster with respect to the learning performed without WISEML.

Summarizing, the main contributions of this paper are:

- WISEML, a framework that uses runtime monitoring to prevent wrong behaviours of an RL agent and to convey prior knowledge of the environment to the agent while it is exploring. The approach is completely independent of the RL algorithm chosen to train the agent.
- WISEML contributes shaping the rewards by using invariant violation as punishments for the RL agent.
- To facilitate the specification of invariants we enable the users to specify them via the use of specification patterns.
- An evaluation conducted on randomly generated environments; the agent has only *partial-observability* of the underlying state of the environment.

The paper is structured as follows. Section II gives an overview of the specification patterns, the reinforcement learning algorithm, and the runtime monitor techniques. Section III survey related works. Section IV presents our approach. Section V presents the case study performed on a gridworld environment, explains how the evaluation has been conducted and introduces the results analysed from the collected data. Finally in Section VII we discuss our results and future work.

II. BACKGROUND

A. Specification patterns

Capturing temporal properties in a concise and correct way is a major challenge [9], [10]. Syntactic correctness can easily be ensured through standard language processing techniques. However, guaranteeing that a property matches a software engineer's intuition is much harder.

Several lightweight specification languages have been proposed in the last years [10]–[12]. A different approach has been proposed by Dwyer et al. [5], which proposed qualitative property specification patterns in the late nineties. They analyzed a set of 555 specifications from at least 35 different sources in order to define a catalogue of eight qualitative specification patterns[1]. These specification patterns are organized in two major groups: *occurrence patterns* and *order patterns*. Occurrence patterns focus on a single event (or state) during system execution (e.g., absence or existence of an event). Order patterns capture relations of multiple events can emerge during system execution (e.g., response or precedence). Specification patterns are automatically translated to temporal logics and query languages, e.g., LTL and CTL [5].

Qualitative specification patterns have been extended to express real-time properties and the result is a catalogue of real-time specification patterns [13]. They have been also extended to express probabilistic quality requirements (e.g., reliability, availability, and performance requirements) and the resulting catalogue is known as probabilistic specification patterns [14]. Finally, the work in [6] presents a unified catalogue that collects the existing specification patterns and combines them together with 40 newly identified or extended patterns.

B. Reinforcement learning

In *reinforcement learning* [15] a software agent collects *observations* from the environment and performs actions. Each observation represents a *state* of the environment and as the agent moves from one state to another it collects a numerical *reward*. The goal of the agent is to maximize the reward collected along the way. The environment can be formally described as a *Markov Decision Process* (MDP) [16]. An MDP is a 5-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$. At each timestep t the agent interacts with the MDP by observing a state $s_t \in \mathcal{S}$ and by choosing an action $a_t \in \mathcal{A}$. The environment in response will transition to the next state s_{t+1} with probability $\mathcal{T}(s_t, a_t)$ and give a reward $r_t \sim \mathcal{R}(s_t, a_t)$. The goal of the agent is to maximize the *return* $G = \sum_{t=0}^{\infty} \gamma^t r_{t+1}$, which is the sum of all the *discounted* rewards, where $\gamma \in [0, 1]$ is known as the *discount factor*. The value of a state $V(s)$ represents how good is for the agent to be in the state s. Formally, it defines the expected sum of rewards from state s.

The agent will learn a *policy* or *value function* used to estimate the action to perform given a state of the environment. It does not learn a model of the environment and so it can not explicitly predict the effect of its action. Instead, it needs to gather actual experience by exploring the environment,

[1] http://patterns.projects.cis.ksu.edu/

which can make the exploration process dangerous. As the agent moves to a real-world environment, it has to respect some safety constraints. An action that violates some safety constraint can cause catastrophic consequences in both the agent and the environment.

C. Runtime verification

Runtime verification (RV) [17], [18] is a technique based on monitoring software executions. It detects violations of properties, occurring while the monitored program is running, eventually providing the possibility of reacting to the incorrect behaviour of the program whenever an error is detected.

Properties verified with RV are specified using any of the following approaches: (i) annotating the source code of the program under scrutiny with *assertions* [19]; (ii) using a high-level specification language [20]; or (iii) using an automaton-based specification language [21]–[23].

One way to verify properties at runtime is through the use of *monitors*. A monitor is a piece of software that runs in parallel to the program under scrutiny, controlling that the execution of the latter does not violate any of the properties. In addition, monitors may create a log file where they add entries reflecting the verdict obtained when a property is verified. In general, monitors are automatically generated from the annotated/specified properties [24], [25].

