LTL and Beyond: Formal Languages for Goal Specification in Reinforcement Learning

Submission #6050

Abstract

In reinforcement learning, an agent is guided by the rewards it receives from the reward function. Unfortunately, it may take many interactions with the environment to learn from sparse rewards, and it can be challenging to specify reward functions that avoid making unintended behaviors optimal. We propose using reward machines, automata-based structures that provide a canonical representation for reward functions that exposes some of the reward function structure to the reinforcement learner to exploit. We show how specifications of reward in various formal languages, including LTL and various regular languages, can be automatically translated into reward machines, easing the burden of complex reward function specification. We further show how the exposed structure can be exploited by tailored q-learning algorithms and automated reward shaping techniques in order to improve the sample efficiency of reinforcement learning.

1 Introduction

In Reinforcement Learning (RL), an agent that is unaware of the dynamics of its environment or its reward model must act exploratorily, performing actions and noting the resultant state and reward, in order to learn how to act – to learn a policy that maximizes its expected cumulative reward. RL is particularly useful in complex environments that are difficult to model and therefore cannot be solved using more conventional sequential decision making techniques.

Two challenges that plague many RL systems are: (i) the difficulty of reward specification, and (ii) sample (in)efficiency. Designing a high-fidelity reward function requires consideration of both the task and the environment. As tasks become more complex, temporally extended, and multi-faceted, it is common for programmers to struggle with reward function design and specification. And once specified, most nontrivial reward functions deliver reward sparsely, necessitating millions of exploratory episodes to converge to a reasonable quality policy.

Recent work by Toro Icarte *et al.* [2018a;2018b] made progress on these two challenges by observing that since the reward function was typically specified by a programmer, it need not be treated as a black box but rather that it could serve as input to a specialized RL learning algorithm that was able to exploit the structure of the reward function. They showed that doing so could drastically improve sample efficiency, resulting in faster convergence to high-quality policies (optimal in the tabular case), even for problems that were heretofore unsolvable using state-of-the-art tabular and Deep RL methods. Toro Icarte and colleagues' early work exploited a subset of Linear Temporal Logic (LTL) to specify the reward function [Toro Icarte *et al.*, 2018a], while subsequent work exploited an automata-inspired structure called a reward machin [Toro Icarte *et al.*, 2018b]. Both of these specification languages supported compact propositional or relational representation of state properties as well as easy specification of temporally extended behavior. Each was paired with specialized q-learning and Deep q-learning algorithms that exploited the structure of the specification language.

Unfortunately, when it comes to specification languages there is no one language that proves compelling for the diversity of RL applications and reward structures. One need only look at the diversity of goal specification languages for AI automated planning and similarly the diversity of specification languages for hardware and software verification and synthesis to appreciate this point. However, to accommodate a diversity of specification languages with the above approach would require development of specialized q-learning algorithms for each language.

Inspired by Toro Icarte et al.'s work and informed by this observation, in this paper we propose to use reward machines as a *lingua franca* – a canonical representation for specifying RL reward functions. To this end, we provide a formal characterization of reward machines in terms of Mealy machines [Mealy, 1955] highlighting the correspondence to finite state automata. We then leverage the correspondence between finite state automata and regular languages [Hopcroft and Ullman, 1979] to show how a diversity of compelling goal and property specification languages, including several variants of LTL, regular expressions, and other goal specification languages, augmented with scalar rewards, can all be translated to reward machines. Indeed, even controlled subsets of natural language can, in principle, be translated to reward machines. We further observe that for many of these languages, automated compilation methods and software already exist. By translating all of these specification languages to reward machines, we can use learning algorithms that exploit reward function structure, without having to write a new learning algorithm for each language. Further, adoption of a canonical representation enables the composition of reward functions initially specified in multiple specification languages. In the latter part of the paper, we propose new reward machine

tailored q-learning algorithms that significantly enhance existing algorithms through the exploitation of reward shaping (e.g, [Ng *et al.*, 1999]). Experiments with tabular and Deep q-learning confirm the merits of our approach in improving sample efficiency and providing high-quality solutions.

