2018 Second IEEE International Conference on Robotic Computing

MoVEMo: a structured approach for engineering reward functions

Piergiuseppe Mallozzi, Raúl Pardo, Vincent Duplessis, Patrizio Pelliccione, Gerardo Schneider
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden
{mallozzi, pardo, patrizio, gersch}@chalmers.se

Abstract—**Reinforcement learning (RL) is a machine learning technique that has been increasingly used in robotic systems. In reinforcement learning, instead of manually pre-program what action to take at each step, we convey the goal a software agent in terms of reward functions. The agent tries different actions in order to maximize a numerical value, i.e. the reward. A misspecified reward function can cause problems such as reward hacking, where the agent finds out ways that maximize the reward without achieving the intended goal.**

As RL agents become more general and autonomous, the design of reward functions that elicit the desired behaviour in the agent becomes more important and cumbersome. In this paper, we present a technique to formally express reward functions in a structured way; this stimulates a proper reward function design and as well enables the formal verification of it. We start by defining the reward function using state machines. In this way, we can statically check that the reward function satisfies certain properties, e.g., high-level requirements of the function to learn. Later we automatically generate a runtime monitor—which runs in parallel with the learning agent—that provides the rewards according to the definition of the state machine and based on the behaviour of the agent.

We use the UPPAAL model checker to design the reward model and verify the TCTL properties that model high-level requirements of the reward function and LARVA to monitor and enforce the reward model to the RL agent at runtime.

I. Introduction

The behaviour of autonomous robotic systems is traditionally designed manually and their correctness relies on extensive testing of their requirements. Other approaches can generate optimal controllers of the robot from a synthesis process that starts from the specification of the system [1], [2]. However, the presence of uncertainty makes it hard to model all the requirements at design-time (the partial knowledge of the environment and its dynamic nature will make the specification necessarily incomplete).

Uncertainty may come from the system's environment, unavailability of resources, the difficulty of predicting users' behaviour, etc. [3], [4], [5]. Consequently, it is extremely difficult to anticipate all the possible and subtle variations of the environment at design-time. In a goal-oriented approach, we set the objective that we want the system to achieve without specifying how. By using *model-free reinforcement learning* [6] we let the agent explore the environment by trial and error, ultimately producing an optimal policy according to the predefined goal. The policy is learned by interacting with the environment and collecting a reinforcement signal, i.e. a numerical *reward* for each action that the agent performs on it.

An incorrect specification of the reward function, and consequently a gap between the designer's intention and the specification, can cause unexpected behaviours in the agent. One of the problems is *reward hacking* [7], meaning that the agent, by taking into account the reward function, manages to get a high reward without achieving the designer's intentions; this is because it might optimize towards the rewards function that is indeed not exactly representing the designer's intentions. For example, in a cleaning robot setting, if the reward function gives a positive reward for not seeing any dirt then the agent might learn to disable its vision rather than cleaning up. Instead, if the reward is given only when the robot actually cleans up then the robot might learn to make a mess first and then cleaning up so that it keeps receiving more and more reward.

By embedding more domain knowledge in the reward function one could avoid the reward hacking phenomenon; this would also help the agent to learn the desired policy faster [8]. However, as reward functions become more complex, in turn, it becomes harder to spot mistakes and to be confident whether the reward values that are finally sent to the agent actually reflect the designer's intentions.

Our work goes in the direction of a better structuring of the reward function with the aim of closing the gap between the designer informal goals and the reward signal. Our contribution is the design and validation of a software infrastructure that enables the verification and enforcement of reward functions to an RL agent. From a high-level perspective our approach, which we have called MoVEMo, consists of four steps:

1) *Modelling* complex reward functions as a network of state machines.
2) *Formally verifying* the correctness of the reward model.
3) *Enforcing* the reward model to the agent at runtime using a monitoring and enforcing approach called LARVA.
4) *Monitoring* the behaviour of the agent as it transverses the state machines to collect the rewards.

Steps 1 and 2 are performed by the designer that iterates the reward function model until it is compliant with the high-level properties that she/he expresses. Steps 3 and 4 are automatically derived and performed from the reward model.

We have validated our approach in an autonomous driving scenario using the TORCS [9] simulation environment. After modelling the goal of the system with our approach, the reinforcement learning agent learns how to steer, accelerate, and break. We are able to detect bugs in reward functions before enforcing them to the agent. We have packaged the software infrastructure in a stand-alone docker image [10].

The rest of the paper is structured as follows. In Section II we introduce background information needed to understand our approach, such as reinforcement learning, formal verification, and runtime enforcement. In Section III we give an overview of existing methods to convey rewards to the agent and the problems that come with it. In Section IV we present the four phases of our approach: modelling, formal verification, enforcement, and monitoring. Finally, in Section V we present a case study in a racing car simulation environment. We conclude and depict future research directions in Section VI.