We will here consider the possibility of *monitoring* the execution of a program for different purposes. We may distinguish three different "kinds" of monitoring: (i) *proper monitoring*, where the monitor collects data, eventually performs simple side-effect free computations (e.g., calculate an average during a specific amount of time), sending the data to another device or monitor; (ii) *runtime verification* is concerned with verification of one or more properties about the expected behaviour of the system under monitoring; (iii) *runtime enforcement* is performed by monitors that carry the code to be executed in the monitored system, send specific commands to control the system, or enforce a given property (as mentioned above) not allowing the system to act differently from the specification.

Researchers usually talk about RV without distinguishing between the above three meanings. In this paper we will instead use the term "monitor" to refer to any of the above three specific uses, and we will clarify when confusion may arise (e.g., we might talk about an "enforcer" if we want to emphasize that the monitor is indeed enforcing a property).

III. Related Work

The literature on safe exploration has highlighted several directions to address the problem [26]–[28]. Thomas et al. [29] focus on ensuring safety with a policy improvement algorithm that provides probabilistic guarantees on the agent policy, given that the environment can be modelled as an MDP, or partially observable Markov decision process (POMDP) [15]. Lipton et al. [30] modified the DQN algorithm with the concept of *intrinsic fear* that shapes the rewards of the agent guiding it away from catastrophes. The agent interacts with the environment through an MDP and the intrinsic fear model is learned using the data collected from a finite sample of states.

Human Intervention RL (HIRL) is an approach by Saunder et al. [31] that uses human overseer to avoid catastrophes in model-free reinforcement learning agents. As the *safety-envelope* proposed by our approach WiseML, the human overseer stands between the agent and the environment and it can either let the agent's actions to be applied to the environment or block them. The decisions taken by the human are used to train a module, the *blocker*, via supervised learning. The main difference with our approach is that they used a trained model to block potentially dangerous actions at run-time. We use a *hand-coding* approach to specifying invariants at design-time. This comes with the trade-off of having less flexibility in terms of recognized violations compared with a supervised learning model, but more assurances in term of safety (since monitors are not based on machine learning models, will always block the violation as specified).

The work in [32] expresses the properties that the agent must satisfy in LTL and produces an MDP, which is the product of the original MDP and a Limit Deterministic Büchi Automaton (LDBA) generated from the LTL properties. In this work one needs to know the complete information about the environment which is modelled as an MDP with labelled *safe* and *unsafe* states. The agent explores only the *safe* parts using Q-learning to learn the optimal policy.

Al-Shedivat et al. [33] focuses on intelligent exploration of the RL agent on complex environment. They takes advantage of a hierarchical framework for RL [34] by training a meta-controller on learning the sequence of known subgoals while a low-level controller learn how achieve each subgoal.

Our approach builds on several of these ideas. We use LTL to easily encode prior knowledge into formal rules and reward shaping as a basic technique to help the agent to reach the goal. In real-world applications, since it is impractical or even impossible to precisely and completely model the environment, the agent partially observes the environment through its sensors. So in our work, the agent has partial observability of the environment. The agent is able to understand the underlying state of the environment by collecting multiple observations and integrating them over-time. This process is performed by *Long Short-Term Memory* (LSTM) [35], a particular kind of *Recurrent Neural Networks* [36] used as main deep learning model of the RL algorithm.

IV. WiseML

WiseML addresses the safe exploration problem of a RL agent, during and after training, in four main directions:

1) *modeling* invariants in terms of property specification patterns [5], [6];
2) *monitoring* the agent in different contexts as it performs actions freely in the environment. We enable the definition of invariants that should be checked and preserved in specific contexts of the environment. This is realized through the use of various monitors that are orchestrated by a meta-monitor.

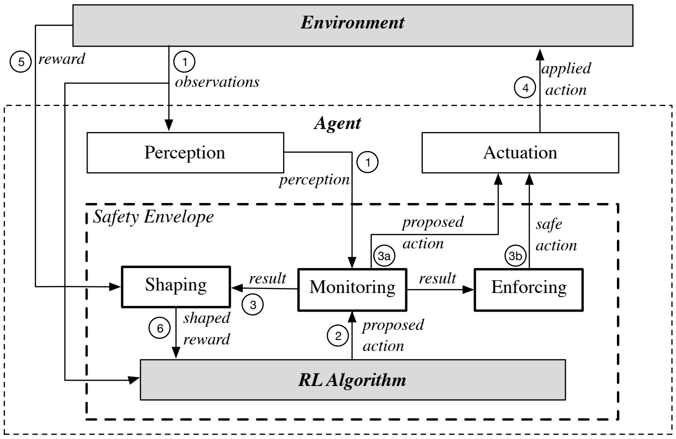


Fig. 1: Overall architecture of WiseML

3) *enforcing* a safe behaviour of the agent when it is about to violate the invariants;
4) *shaping* the reward of the agent so it learns to avoid future bad situations, converging faster to its goal.