2 Background on Reinforcement Learning

Reinforcement learning involves an agent interacting with an unknown environment, learning how find rewards [Sutton and Barto, 1998]. Typically, the environment is modeled as a Markov Decision Process (MDP). An MDP with an initial state is a tuple $\mathcal{M} = \langle S, A, s_0, T, r, \gamma \rangle$ where S is a finite set of *states*, A is a finite set of *actions*, $s_0 \in S$ is the initial state, $T(s_{t+1}|s_t, a_t)$ is the *transition probability distribution*, $r : S \times A \times S \to \mathbb{R}$ is the *reward function*, and $\gamma \in (0, 1]$ is the *discount factor*.

At each time t, the agent picks an action a_t and transitions from a state $s_t \in S$ to a state $s_{t+1} \in S$ drawn from the distribution $T(\cdot|s_t, a_t)$. We will call a triple (s_t, a_t, s_{t+1}) an *experience*. From that experience (i.e., after performing a_t) the agent gets the reward $r(s_t, a_t, s_{t+1})$.

In RL, the agent starts not knowing T and r, and tries to learn a policy. A policy is a probability distribution $\pi(a|s)$ over actions given a state. An optimal policy is one that maximizes the expected discounted future reward from each $s \in S$. Given a policy π, we can define its corresponding *q-function* $q^\pi(s, a)$ as the expected discounted future reward if action a is taken in s and policy π is followed to pick all later actions. Any optimal policy π^* will be such that its corresponding q-function q^* satisfies the Bellman equation:

$$q^*(s, a) = \sum_{s' \in S} T(s'|s, a) \left(r(s, a, s') + \gamma \max_{a' \in A} q^*(s', a') \right)$$

Furthermore, it's well-known that, given a function q^* satisfying that equation, an optimal policy can be extracted.

2.1 Q-Learning

Tabular q-learning [Watkins and Dayan, 1992] learns policies by learning to approximate the optimal q-function. An approximate q-function $\tilde{q}(s, a)$ is initialized in some manner (e.g. randomly), and after each experience (s, a, s') the approximation is updated according to the rule

$$\tilde{q}(s, a) \xleftarrow{\alpha} r(s, a, s') + \gamma \max_{a'} \tilde{q}(s', a')$$

where $x \xleftarrow{\alpha} y$ abbreviates $x \leftarrow x + \alpha \cdot (y - x)$ and α is a hyperparameter (the *learning rate*).

Tabular q-learning has some attractive features. It is an *off-policy* method that can learn from the experiences generated by any policy. The algorithm converges to an optimal policy if, in the limit, the agent visits each state-action pair infinitely often. However, for problems with large or continuous state spaces, tabular q-learning is impractical.

Deep Q-Networks (DQN) [Mnih *et al.*, 2015] is a more modern variation of q-learning that approximates the q-function using a deep neural network. Like tabular q-learning, DQN is an off-policy algorithm, but unlike tabular q-learning, DQN is not guaranteed to converge to an optimal policy, and can even diverge due to *delusional bias* [Lu *et al.*, 2018] and other instability pathologies. In our experiments with Deep RL in this paper, we used two enhancements on DQN, *Double DQN* [Van Hasselt *et al.*, 2016] and *Prioritized Experience Replay* [Schaul *et al.*, 2015], as [Toro Icarte *et al.*, 2018b] did.

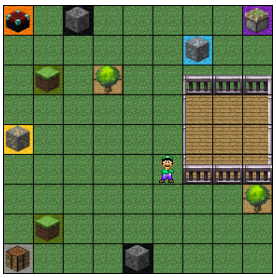


Figure 1: Minecraft-inspired domain containing natural resources, buildings where they can be processed, and perils such as zombies (adapted from [Andreas *et al.*, 2017]).