II. Background

A. Reinforcement Learning

Reinforcement learning is a machine learning technique that involves an agent acting in an environment by choosing predefined actions with the goal of maximizing a numerical reward.

At each time step t an agent receives an observation from the environment that it associates with a certain state. It chooses an action from that state and applies it to the environment that moves to a new state. A reward associated with this transition *state-action-newstate* is determined and sent back to the agent.

In model-free RL, a learning agent starts with no prior knowledge about the environment and, as it receives observations, it tries actions and then collects a reward. It selects its actions by exploiting the knowledge from its past experiences but also by exploring the environment by trial and error learning. The environment evaluates the action taken by the agent at each step by sending back a reward signal to the agent. The goal of the agent is to maximize the expected rewards over time, also known as *return*.

The agent does not learn the reward function but instead it learns the *Q-values*, which are numerical values associated to each action that it can take in a state. In Q-learning [11], a model-free reinforcement learning algorithm, at each time step the agent updates the Q-values associated to the state-action pair as follows:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \cdot max_a(Q(s_{t+1}, a)) - Q(s_t, a_t))$$

where:

- r_t is the reward for the current agent's state;
- s_t is the current state of the agent;
- a_t is the action picked by the agent;
- α is the learning rate; it indicates how much the agent will consider the newly acquired information into its previous state-action value;
- γ is the discount factor; If the factor is close to 0 the agent will only consider the current reward; contrariwise, if the factor is close to 1 the agent will try to maximize the long-term sum of the future rewards.

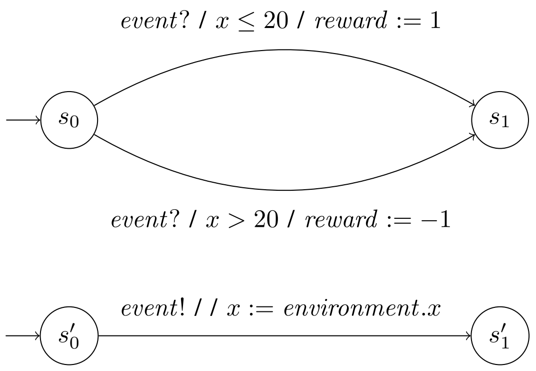
In this paper, we use the Deep Deterministic Policy Gradient (DDPG) algorithm, which uses the ideas of Q-learning in a continuous action domain [12]. It is an *actor-critic* algorithm and it uses two neural networks, one for the actor and one for the critic. The actor is a policy: it produces the action a given a state s. The critic is the value function: it estimates the action-value function $Q(s, a)$. The critic estimates the value of the current policy by Q-learning, so using the rewards from the environment to improve its estimations. The critic also provides a loss function to the actor that updates its policy in a direction that improves the Q value.

B. Formal Verification

Depending on the system to be analyzed and the kind of properties to be proved, different formalisms are used for their description, giving also rise to different verification techniques. One common technique consists in using finite state machines (e.g., *automata*) to model the system and temporal logic to describe the specification or properties. Formulating specifications in temporal properties is an error-prone task that requires mathematical expertise. In order to facilitate the task of specifying formulas in a correct and accurate way, we plan to use user-friendly notations and approaches, like [13], [14].

In this paper, we use the model-checker UPPAAL [15] to guarantee that the reward function of the reinforcement learning algorithm complies with certain requirements. In UPPAAL the system is modelled as a network of *timed automata* [16]. These automata are state machines where nodes can be labelled with invariants and transitions are labelled with synchronization primitives, guards, and updates. It also allows the definition of discrete variables, and in particular, a special kind of continuous variables called *clocks* to measure the pass of time. Synchronization is carried out using channels. There are two synchronization primitives associated with channels. For a channel c, we can send information to the channel ($c!$) or receive from it ($c?$). Receiving is a blocking primitive, therefore the automaton waits until another automaton sends information to the channel. Guards are Boolean expressions involving any of the variables of the automaton. Likewise, using updates we can modify the value of variables of the automaton. Figure 1 shows an example of a network of two automata, the labels on the transitions are in the form of $synch/guard/update$.

In this network of automata, there is a channel $event$, and two integer variables x and $reward$. Intuitively, the top automaton models a reward function, and the bottom automaton the environment in which the reinforcement learning agent is working. Initially, the automaton at the top is waiting for an event to occur, since it is in state s_0 and both transitions are labelled with $event?$. When the bottom automaton—initially in state s'_0—changes state, it sends a message to the channel event ($event!$) and updates the value of variable x of the



$event? \ / \ x \leq 20 \ / \ reward := 1$
s_0
s_1
$event? \ / \ x > 20 \ / \ reward := -1$
s'_0
$event! \ / \ / \ x := environment.x$
s'_1

Fig. 1: Example of a network of UPPAAL automata

automaton with the value of variable x in the environment ($x = environment.x$). The automaton modelling the reward function now reads the value of x and depending on whether it is greater than 20, it gives the agent a positive or a negative reward.