Our approach consists of a *safety envelope* around the agent so that it is protected from performing dangerous actions for the environment or itself. Before a violation of any invariant is about to happen, the monitors stop the *unsafe* action from being executed on the environment. The agent still learns from its mistake as WiseML shapes its reward, meaning that the final reward coming from the environment is modified to take into account the blocked violation.

Figure 1 shows the main architecture of WiseML. Both the agent and the environment are unaware of the presence of WiseML. In this sense, our approach is agnostic to the RL algorithm used. The *Safety Envelope* surrounds the RL agent. The data between the safety envelope and the environment is processed by the *Perception* and *Actuation* components.

From a high-level perspective, WiseML works as follows. At first, the environment sends the observations of its current state to the RL agent. The perception component analyses raw observations from the environment and converts them in *perceptions*, more high-level representations of the world outside the agent (*1*). The perceptions are used to model the invariants in the monitors. The monitoring component processes the perceptions and the proposed action by the RL agent. In this phase, the meta-monitor activates the monitors according to their *context* and checks the satisfaction of the monitored invariants. The results of this analysis are sent to the shaping and enforcing component (*3*). Each monitor contributes to computing the overall *shaped reward* (*6*) and the final action sent to the environment (*4*). This action can be either the same proposed by the agent (*3a*) or a *safe action* computed by the enforcing component (*3b*) according to the state of the monitors and their *operational mode*. Each monitor can be configured to be in *shaping* or *enforcing* operational mode. In shaping mode the monitor only influences the reward given back to the agent; in enforcing mode, besides the reward, it also affects the action proposed as explained in Section IV-C.

With WiseML we can model each invariant separately as one monitor to be activated in a specific *context*. The meta-monitor will dynamically trigger all the monitors that have the context matching with the current execution of the agent in the environment. A context can be a function of the agent's perceptions, actions or both. It can also indicate that the invariant holds in every situation, regardless of the context. Different monitors can be combined in order to model all the invariants of the agent. For each monitor, the designer can specify the *rewards* to be given to the RL agent in case of violation or compliance with the monitored invariant. WiseML provides a simple interface where the designer can specify all the monitors in a user-friendly `JSON` file. The designer has to specify a few parameters for each monitor in order to activate them as follows: (i) `monitor-name` (ii) `monitor-type`, (iii) `monitor-context`, (iv) `monitor-invariant`, (v) `monitor-operational-mode`, and (vi) `rewards`.

In the following, we will describe in more detail the main components of the safety envelope at the core of WiseML: *monitoring*, *shaping*, and *enforcing*.

A. Monitoring

The monitoring component continuously examines the status of the agenxt and of the environment and communicates the results of its analysis to the shaping and enforcing components. The analysis is performed by several monitors, one for each invariant that the agent should respect in a specific context.

Invariants can be expressed as temporal specifications of constraints and preferences, similarly to when specifying properties in Linear Time Logic (LTL) [7]. The objective is to verify that, under specific context, the conditions specified by the system designer are satisfied by the RL agent. A context C is a function $f(s_e)$, while a condition A can be expressed as a function $f(s_e, s_a, a_p)$ where:

- s_e is the current state of the environment (as perceived by the agent through the perception component);
- s_a is the current state of the agent;
- a_p is the action proposed by the agent to be executed on the environment.

The system designer can specify the context and the conditions using one of the available patterns. In the following, we describe the patterns supported by WiseML and the equivalent LTL formula, where A and B are the monitor conditions, and C is a context condition as described above. We have chosen the following patterns in order to capture the *occurrences* and the *order* of events and operations that can occur while the agent is training by exploring the environment:

- *Absence:* $C \implies \Box(!A)$.
  A is never true. If active, this type of monitor verify that the condition A is false at all times.
- *Universally:* $C \implies \Box(A)$.
  A is always true. If active, the universally monitor ensures that the condition A is true at each step taken by the agent.
- *Precedence:* $C \implies \Box(!B\mathcal{W}A)$.
  Globally, A precedes B. If active, if B is true then the monitor checks that A has become true in the past.