3 Specifying Rewards

While associating reward with the change in score of a video game is straightforward, other reward functions can be challenging to specify, because reward can be associated with complex behavior as well as behavior avoidance, including safety constraints. Just as with programming languages, or goal specifications for AI model-based planning (e.g., [Baier *et al.*, 2008]), RL reward specification can benefit from a well-designed high-level specification languages. Since rewards can be functions of both state and actions, a reward specification language should be able to make reference to properties of the state of the system as well as its dynamics. Further, since we may want to talk about behavior that manifests over time, we should be able to specify temporally extended patterns of behavior. Finally, it may not be the case that one language will be well-suited to specification of every type of reward-worthy behavior. As such, we might like to use different languages for different aspects of our reward function, and we'd like these diverse specifications to be compositional and with a compelling, human-friendly syntax.

Illustrative Example: We consider a variant of a Minecraft-inspired domain originally proposed in [Andreas *et al.*, 2017], and depicted in Figure 1. In this 2D grid world, Luigi can extract raw material such as wood, grass, iron, gold, and gems, and can interact with his environment using a factory, toolshed or a workbench to construct things such as ropes or bridges. There are also dangers such as zombies that lurk at night. Here are some English-specified examples of reward-worthy behavior in this environment.

- *"Make a bridge by collecting wood and iron in any order, and using the factory afterwards."*
- *"If it's night time, stay in the shed until daylight."*
- *"Always avoid zombies."*
- *"While there are gems on the ground, put them in your bag. When your bag is full, deliver the gems to the shed, and get an empty bag."*

Reward functions are classically Markovian. These rather simple examples motivate the need for languages that support easy specification of non-Markovian reward functions that capture conditional or even negated temporally extended properties over states and/or actions, sometimes with looping.

Adapting [Bacchus *et al.*, 1996], we define a **Non-Markovian Reward Decision Process (NMRDP)** as a tuple $\langle S, A, s_0, T, R, \gamma \rangle$, where S, A, s_0, T, γ are defined as in MDPs, and $R : (S \times A)^+ \times S \to \mathbb{R}$ is a *non-Markovian reward function* that maps finite state-action histories into a real value. The *discounted cumulative non-Markovian reward* received by the agent along a state-action-state trajectory $(s_0, a_0) \cdots (s_n, a_n) s_{n+1}$ is $\Sigma_{i=0}^{n} \gamma_i R(h_i, s_{i+1})$, where $h_i = (s_0, a_0) \cdots (s_i, a_i)$. Optimal strategies are those that maximize the expected discounted cumulative non-Markovian reward. A key aspect to observe is that, in contrast to MDPs with Markovian rewards, optimal solutions may not take the (simple, memoryless) form of policies anymore because they depend on the state history. They take the more general form of *strategies*, or mappings $\pi : (S \times A)^* \times S \to A$.

In the following section, we introduce reward machines, an automata-like canonical representation which is able to serve as a *lingua franca* for representing non-Markovian reward functions by leveraging the well-established correspondence between regular languages and automata. Following the introduction of reward machines, we catalogue a diversity of compelling regular languages that can serve as specifications languages for non-Markovian reward functions and can be automatically translated to reward machines.

4 Reward Machines

In this section, we define the notion of a reward machine (RM). As we will see, RMs are defined with respect to a set of propositional symbols that describe features or events of the concrete state of the environment. Returning to our Minecraft-like example, propositional symbols would include got_wood, $used\_factory$, and the like. Suppose that we have a set $\mathcal{P}$ of propositional symbols describing features or events. A labeling function is a function $L : S \times A \times S \to 2^{\mathcal{P}}$, i.e., it maps experiences to truth assignments over the vocabulary $\mathcal{P}$. We have defined L slightly more generally than the labeling functions in [Toro Icarte *et al.*, 2018b] by allowing the label to depend on the last action and previous state in addition to the current state. This makes it easier for labels to refer to events like movement that involve change of state. With the notion of a labeling function in hand, we can now describe RMs.

Intuitively, what an RM does is to indicate what reward function should currently be used to provide the reward signal, given the sequence of state labels (truth assignments in $\mathcal{P}$) that the agent has seen so far. Any RM can be thought of as a *Mealy machine* [Mealy, 1955] where the *input alphabet* is the set of possible state labels and the *output alphabet* is a set of reward functions.