In UPPAAL, properties to verify are written in Timed Computational Tree Logic (TCTL) [17]. This logic consists of the usual propositional logical connectives: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and some operators to quantify over the execution paths of the system and their states. In this paper, we only make use of the operator AG—Always Globally, or, more formally, for all execution paths and in all states. The meaning of this operator is better explained with an example. Consider the automata above, the following TCTL formula checks that *for all executions paths and all states in the system the value of* $reward$ *will always be either -1 and 1*,

$$\mathsf{AG}(reward = 1 \vee reward = -1)$$

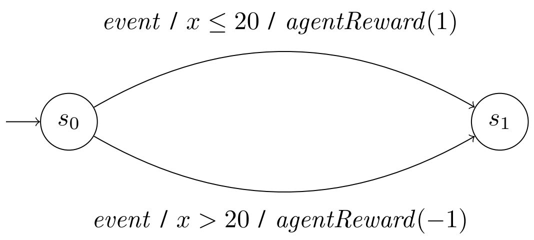
this formula will be satisfied in our model of the reward function. We can also specify properties of states where some condition holds. Consider the following property *if* x *goes above 20 the agent should always get a negative reward*. This property can be expressed as follows:

$$\mathsf{AG}(x > 20 \Rightarrow reward < 0)$$

This property only considers the state of the execution paths where $x > 20$. Again, it is easy to see that the previous property is satisfied in the system.

UPPAAL offers the possibility of defining functions that can be associated with the $guard$ or $update$ fields of the automata. This is very convenient as the user can specify complex reward functions in a syntax similar to C and then use the model checker to verify that the specified properties hold across all possible input values.

As we show in the following sections, we formalize the high-level requirements of the reward function in TCTL so that we can automatically check that our model of the reward function satisfies them. Nevertheless, there is still a gap between the model of the reward function and its implementation. We bridge this gap using runtime enforcement tools.



$event \ / \ x \leq 20 \ / \ agentReward(1)$
s_0
s_1
$event \ / \ x > 20 \ / \ agentReward(-1)$

Fig. 2: Example LARVA automaton

C. Runtime Enforcement

Runtime enforcement is a technique that ensures that the behaviour of a system—to which we do not have access a priori, i.e., we can only access the system after it has been deployed and is running—complies with certain properties. Runtime monitors are small programs that passively run in parallel with the system while the enforcers act on the monitored systems. Monitors observe the system and based on their perceptions, influence the behaviour of the system to guarantee that some properties are satisfied.

The tool LARVA [18] offers the possibility of defining runtime monitors using special kind of automata called *DATEs* (*Dynamic Automata with Timers and Events*) [19]. DATEs can be automatically compiled into an executable Java program. All elements in a timed automaton can be converted into a DATE (see Section IV-C for the details), which makes LARVA an ideal candidate to define (and generate) the monitor for reward function specified using timed automata. DATEs consist of states and transitions labelled with triples *event/guard/update*. The labels on the transitions mean that if a matching event occurs in the system and the guard—based on event parameters and the automaton state—holds, then the update is carried out and the current state of the automaton updated.

Consider the LARVA automaton[1] in Fig. 2. This automaton is almost identical to the reward function UPPAAL automaton presented above. There are only two differences: i) $event$ represents that the transition that will be triggered every time the monitor observes that the reinforcement learning agent performed an event; ii) the function $agentReward$ sends the reward to the reinforcement learning agent, instead of storing it in a local variable. As in UPPAAL automata, transitions are triggered if the guards are true.

We translate UPPAAL automata to LARVA (we describe the details of the translation in Section IV-C) to automatically generate a Java monitor. This monitor will run in parallel with the learning agent. LARVA automatically instruments the monitor and the learning agent so that the monitor can observe the events that the agent performs and its impact on the environment. Based on the events and the observation the monitor will provide the agent with the corresponding reward.

[1]In what follows we will use both "DATEs" and "LARVA automata" to refer to the underlying specification language of LARVA.

The reward that the monitor produce is guaranteed to comply with the high-level requirements of the agent since the monitor behaves as defined in the UPPAAL automaton which has been formally verified.

III. REWARD ENGINEERING: STATE OF THE ART

A. Conveying rewards to the agents

Daniel Dewey stated the *reward engineering principle* [20] as follows: as reinforcement-learning-based AI systems become more general and autonomous, the design of reward mechanisms that elicit the desired behaviour becomes both more important and more difficult.

In reinforcement learning the only way to convey the goal that the agent has to achieve is the reward signal, which determines the success of an action's outcome. The system designer has to engineer the reward function so that it encodes the informal objective that he wants the agent to achieve. He has to translate these goals into a numerical form, rewarding the agent for acting *good* and penalizing it for acting *bad*.