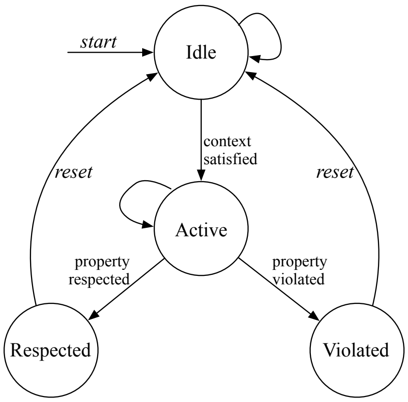


Fig. 2: Absence/Universally pattern

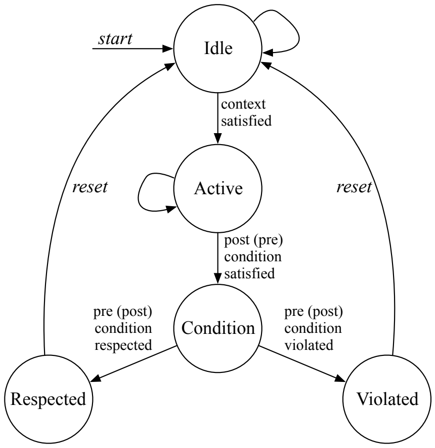


Fig. 3: Precedence/Response pattern

- *Response:* $C \implies \Box(A \rightarrow \Diamond B)$.
  Globally, B is eventually the response to A. If active, when A becomes true the monitor checks that B will become true as well.

Figure 2 shows the monitors associated with the absence and universally patterns. Whenever a monitor is in an active state it can check if the invariant is satisfied or violated. Figure 3 shows the monitors associated with the precedence and response patterns. For the precedence, the monitor is triggered by the post-condition and later checks violation of the pre-condition. For the response, the monitor is triggered by the pre-condition and later checks violation of the post-condition. All monitors are reset after the invariants have been checked. When there is no label in the transition a monitor maintains the same state.

B. Shaping

Reward shaping is a technique that allows modifying the rewards given to the agent in some states of the environment so as to help the RL agent to learn more accurately and converge to the goal faster. It provides more guidance to the agent as it explores the environment. However, one has to be careful on how to modify the rewards because this can lead to unexpected consequences. For example, if we only reward an agent for going in the right direction, the agent could learn to go in circles rather than reach the goal [37]. WiseML utilizes the rewards defined by the designer in order to shape the reward received by the agent during learning. Ultimately, the shaping component will reward the agent for respecting the invariants defined by the monitors and will punish it for violating them.

C. Enforcing

Monitors can be configured to act as *enforcers*. In this case, they block the action proposed by the agent from being executed in the environment if a violation of the monitored invariant is about to happen. If the violation is not safety-critical a monitor can be configured only to shape the reward in order to help the agent with extra domain knowledge. On the other hand, if the invariants modelled in a monitor are safety-critical, the monitor can enforce them by blocking the dangerous action and executing a safe one. In the current implementation of WiseML the developer can specify a particular action to enforce. If no safe-action is specified and a violation is about to happen the RL agent is asked to produce a new action that does not cause any violation.

Since several monitors can run in parallel, each modelling and possibly enforcing different invariants, the enforcing module has to issue an action that satisfies all invariants of the monitors. At each step, when the agent proposes an action, it is possible that more than one monitor can transition to a *violation* state. The monitoring component communicates the *unsafe actions* of each monitor to the enforcing component that executes the actions that the designer has specified to manage the violation of the invariant.

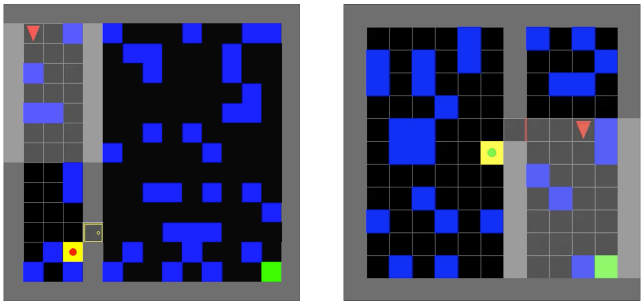
V. Evaluation

A. Gridworld Environment

In order to evaluate our approach, we have designed the WiseML framework and developed an extension of the *Minimalist Gridworld Environment* for OpenAI Gym [8] that supports the WiseML framework. The agent consists of a known-working RL implementation [38]. It is based on a variant of one of the latest RL algorithms developed by Google Deep Mind: *Asynchronous Advantage Actor-Critic* method (A3C) [39]. The algorithm used here is often referred to as A2C since it is a *synchronous* version of the A3C [40].