Definition 1 *A* Mealy machine *is a tuple $\langle Q, q_0, \Sigma, \mathcal{R}, \delta, \rho \rangle$ where Q is a finite set of states, q_0 is the initial state, Σ is the finite input alphabet, $\mathcal{R}$ is the finite output alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, and $\rho : Q \times \Sigma \to \mathcal{R}$ is the output function.*

A Mealy machine takes input and produces output. On each step, the machine consumes an input symbol $\sigma \in \Sigma$, transitions from the state $q \in Q$ it started the step in to state $\delta(q, \sigma) \in Q$, and outputs the symbol $\rho(q, \sigma) \in \mathcal{R}$.

In order to formally define an RM, first suppose we have a finite set of (environment) states S (not to be confused with Mealy machine states), a finite set of actions A, a finite set of propositional symbols $\mathcal{P}$, and a labeling function $L : S \times A \times S \to 2^{\mathcal{P}}$. We will call $\langle S, A, \mathcal{P}, L \rangle$ a *setting*.

Definition 2 *A* reward machine (RM) *for the setting $\langle S, A, \mathcal{P}, L \rangle$ is a Mealy machine $\langle Q, q_0, \Sigma, \mathcal{R}, \delta, \rho \rangle$ where the input alphabet is $\Sigma = 2^{\mathcal{P}}$, and $\mathcal{R}$ is a finite set where each $R \in \mathcal{R}$ is a reward function from $S \times A \times S$ to $\mathbb{R}$,*

Toro Icarte *et al.* used RMs as compact representations of non-Markovian reward functions in NMRDPs. Upon performing its ith action, a_i, in an NMRDP, the agent gets the experience (s_i, a_i, s_{i+1}). If the current RM state was $q^i \in Q$, the RM then transitions to $q^{i+1} = \delta(q^i, \sigma_i)$, where $\sigma_i = L(s_i, a_i, s_{i+1})$. In doing so, the RM issues reward $r(s_i, a_{i+1}, s_{i+1})$, where $r = \rho(q^i, \sigma_i)$. This procedure implicitly defines a non-Markovian reward function.

Definition 3 *The non-Markovian reward function R* induced *by an RM $\langle Q, q_0, \Sigma, \mathcal{R}, \delta, \rho \rangle$, given a setting $\langle S, A, \mathcal{P}, L \rangle$, is $R((s_0, a_0) \cdots (s_n, a_n) s_{n+1}) := r(s_n, a_n, s_{n+1})$, where $r = \rho(q^n, \sigma_n)$ and q^n is defined inductively by $q^0 = q_0$ and $q^{i+1} = \delta(q^i, L(s_i, a_i, s_{i+1}))$.*

Relationship to Automata: Mealy machines are similar to *deterministic finite automata* (DFAs), except DFAs produce a binary output (accept/reject) given a input string, rather than an output string. Formally, a DFA is a tuple $\langle Q, \Sigma, q_0, \delta, F \rangle$, where Q is a finite set of automaton states, $q_0 \in Q$ is the initial state of the automaton, and $\delta : Q \times \Sigma \to Q$ is a transition function. Instead of an output function, a DFA has a set of *accepting states* $F \subseteq Q$. A DFA accepts an input string $\sigma_0 \cdots \sigma_{n-1} \in \Sigma^*$ iff $q^n \in F$, where $q^0 = q_0$ and $q^{i+1} = \delta(q^i, \sigma_i)$.

It s well-known that DFAs can recognize all and only regular languages. In the next section we consider various formal languages whose formulas can be converted into DFAs. These languages can be used to define reward functions by associating rewards with satisfaction of formulas (equivalently, with reaching accepting states in corresponding DFAs). We describe a two-step conversion of formulas with associated rewards into RMs, by first converting a formula into a DFA and then transforming the DFA into an RM by adding an appropriate output language and output function.