A well-defined reward function has demonstrated to be successful in several cases such as Atari games [21] and board games [21]. These examples show that a simple reward function, such as the score of the game, can teach the agent to achieve the optimal policy. However, in many tasks the *right* reward function is less clear. In more complex domains such as in automotive, we need more complex functions to produce the desired behaviour of the agent. For example, let us assume that we want to make an agent steering a car. Then, imagine that as reward function we only model the travel time from a point A to a point B. The agent might take an almost infinite amount of trials to drive properly, since it will have to try different combinations of steering, accelerating, and breaking actions.

When the action domain of the agent is continuous, a common way to convey the right rewards is to elicit it from demonstrations of an expert. In methods such as apprenticeship learning, the reward function is learned from observations [22]. The idea is that we take as given an expert optimal policy and we determine the underlying reward structure. This problem is also known as *inverse reinforcement learning*. The given policy follows some optimal reward function but there is no need to articulate it. The agent will derive it by seeing demonstrations.

Another issue with determining the right reward is that the agents need an early feedback on the success of their actions without having to wait for the end of the task. Reward shaping has been addressing such challenge by providing guidance to the agent and incorporating prior knowledge in the reinforcement learning. For example, in potential-based reward shaping we provide heuristic knowledge by an additional reward $F(s, s') = \gamma\phi(s') - \phi(s)$ when moving from state s to s'. Where $\phi(s)$ is a potential function associated with the state s. Prior knowledge can also be encoded directly into the initial Q-values of the agent, which can be equivalent to shape the reward by using a potential function [23].

Multiple sources of rewards can also make the reward function more robust and difficult to hack, as also proposed by Amodei et al. [24]. The reward signals might be independent, complementary, or conflicting with each other. When the RL agent has to deal with multiple reward signals we refer to Multi-Objective Reinforcement Learning (MORL) [25]. There are two main categories in MORL depending on the number of policies to be learned by the agent: single-policy and multiple-policy approach [26]. In our approach, the RL agent learns a single policy based on multiple sources of rewards.

B. Unexpected behaviours

The encoding of system goals into a reward function can lead to unexpected behaviours in the agent, either because the designer does not include or have the correct information or because she/he makes mistakes during the design of the reward function. Amodei et al. [24] point out some of the major problems in achieving safe and expected behaviours in machine learning agents, such as avoiding *negative side effects* and avoiding *reward hacking*.

The first problem emerges in large environments when the designer of the reward function focuses on few aspects of the environment omitting others. The agent might manage to achieve the designer's goal by doing something unrelated or destructive to parts of the environment that the designer did not include in the reward function. Basically, the designer should include constraints to what the agent can and cannot do in the environment and not only inferring the system goal. However, in large environments, it might not be possible to identify all the constraints.

We have talked about how simple reward functions such as the score of the game can lead the agent to play several ATARI games. This is not always the case as the agent might simply learn to maximize the reward function without satisfying the specifications of the game (*reward hacking*). For example, in CoastRunner, a boat race game, a human player understands that the goal is to finish the race as quickly as possible while collecting points on the way. The reinforcement learning agent instead keeps hitting some targets on the way without finishing the race because it learns that by doing so it can gain a higher score[2]. There is a wrong assumption here: the reward function does not properly reflect the informal goal of the game to finish the race but rather to simply maximise the score.

Finally, we have a designer that wants the agent to accomplish a certain objective. He has to encode the objective as a reward function and convey it to the agent. We have seen how this encoding can lead to unexpected behaviour in the agent, either because the designer does not include or have the correct information or because he makes mistakes in the designing of the reward function.

IV. MOVEMO

We propose an approach called MOVEMO to *model*, *verify*, *enforce* and *monitor* reward functions. MOVEMO goes in the

[2]Faulty reward functions https://blog.openai.com/faulty-reward-functions/

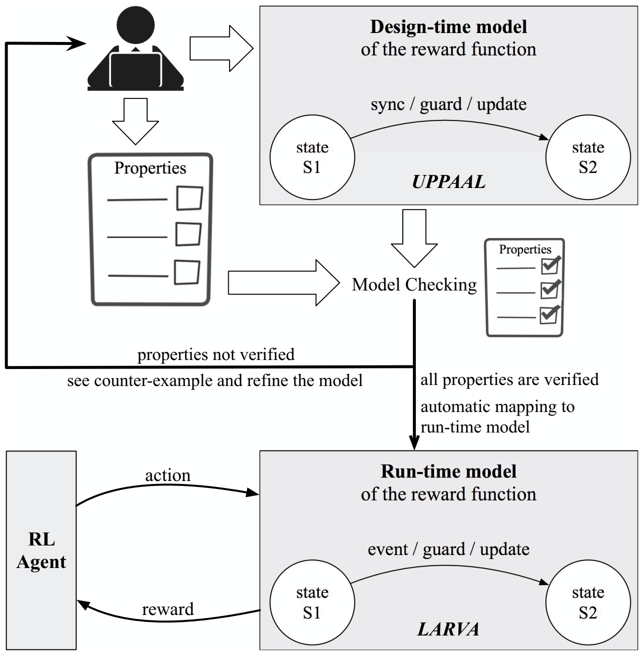


Fig. 3: Proposed approach: MoVEMo

direction of carefully engineering complex reward functions by using formal methods combined with the use of multiple reward signals. Our approach is shown in Figure 3 and consists of four main steps:

1) The designer graphically models different goals that want the system to achieve as state machines using UPPAAL, hence creating a *reward model*.
2) The designer expresses the requirements of its reward function in terms of TCTL properties. UPPAAL automatically verifies that the properties hold in the *reward model* across all possible inputs that the environment could issue.
3) The *reward model* is automatically translated into a LARVA model preserving the original behaviour. This model interacts with the RL agent at runtime issuing rewards.
4) The LARVA model also serves the purpose of *monitoring* the behaviour of the RL agent.

A. Step 1: From requirements to reward function

Traditional software development cycle starts with the definition of high-level requirements of the system that are then broken down into smaller objectives that have to be achieved by the individual components. Goal-oriented approaches can be helpful in refining high-level functional requirements into operational goals and non-functional requirements in system invariants [27], [28], [29].

In this phase, after an initial elicitation of the high-level goals to be conveyed to the agent, the designer models them as automata in UPPAAL. Each goal is represented by a separate automaton whose states encode a configuration of the agent in the environment. A reward function is associated with each state.

At all times, the state of the agent resolves in multiple states in the monitor, one state for each automaton. Each state is issuing a reward proportional to how distant is the agent to the goal modelled in the automata. We compute a single scalar value from all individual rewards that we then feed to the agent as in the standard reinforcement learning framework.

B. Step 2: Verifying the requirement

The designer expresses the reward function requirements in terms of TCTL properties. Furthermore, all the environment variables that interact with the reward model have to be expressed as simple automata that can produce values within a certain range. UPPAAL can then formally verify the compliance of the *reward model* with the properties expressed by the designer.

In case any of the properties is not verified, UPPAAL will show a counterexample. The designer is then able to understand in which case its reward function does not hold the specified property. This is a very useful feature that allows the designer to go back and iterate on the reward model until she/he is satisfied with the result.

C. Step3: Enforcing the reward function

To enforce the reward function we convert the UPPAAL model into a LARVA monitor. To do so, we have established mapping rules between UPPAAL and LARVA automata. More specifically:

- *Channels* in UPPAAL automata become event listeners in LARVA. In UPPAAL the reward model *sync* with the environment model. In LARVA the *events* are automatically fired from the real or simulated environment values.
- *Guards* on the transitions of UPPAAL automata are directly mapped as *guards* in LARVA automata.
- *Updates* in UPPAAL are also mapped directly to updates in LARVA, which can be associated with actions such as the execution of an arbitrary program. This transformation also requires defining the same variables in both automata. For example, the variable associated with the rewards in UPPAAL are converted into equivalent variables in larva that are then updated and sent to the RL agent as part of a LARVA update.
- *Clocks* in UPPAAL become timers in LARVA.

Following the previous steps, the UPPAAL model is automatically translated into a LARVA DATE that issues rewards to the RL agent at runtime.

The LARVA model encapsulates all the states in which the agent can be at any time in the environment. This model can also be used to monitor the agent behaviour. At the moment the model only collects data on which state the agent visits and issues a reward for each state. However, having a model that encapsulates the possible behaviours of the system into states can serve the purpose of preventing the agent to reach *bad* states. We are currently exploiting such aspects of *preventive monitoring*.

V. Autonomous Driving with TORCS

In order to validate our approach, we have used The Open Racing Car Simulator (TORCS) [9] to simulate an environment where a RL agent can apply its action and learn how to drive. We have used the DDPG algorithm [12] as the decision-making policy of the agent. We have extended an existing implementation [30] of the DDPG in order to support external rewards coming from the LARVA model. We have encapsulated all the software platform in a docker image so that it is very easy to build a reward function, lunch a simulation and collect the results [10].

The RL agent uses the following signals from the simulation environment:

- *TrackPos*: Distance between the car and the track axis.
- *Track*: Vector of 19 range finder sensors around the vehicle. Each sensor returns the distance between the track edge and the car within a range of 200 meters.
- *Opponents*: Vector of 36 sensors around the vehicle. Each sensor returns the distance of the closest opponent within 200 meters range.
- *Damage*: Current damage of the vehicle, the higher is the value the higher is the damage.
- *Angle*: Angle between the car direction and the track axis
- *SpeedX*: Speed of the car along its longitudinal axis.