A gridworld environment consists of a two-dimensional grid of cells. The agent always occupies one cell of the grid facing one of the four adjacent cells. It can interact only with the cell it is currently facing or change direction inside its cell. At each step the agent can choose to perform one of the following actions: *move forward*, *turn left*, *turn right*, *toggle*, and *wait*. The action *toggle* can both open/close a door and turn on/off a light switch.

We have extended the existing grid by introducing new elements in order to evaluate some *safety-critical* scenarios with WiseML. Figure 4 shows some examples of randomly generated safety-critical environments. The agent is depicted as a red triangle (top-left corner). The green cell (bottom-right corner) is the agent *goal*, in all the environments the agent has to learn to navigate safely from the initial point of the grid to



(a) An initial configuration. The light is off and the door is closed.

(b) An intermediate step. The light is on and the door is opened.

Fig. 4: Examples of randomly generated environments.

the goal. The lighter cells around the agent represent its field of view, meaning the observations that agent perceives from the environment. These cells are perceived at each step and semantically analyzed by the perception component. The blue cells (randomly positioned in around all the grid) are *water* cells, if the agent steps on them it drowns and dies, specifically the RL algorithm terminates one episode and the agent starts again from the initial position. There are also more complex elements such as doors and a light switches (yellow tile with a red dot in the middle). By turning on the light-switch next to the door the agent is able to perceive observations in the other room, otherwise, its observations are altered and it can not see potential hazards such as the water. Before reaching the goal cell the agent has to learn to turn on the light by toggling the switch *before* entering a new room.

The environment generation randomly places the wall, door, and n water tiles. The minimum width of each room is two tiles. The light-switch is always placed next to the door. The position of the agent and the goal are fixed to ensure that the agent needs to traverse the entire room to reach the goal. The algorithm also validates that the generated environment has a solution by checking that the goal is reachable from the initial position (e.g., it checks that water cells are not blocking it).

The goal of the agent is to localize the goal position and step on it. However, when the light is off, due to the assumption that the sensors need a minimum amount of light to work, the agent is not able to perceive any observations. Hence, an implicit sub-goal is to turn on the light on before going back to the original goal of reaching its final position.

We have modelled the invariants of the RL agent as LTL properties using the patterns of WISEML. See below a list of some of the conditions used in the formulation of the properties. Each condition is triggered by the perception component of the agent.

Conditions used to detect the *context* of the agent (function of the agent's *perceptions*). (i) p_{wt}: the agent is near the water, (ii) p_{dr}: the agent detects a door in front, (iii) p_{do}: the agent detects that the door is open, (iv) p_{dc}: the agent detects that the door is closed, (v) p_{lw}: the agent detects light-switch, (vi) p_{lo}: the agent detects that the light is on, (vii), and (viii) p_{lf}: the agent detects that the light is off.

Conditions used to model the monitor invariants of the agent. These are situations that might trigger the monitor (functions of the agent perception and of the *action* proposed by the agent):

- a_{fwt}: the agent moves forward when facing water;
- a_{tgl}: the agent is about to toggle a light-switch.

From the above conditions, we have formulated the following properties, and easily model them as monitors in WISEML:

- (*Absence*) $p_{wt} \implies \Box(!a_{fwt})$. Always avoid to step on water.
- (*Universally*) $p_{lw} \implies \Box(p_{lo})$. The light is always on.
- (*Precedence*) $p_{dr} \implies \Box(!a_{frm}\mathcal{W}p_{lo})$. The light should have been turned on before entering a room.
- (*Response*) $p_{lw} \implies \Box(p_{lf} \rightarrow \Diamond a_{tgl})$. If a light switch is detected and the light is off. Enforced action: *toggle*
- (*Response*) $p_{dr} \implies \Box(p_{dc} \rightarrow \Diamond a_{tgl})$. If a door is detected and the door is closed. Enforced action: *toggle*.

In our experiment each invariant is *enforced* to the agent. It is important to highlight that for the response properties, we implemented the monitor to respond immediately to the pre-condition. Then the post-condition is enforced to happen in the next state. This is a valid implementation since we have the knowledge from the environment that, when the pre-condition is true, it is always possible to perform immediately the actions that satisfy the post-condition.