5 Formal Reward Specifications

5.1 Formal Languages and Temporal Logics

In this section, we review a myriad of formal languages that can be used to describe temporally extended behavior and, therefore, to specify when reward should be given to the agent. NMRDPs reward has previously been specified using a variety of temporal logics: PLTL [Bacchus *et al.*, 1996;

Bacchus *et al.*, 1997], $FLTL [Thiébaux *et al.*, 2006], and $\mathsf{LTL_f}$ [Camacho *et al.*, 2017; Brafman *et al.*, 2018]. With perhaps the exception of $FLTL, all of them can be transformed into DFA. Later on, we show how to exploit this property to construct suitable RMs that induce the same reward function as the reward specification.

Linear Temporal Logic *Linear Temporal Logic* (LTL) is a modal logic that extends propositional logic with temporal operators *next* ($\bigcirc$) and *until* ($\mathcal{U}$). The semantics of an LTL formula is evaluated over infinite-length state traces. Intuitively, $\bigcirc\alpha$ tells that α needs to hold in the next timestep, and $\alpha\mathcal{U}\beta$ tells that α needs to hold until β holds. Other modalities such as *eventually* ($\Diamond$) and *always* ($\Box$) are typically used. Within LTL, the *safe* and *co-safe* fragments are of special interest. Safe properties tell that a *bad prefix* never has to occur. In contrast, co-safe properties tell that a *good prefix* has to eventually occur. Safe and co-safe properties can be transformed into DFA with absorbing accepting states that recognize good and bad prefixes, respectively. The transformation is double exponential [Kupferman and Vardi, 2001], and efficient transformation tools exist (e.g. [Duret-Lutz *et al.*, 2016])[1].

Linear Temporal Logic on Finite Traces Different variants of LTL interpreted over finite traces have been studied (e.g. [Baier and McIlraith, 2006; De Giacomo and Vardi, 2013]). $\mathsf{LTL_f}$ is one of the recent examples. $\mathsf{LTL_f}$ can be transformed into DFA in double exponential time, and automated tools exist (e.g. [Baier and McIlraith, 2006; Zhu *et al.*, 2017]).

Linear Temporal Logic of the Past Another variant of LTL interpreted over finite traces is the *Linear Temporal Logic of the Past* (PLTL) [Emerson, 1990]. The syntax of PLTL extends propositional logic with modal operators *yesterday* ($\ominus$) and *since* ($\mathcal{S}$). The semantics of PLTL formulas are analogous to the semantics of $\mathsf{LTL_f}$ formulas, except that they look back in time from the current state, rather than looking to the future. PLTL can be transformed into DFA in double exponential time (cf. [Sohrabi *et al.*, 2011]).

Linear Dynamic Logic on Finite Traces Whereas $\mathsf{LTL_f}$ can be transformed into DFA, it is cannot capture all languages that can be captured with DFA. *Linear Dynamic Logic on Finite Traces* ($\mathsf{LDL_f}$) borrows the syntax of *Propositional Dynamic Logic* (PDL), and interprets it on finite traces. While the syntax of $\mathsf{LDL_f}$ is not as intuitive as $\mathsf{LTL_f}$, its expressiveness is the same as DFA and the transformation is still double exponential [De Giacomo and Vardi, 2013].

LTL with Regular Expressions for Finite Traces (LTL-RE) LTL-RE [Triantafillou *et al.*, 2015] has the same expressive power as $\mathsf{LDL_f}$, but with a user-friendly syntax.

Subset of TLA$^+$ The *Temporal Logic of Actions* (TLA$^+$) is a formal language used to design and verify concurrent systems [Lamport, 2002]. In TLA$^+$ it is possible to express behaviors in terms of states, actions, next-state relations, and temporal formulas. While its syntax allows for complex specifications, a subset of TLA$^+$ can be mapped into other temporal logics (e.g. $\mathsf{LTL_f}$), and therefore transformed into DFA.