Based on the above observations the RL agent can apply the following actions:

- *Steering*: The steering value can be between -1 (full left) and 1 (full right).
- *Accelerating*: The value of the virtual gas pedal can be between 0 (no gas) and 1 (full gas).
- *Breaking*: The value of the virtual break pedal can be between 0 (no break) and 1 (full break).

A. Conveying the goals to the agent

Each state provides a reward value that should reflect how much the agent is compliant with the goal associated with the automata that the state belongs to. The reward value can be a simple scalar or it can come from a complex function. Below we describe the goals we have modelled.

1) Staying in the middle of the lane: This goal corresponds to try to keep the value of $TrackPos$ equal to zero. We have modelled 4 states in which the vehicle can be: `CenterRoad`, `LimitRoad`, `RightOffRoad`, `LeftOffRoad` according to the position of the vehicle on the road. In each state the reward function is proportional to the error with $TrackPos$. Each state has additional rewards according to how far the car is from the centre. Furthermore, we take into consideration the previous action taken by the agent, penalizing it more if it keeps steering towards the wrong direction.

2) Keeping a certain speed: We want the agent to keep a constant speed of 100 Km/h, slowing down when approaching a curve. For this goal we have modelled 3 states in UPPAAL, according to the state of the vehicle in the road: `GoingStraight` and `Curve`. The reward, when there are no curves ahead of the vehicle, is proportional to the error related to the goal of keeping the speed at 100Km/h. In order to detect if a curve is in front of the car, we have used the 19 range finder sensors around the vehicle (*Track*) to build a function that is used in the guard to transit to the state `Curve`. Since we want to stimulate the car to slow down when a curve is detected, the reward function of this state is proportional to the error of a lower speed than the goal speed of 100 Km/h.

3) Avoid damages: We have modeled this goal in two states: `Damage` and `Normal`. The reward function penalizes the agent by a constant value every time it receives a damage, this can happen by hitting other vehicles or going off-track.

4) Avoid getting stuck: When the LARVA monitor detects that the vehicle keeps going at a very slow speed for a while, it might be because the vehicle is in a state where is trying actions but it is physically blocked by the limit of the road. We have modelled this situation in a `Stuck` state where the agent gets penalized if it stops going forward. At the same time, it is also encouraged to try hard steering manoeuvres that can get it out of this state.

5) Avoid other vehicles: When racing with other cars we want to avoid collisions with the other vehicles. We have split this goal into four UPPAAL automata with the purpose of detecting the presence of other vehicles in the four sides of the car: *Ahead*, *Behind*, *Left*, *Right*. Each automaton is composed of several states according to how distant are the other cars to the side of the vehicle, a reward function is assigned to each of these states penalizing the agent for being close to other cars.

Furthermore, in the presence of other vehicles, staying in the middle might not always be the best policy. The RL agent might want to overtake or simply avoid collisions with other vehicles by going to the right or to the left of the road. This is why, when other vehicles are racing, we update the $TrackPos$ goal according to which side of the road is free to drive. For example, if a car is detected on the right side, we reward the agent for driving to the left side, and so on. This is directly encoded in the UPPAAL model so we do not need to re-run the verification of the properties.

B. Verifying properties

Combining multiple sources of rewards, each contributing with complex functions, will create a big reward model with many states. As we have chosen UPPAAL as modelling environment, we can verify that the properties hold across all possible states of the system.

We have only modeled the signals that the reward model uses to compute the reward such as `TrakcPos`, `SpeedX`, `Angle`, `Damage`. After discretizing each signal with a step value and a range with a lower and upper bound, we are able to run simulations in UPPAAL and verify that the properties that we specify hold across all possible states. In our example, we have modelled 6 automata with a total of 21 states.

We have verified several high-level design goals that helped us fixing mistakes or bugs in the reward functions that we have encoded in UPPAAL. Though we could have assigned any value as a reward, in our model we have assigned only positive