The developer can choose to specify a particular action to enforce in case of violation or let the agent propose a new suitable action. It is important to notice that the invariants that we have modelled are very simple requirements of the agent and are not related to the task that the agent has to achieve.

B. Evaluation

In order to evaluate WISEML, we have run the experiments in randomly generated environments of different sizes. Starting from a configuration file we generate the environment and the monitors inside WISEML and run the experiments inside a Docker container.

In each experiment, we have compared the performances of the RL agent with the *safety envelope* of WISEML with another RL agent that uses the exact same RL algorithm and configurations but it is no wrapped with the WISEML framework. For the rest of the paper, we will refer to the WISEML agent as the one with the *safety-envelope* and as CLASSICRL agent to the one without.

After modelling the invariants into WISEML, we have collected the results as follows. First we generate a random safety-critical environment `Env` of some size `N`. Then we launch the training of WISEML agent and later the CLASSICRL agent from scratch on `Env` until they converge and repeat the process for `M` iterations. Finally, we collect the results of these runs and we start again the training on a different random environment.

The environment is randomly generated regarding the number and position of the water tiles, and the position of the door, light-switch and wall. Regarding the *convergence* of the agents, we consider an agent to have terminated the training when its convergence conditions, described below, are met for

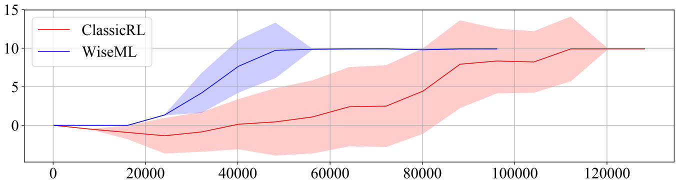


Fig. 5: Example of one experiment, comparing the convergence with and without WiseML. The Y-axis represents the reward accumulated at each episode by the RL agent. The X-axis represents the number of steps.

Size	Max. number of steps	Convergence (%)	
		WiseML	ClassicRL
7	490000	96.33	78.00
9	810000	91.33	47.67
11	1210000	80.67	33.00
13	1690000	55.33	9.00
15	2250000	66.00	2.00

TABLE I: Percentage of learning iterations that converged.

Size	Faster (%)	Catastrophes	
		WiseML	ClassicRL
7	45.96	0.00	4483.83
9	38.77	0.00	8301.93
11	41.64	0.00	5970.11
13	41.07	0.00	2663.88
15	54.92	0.00	3053.00

TABLE II: Comparison between the learning iterations that converged.

several consecutive episodes. An episode terminates when the agent reaches a terminal state, so when it reaches the goal, it dies (e.g. steps on the water) or it reaches a maximum number of steps which is proportional to the size of the grid. The algorithm converges if all the following conditions are satisfied: (i) the goal is achieved, (ii) the number of steps to the goal stabilizes, (iii) the value loss is less than 0.01, and (iv) the mean cumulative reward is positive.

The *value loss* measures the error between the predicted value of a state and the updated value after the reward has been received from the environment.

We have formulated the following research questions:

RQ1 *To what extent* WiseML *can assure the respect of invariants on a reinforcement learning agent using runtime monitoring?*

RQ2 *Can* WiseML *help the agent to converge to its goal faster by combining runtime monitoring with the use of reward shaping?*

To answer the research questions we have modelled the five invariants mentioned in this section as monitors in WiseML.

In our study, we have defined environments (grids) of sizes 7x7, 9x9, 11x11, 13x13, and 15x15. For each size, we have generated 30 random gridworld safety environments, and for each environment, we have performed 10 iterations of the WiseML and ClassicRL agent for a total of *3000 runs* of the RL algorithm until the convergence criteria are met, or the maximum number of steps is reached. Each iteration has a maximum of $size^2 * 10000$ time-steps. The number of water tiles was defined as 25% of the free tiles. The agent receives a positive reward of 10 when it reaches a goal and a negative reward of -10 for death, -0.1 for violations (only for the WiseML agent) and -0.005 for each step. The agent view of the environment is a grid of size 7x7. We have chosen such coefficient by empirically trying several values and noticing that the agent converged better with these ones. Generally, the worst state is the more negative is the reward.

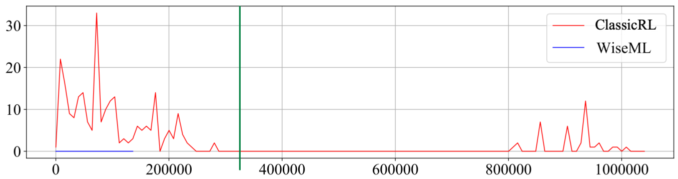


Fig. 6: Example of one experiment, showing the number of deaths accumulated over time. The ClassicRL agent can continue to die also after convergence (indicated with the vertical green line).