Subset of Golog Golog (Algol in Logic) is a logic-based agent programming language. Its syntax includes procedural programming constructs such as if-then-else and while loops, together with non-deterministic choice of actions and arguments. Subsets of Golog can be transformed into DFA. For reference, see [Baier *et al.*, 2007; Baier *et al.*, 2008].

Regular Expressions Regular expressions are used to describe regular languages, which are all and only the languages that have a DFA representation. A regular expression over a finite alphabet of symbols Σ consists of constants and operator symbols. Constants are sets of strings (including the empty set, the set with an empty string, and strings with a single character from Σ). Operations over regular expressions include the concatenation, alternation, and Kleene star – which contains, for a string s the set of all strings that can be formed by concatenating any finite number of occurrences of s. Regular expressions can be transformed into DFA with, e.g., the Thompson's construction [Thompson, 1968].

5.2 Specifying Rewards with Temporal Logics

Following previous work on NMRDPs, we use formal languages as means to specify when to assign reward to the agent relative to their execution trace (Definition 4).[2] Intuitively, a pair $(r : \varphi)$ tells that reward r should be given when the sequence of experiences along execution trace $\pi = (s_1, a_1) \cdots (s_n, a_n)s_{n+1}$ satisfies a property described by φ. Note that the formulas have alphabet Σ. Thus, φ is evaluated over a sequence $\sigma_1, \ldots, \sigma_n$ of Σ-characters, where each $\sigma_i = L(s_i, a_i, s_{i+1})$ is obtained by projecting the ith experience by means of L.

The expressivity of our reward specifications differs from previous work on NMRDPs that considered rewards expressed by pairs $(r : \varphi)$. In previous work, states were assumed to be propositional, and φ was evaluated directly over the sequence of states along execution (cf. [Bacchus *et al.*, 1996; Camacho *et al.*, 2017; Brafman *et al.*, 2018]). In contrast, our reward specifications are evaluated over sequences of (projected) experiences, which contain information on the current state, action, and next state. This allow us to express richer properties. We formalize the concepts below.

Definition 4 *A reward specification is a set $R = \{(r_1 : \varphi_1), \ldots, (r_n : \varphi_n)\}$, where each r_i is a real value and φ_i is a formula over propositional variables Σ.*

As usual, let $\pi = (s_0, a_0)(s_1, a_1) \cdots (s_n, a_n)s_{n+1} \in (S \times A)^+ \times S$ be a sequence of states and actions representing the agent's execution history. The sequence of experiences received by the agent along π is the sequence $\{(s_i, a_i, s_{i+1})\}_{1 \leq i \leq n}$ – note that the second state in a experience overlaps with the first state in the next experience. We say that the projection of the experiences of π by L *entails* φ, and we write $\pi \models_L \varphi$, if $L(s_0, a_0, s_1) \cdots L(s_n, a_n, s_{n+1})$ entails φ. When L is clear from the context, we simply write $\pi \models \varphi$.

Following [Bacchus *et al.*, 1996], we define how rewards should be assigned for a reward specification (Definition 5).

[1] Spot software: https://spot.lrde.epita.fr/index.html

[2] Previous research in NMRDPs often do not make a clear distinction between what constitutes a non-Markovian reward function and the reward specification in terms of reward-formula pairs.

Definition 5 *For a setting $\langle P, L, S, A\rangle$, the non-Markovian reward function* induced *by a reward specification $R = \{(r_1 : \varphi_1), \ldots, (r_n : \varphi_n)\}$ assigns reward $R(\pi) \coloneqq \Sigma_{k=1}^{n} r_k \cdot \mathbb{1}(\pi \models \varphi_k)$ to a trace $\pi = (s_0, a_0) \cdots (s_n, a_n)s_{n+1} \in (S \times A)^+ \times S$.*

5.3 Canonical Reward Machine Transformations

We show here that reward machines can be constructed from a formal reward specification, provided that the reward formulas can be transformed into DFA– that is, represent regular languages. The rationale for transforming reward specifications into RMs is to adopt RMs as a canonical form to express non-Markovian reward. By doing so, off-the-shelf new techniques for reinforcement learning in NMRDPs, agnostic of the input language, can be developed and for use in a handful of reward specification languages.