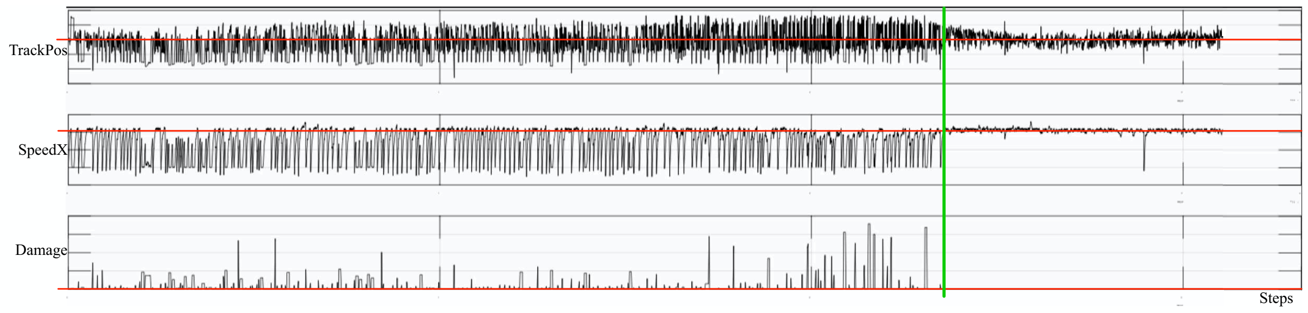


Fig. 4: Some of the monitored values from one of the simulations in TORCS. The red lines are the goals inferred by the UPPAAL reward model.

values to states of the system that are *good* and negative values when it deviates from the goal of the system. In the following, we describe two examples of TCTL properties that we have verified in the reward model for TORCS.

1) If the deviation from the speed goal is more than 10 km/h, the reward associated with the speed should be negative:

$$\mathsf{AG}((Speed.goingStraight) \wedge (goalSpeedStraight - speedX > 10) \implies speedX_{reward} < 0)$$

where $Speed$ is the automaton connected to the speed goal and it is in the state $GoingStraight$, where there are no curves ahead of the car. $goalSpeedStraight$ and $speedX$ are two variables representing respectively the target speed for when the road ahead is straight and the current speed of the vehicle.

2) If the vehicle is proceeding in the centre of the road, it has no damage and with a speed less than 10Km/s from the goal speed, then the overall reward should be positive:

$$\mathsf{AG}(TrackPos.centerRoad \wedge Speed.goingStraight \wedge Damage.no \wedge (speed_{error} < 10) \wedge (goalSpeedStraight - speedX > 10) \implies combined_{reward} > 0)$$

In this case we take in consideration three goal models $TrackPos$, $Speed$, $Damage$ and we verify that the combination of all their rewards is positive.

C. Results

We have compared our UPPAAL reward model (URM) with the reward function proposed in [30] that has already been shown to be an improvement of the original reward function proposed by Google in [12]. We will refer to this function as BRF (benchmark reward function) and it is expressed as follows:

$$R_t = V_x * cos(\theta) - V_x sin(\theta) - V_x * |TrackPos|$$

Where V_x is the vector representing the car velocity, and θ is the angle between V_x and the track axis. The above reward function aims to maximize the longitudinal velocity (first term), minimize transverse velocity (second term) while penalizing the agent if it deviates from the center of the road (third term).

	Without opponents		With opponents	
	BRF	URM	BRF	URM
Time (h)	3.4	2.1	13.1	12.0
Episodes (#)	302.8	146.2	1312.7	925.7
Center (%)	60.9	83.6	47.3	61.4
Stuck (%)	5.3	4.6	29.7	37.2

TABLE I: Average results of the Uppaal Reward Model (URM) compared with a Benchmark Reward Function (BRF).

The results show that the agent learns much faster by using reward functions produced through our approach. Table I shows the average results of *100 iterations*. In order to complete one iteration, the agent must learn how to drive on the track and complete *20 laps*. An episode of the algorithm ends when the vehicle is perpendicular to the track axis.

The first two columns are the results when the car is racing without opponents while the last two columns are the results with other vehicle racing together. Table I shows the values of Time (in hours) and the number of episodes to complete 20 laps. Furthermore, we see the percentage the agent stayed in the state *Center* of the road (the higher the better) and how much it got the state *Stuck*. We see that with the UPPAAL reward function it achieves its goal faster and with less number of episodes than the simple python function. The agent performs quite well when there are no other cars on the track, but not that good when racing with other vehicles. Dealing with opponents is a more complicated problem and we believe the reward function can be further improved.

Figure 4 shows some of the monitored values i.e. `TrackPos`, `SpeedX`, `Damage` of one iteration of the agent using the UPPAAL reward model. The red lines indicate the goals set in the UPPAAL model for each value: stay close to the centre of the road, keep a speed of `100 Km/h` and avoid damages. We can see that the agent starts with a random behaviour, receiving a lot of damages and continuously changing speed as it learns the goals by trial and error. The green line indicates the point when the agent has learned the goals and continues going at the desired speed while avoiding damages and staying more or less in the middle of the road.

VI. Conclusion and Future Work

We have presented a framework in which one can model reward functions for reinforcement learning agents as state machines. This framework allows us to verify properties of the reward function at design-time. We have also formalised and implemented an automatic mapping between the design-time model of the reward function and its implementation that is actually exploited by the agent at runtime. Our results show that, by building more complex and robust reward functions, the agent can learn faster to achieve its goal.

This work is a first step toward the engineering of the reward function. This can help the designer eliciting the requirements in terms of goals to be achieved by the agent as well as to identify the possible uncertainties due to, e.g. the unpredictability of the environment.