Table I shows the percentage of iterations in which the reinforcement learning converged before a predefined maximum number of steps. Table II shows a comparison between the WiseML and ClassicRL agents for the cases in which the reinforcement learning algorithm converged. The first column shows the average of the comparison in terms of average time-steps between the WiseML agent and ClassicRL agent on the same random environment. In some environments, it was not possible to do the comparison because the ClassicRL agent never converged for such environment. The second and third columns show the average number of catastrophes in each iteration for the WiseML and ClassicRL agent.

Our experiments show that the WiseML agent converges from 55% to 96% of the times, while the ClassicRL agent only converges between 2% to 78% of the times depending on the size of the random environment. Moreover, the WiseML agent is on average faster to converge than the ClassicRL agent. Also, the monitors have prevented catastrophic events to occur (i.e. stepping on the water).

Figure 5 and Figure 6 show an example of two experiments. Figure 5 shows the convergence of WiseML and the ClassicRL agents. It represents the total reward (mean and standard error) accumulated by each agent until convergence, averaged every 8000 steps. Figure 6 shows the number of deaths accumulated by the agents over one run; in particular, we show how the ClassicRL agent can keep dying also long after its convergence (indicated with the vertical green line). Obviously, assuming that the perception layer is perfect, the WiseML agent never died during all the experiments. All our results, including some videos, are available in the link below[2] and they are all reproducible by launching the same experiments via the original code on Github[3] or simply launching them via the Docker image[4].

To answer our research questions, our results show that with WiseML the agent never violates its modelled requirements.

[2] https://goo.gl/FzgEdo

[3] https://github.com/pierg/wiseml-patterns

[4] https://hub.docker.com/r/pmallozzi/wiseml-patterns

Indeed, there is the implicit assumption that the requirements should be correctly modelled using the specification patterns and that the perception component is working correctly. Furthermore, thanks to runtime monitoring and *reward shaping* the agent will converge faster while avoiding catastrophes.

VI. Conclusions and Future Work

We presented WiseML, an approach that uses runtime monitoring to prevent a RL agent from performing actions that can be dangerous to the environment or to itself. We specified *invariants* through the use of specification patterns, which are translated into *monitors* that block and enforce them. The use of WiseML during learning time improves also the learning of the RL agent. Our approach is agnostic with respect to the chosen RL algorithm and it assumes that the RL agent has no previous knowledge about the environment. Furthermore the agent only has a *partial-observability* of the environment, making it closer to *real-world* applications. We developed and evaluated our approach using one of the latest RL algorithms. We collected data from randomly generated environments and showed how the monitors always enforce their invariants while helping the RL agent to converge to its goal.

As future work, we plan to perform larger experimentation by using other RL algorithms and more complex environments. Moreover, we will investigate more systematic ways of identifying invariants. Finally, we plan to extend our approach by automatically synthesizing all the monitors from the invariants that we want to preserve.

VII. Acknowledgement

This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP).

References

[1] R. S. Sutton and a. G. Barto, "Reinforcement learning: an introduction." p. 1054, 2017.

[2] J. García and F. Fernández, "A Comprehensive Survey on Safe Reinforcement Learning," *Journal of Machine Learning Research*, vol. 16, pp. 1437–1480, 2015.

[3] P. Mallozzi, R. Pardo, V. Duplessis, P. Pelliccione, and G. Schneider, "MoVEMo: A Structured Approach for Engineering Reward Functions," in *IRC'18*, Jan 2018, pp. 250–257.

[4] P. Mallozzi, P. Pelliccione, and C. Menghi, "Keep intelligence under control," in *SE4COG'18*. IEEE Press, 2018.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE'99*. ACM, 1999, pp. 411–420.

[6] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Trans. on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.

[7] A. Pnueli, "The temporal logic of programs," in *FOCS'77*, vol. 00, 1977, pp. 46–57.

[8] L. W. Maxime Chevalier-Boisvert, "Minimalistic gridworld environment for openai gym," https://github.com/maximecb/gym-minigrid, 2018.

[9] G. J. Holzmann, "The logic of bugs," in *FSE'02*, ser. SIGSOFT. ACM, 2002, pp. 81–87.

[10] M. Autili, P. Inverardi, and P. Pelliccione, "Graphical scenarios for specifying temporal properties: An automated approach," *Automated Software Engg.*, vol. 14, no. 3, pp. 293–340, Sep. 2007.

[11] V. Braberman, N. Kicillof, and A. Olivero, "A scenario-matching approach to the description and model checking of real-time properties," *IEEE Trans. on Soft. Eng.*, vol. 31, no. 12, pp. 1028–1041, 2005.

[12] D. Harel and A. Kantor, "Multi-modal scenarios revisited: A net-based representation," *Theor. Comput. Sci.*, vol. 429, pp. 118–127, Apr. 2012.

[13] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," in *ICSE'05*. ACM, 2005, pp. 372–381.

[14] L. Grunske, "Specification patterns for probabilistic quality properties," in *ICSE'08*. ACM, 2008, pp. 31–40.

[15] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[16] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[17] K. Havelund and G. Roşu, "Runtime verification," in *Computer Aided Verification (CAV'01) satellite workshop*, ser. ENTCS, vol. 55, 2001.

[18] M. Leucker and C. Schallhart, "A Brief Account of Runtime Verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009.

[19] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: a java modeling language," in *Workshop (at OOPSLA'98)*, 1998.

[20] A. Pnueli, "The temporal logic of programs," in *Proc. 18th IEEE Symposium on Foundation of Computer Science*, 1977, pp. 46–57.

[21] W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider, "A specification language for static and runtime verification of data and control properties," in *FM'15*, 2015, vol. 9109, pp. 108–125.

[22] C. Colombo, G. J. Pace, and G. Schneider, "Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties," in *FMICS'08*, ser. LNCS, vol. 5596. Springer, 2009, pp. 135–149.

[23] G. Reger, H. C. Cruz, and D. E. Rydeheard, "MarQ: Monitoring at Runtime with QEA." in *TACAS*, ser. LNCS, vol. 9035. Springer, 2015, pp. 596–610.

[24] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "Lola: Runtime monitoring of synchronous systems," in *TIME'05*. IEEE Computer Society Press, June 2005, pp. 166–174.

[25] C. Colombo, G. J. Pace, and G. Schneider, "LARVA - Safer Monitoring of Real-Time Java Programs (Tool Paper)," in *SEFM'09*. IEEE Computer Society, 2009, pp. 33–37.

[26] J. Garcıa and F. Fernández, "A comprehensive survey on safe reinforcement learning," *Mach. Learn. Res.*, vol. 16, no. 1, pp. 1437–1480, 2015.

[27] T. M. Moldovan and P. Abbeel, "Safe exploration in markov decision processes," *arXiv preprint arXiv:1205.4810*, 2012.

[28] M. Turchetta, F. Berkenkamp, and A. Krause, "Safe exploration in finite markov decision processes with gaussian processes," in *Advances in Neural Information Processing Systems*, 2016, pp. 4312–4320.

[29] P. Thomas, G. Theocharous, and M. Ghavamzadeh, "High confidence policy improvement," in *ICML'15*, 2015, pp. 2380–2388.

[30] Z. C. Lipton, A. Kumar, L. Li, J. Gao, and L. Deng, "Combating Reinforcement Learning's Sisyphean Curse with Intrinsic Fear," pp. 1–14, 2016.

[31] W. Saunders, G. Sastry, A. Stuhlmueller, and O. Evans, "Trial without Error: Towards Safe Reinforcement Learning via Human Intervention," 2017.

[32] M. Hasanbeig, A. Abate, and D. Kroening, "Logically-Constrained Reinforcement Learning," 2018.

[33] M. Al-Shedivat, L. Lee, R. Salakhutdinov, and E. Xing, "On the complexity of exploration in goal-driven navigation," *arXiv preprint arXiv:1811.06889*, 2018.

[34] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in *Advances in neural information processing systems*, 2016, pp. 3675–3683.

[35] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[36] L. Medsker and L. C. Jain, *Recurrent neural networks: design and applications*. CRC press, 1999.

[37] J. Randløv and P. Alstrøm, "Learning to drive a bicycle using reinforcement learning and shaping." in *ICML'98*, vol. 98, 1998, pp. 463–471.

[38] I. Kostrikov, "Pytorch implementations of reinforcement learning algorithms," https://github.com/ikostrikov/pytorch-a2c-ppo-acktr, 2018.

[39] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *ICML'16*, 2016, pp. 1928–1937.

[40] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick, "Learning to reinforcement learn," *CoRR*, vol. abs/1611.05763, 2016.