The reward model can also be used to monitor the behaviour of the RL agent at runtime. As future work, we envision to use such monitor as a safety envelope for the agent. By doing preventive monitoring we can detect anomalies in the behaviour of the agent and prevent potentially dangerous actions to be executed on the environment.

Acknowledgment

This work has been partially supported by: the Wallenberg Autonomous Systems Program (WASP), the Swedish funding agency SSF under the grant *Data Driven Secure Business Intelligence*, the Swedish Research Council (*Vetenskapsrådet*) under grant Nr. 2015-04154 (*PolUser: Rich User-Controlled Privacy Policies*), and the European ICT COST Action IC1402 (*Runtime Verification beyond Monitoring (ARVI)*).

References

[1] S. Jha and V. Raman, "Automated synthesis of safe autonomous vehicle control under perception uncertainty," in *NASA Formal Methods Symposium*. Springer, 2016, pp. 117–132.

[2] K. He, M. Lahijanian, L. E. Kavraki, and M. Y. Vardi, "Reactive synthesis for finite tasks under resource constraints," in *International Conference on Intelligent Robots and Systems (IROS)*. Vancouver, BC, Canada: IEEE, Sep. 2017, to Appear.

[3] N. Esfahani and S. Malek, *Uncertainty in Self-Adaptive Software Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 214–238. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35813-5_9

[4] M. Autili, V. Cortellessa, D. Di Ruscio, P. Inverardi, P. Pelliccione, and M. Tivoli, "Eagle: Engineering software in the ubiquitous globe by leveraging uncertainty," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 488–491. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025199

[5] D. Garlan, "Software engineering in an uncertain world," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 125–128. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882389

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

[7] T. Everitt, V. Krakovna, L. Orseau, M. Hutter, and S. Legg, "Reinforcement learning with a corrupted reward channel," in *26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[8] X. Li, Y. Ma, and C. Belta, "A policy search method for temporal logic specified reinforcement learning tasks," *arXiv preprint arXiv:1709.09611*, 2017.

[9] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner, "Torcs, the open racing car simulator," *Software available at http://torcs. sourceforge. net*, 2000.

[10] *TORCS-*Larva *Docker Image.*, https://hub.docker.com/r/pmallozzi/rl_-monitor/ [Accessed: 2017-12-18].

[11] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[12] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[13] 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 Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, July 2015.

[14] 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. [Online]. Available: http://dx.doi.org/10.1007/s10515-007-0012-6

[15] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems," in *Proc. of Workshop on Verification and Control of Hybrid Systems III*, ser. LNCS, no. 1066. Springer–Verlag, October 1995, pp. 232–243.

[16] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[17] R. Alur, C. Courcoubetis, and D. L. Dill, "Model-checking in dense real-time," *Inf. Comput.*, vol. 104, no. 1, pp. 2–34, 1993. [Online]. Available: https://doi.org/10.1006/inco.1993.1024

[18] C. Colombo, G. J. Pace, and G. Schneider, "LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)," in *7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09)*, 2009, pp. 33–37.

[19] ——, "Dynamic event-based runtime monitoring of real-time and contextual properties," in *International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 2008, pp. 135–149.

[20] D. Dewey, "Reinforcement Learning and the Reward Engineering Principle," pp. 1–8.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv preprint arXiv: . . .*, pp. 1–9, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[22] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the twenty-first international conference on Machine learning*. ACM, 2004, p. 1.

[23] E. Wiewiora, "Potential-based shaping and q-value initialization are equivalent," *Journal of Artificial Intelligence Research*, vol. 19, pp. 205–208, 2003.

[24] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete Problems in AI Safety," *arXiv*, pp. 1–29, 2016. [Online]. Available: http://arxiv.org/abs/1606.06565

[25] A. Nowe, K. Van Moffaert, and M. Drugan, "Multi-objective reinforcement learning," in *European Workshop on Reinforcement Learning*, 2013.

[26] C. Liu, X. Xu, and D. Hu, "Multiobjective reinforcement learning: A comprehensive overview," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 45, no. 3, pp. 385–398, 2015.

[27] E. Letier and A. Van Lamsweerde, "Deriving operational software specifications from system goals," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, p. 119, 2002. [Online]. Available: http://portal.acm.org/citation.cfm?doid=605466.605485

[28] A. Dardenne, A. Van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," *Science of computer programming*, vol. 20, no. 1-2, pp. 3–50, 1993.

[29] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using nonfunctional requirements: A process-oriented approach," *IEEE Transactions on software engineering*, vol. 18, no. 6, pp. 483–497, 1992.

[30] "Using Keras and Deep Deterministic Policy Gradient to play TORCS," https://github.com/yanpanlau/DDPG-Keras-Torcs.