Consider a setting $\langle S, A, \mathcal{P}, L\rangle$ and a reward specification $R = \{(r_1 : \varphi_1), \ldots (r_n : \varphi_n)\}$, where each φ_i is a formula that represents some regular language (expressed in e.g. $\mathsf{LTL_f}$). W.l.o.g., we assume all formulae share a common alphabet $\Sigma = 2^{\mathcal{P}\cup A}$. The construction of a RM consistent with R follows the steps below.

Step 1: Construction of the DFA In the first step, each φ_i is transformed into a DFA, $\mathcal{A}^{(i)}$. For safe and co-safe LTL, $\mathsf{LTL_f}$, and PLTL, the construction is double-exponential in the size of the formula.

Step 2: Construction of the RM Let $\mathcal{A}^{(i)} = \langle Q^{(i)}, \Sigma, q_0^{(i)}, \delta^{(i)}, F^{(i)}\rangle$ be DFA transformations of each φ_i, respectively. We define $M_R = \langle Q, \boldsymbol{q_0}, \Sigma, \mathcal{R}, \delta, \rho\rangle$ be the RM with components

$$
\begin{aligned}
Q &\coloneqq Q^{(1)} \times \cdots \times Q^{(n)} \\
\boldsymbol{q_0} &\coloneqq (q_0^{(1)}, \ldots, q_0^{(n)}) \\
\delta(\boldsymbol{q}, \sigma) &\coloneqq (\delta^{(1)}(q^{(1)}, \sigma), \ldots, \delta^{(n)}(q^{(n)}, \sigma)) \\
\rho(\boldsymbol{q}, \sigma) &\coloneqq \Sigma_{k=1}^{n} \rho^{(k)}(q^{(k)}, \sigma)
\end{aligned}
$$

where $\rho^{(k)}(q, \sigma)(s, a, s') \coloneqq r_k \cdot \mathbb{1}(\delta^{(k)}(q, \sigma) \in F^{(k)})$ and $\mathbb{1}(x)$ is the indicator function that evaluates to one if x is true, and zero otherwhise. Note that each $\rho^{(k)}(q, \sigma)$ is a constant function – that is, its value does not depend on (s, a, s'). As such, the space of reward functions $\mathcal{R}$ can be defined as the set containing, for each subset of $\{r_1, \ldots, r_n\}$, a constant reward function returning the sum of that subset.

The following theorems formulate the correctness of the construction, and provide bounds on the size of the RM with respect to the DFA transformations of reward formulae.

Theorem 1 *The RM M_R and R induce the same non-Markovian reward function.*

Proof sketch. Let $\pi = (s_0, a_0) \cdots (s_n, a_n)s_{n+1} \in (S \times A)^+ \times S$. We want to prove that the reward issued by the RM M_R after processing π equals the reward induced by R, which is $R(\pi) \coloneqq \Sigma_{k=1}^{n} r_k \cdot \mathbb{1}(\pi \models_L \varphi_k)$.

Let's see what is the reward induced by M_R on π. The states in M_R are vectors, where the kth component keeps track of the runs of the DFA on the projection of the experiences of π by L. Let $\boldsymbol{q_{n-1}} = (q_{n-1}^{(1)}, \ldots, q_{n-1}^{(n)})$ be the state of M_R after processing all minus one experience, i.e., $L(s_0, a_0, s_1) \cdots L(s_{n-1}, a_{n-1}, s_n)$. By construction of $\mathcal{A}^{(k)}$, it follows that $q_n^{(k)} = \delta(q_{n-1}^{(k)}, L(s_n, a_n, s_{n+1})) \in F^{(k)}$ iff $L(s_0, a_0, s_1) \cdots L(s_n, a_n, s_{n+1})$ entails φ_k. Using notation from Section 5.2, the latter condition reads $\pi \models_L \varphi_k$. Now, the reward issued by M_R after processing all the experiences is $\rho(\boldsymbol{q_{n-1}}, L(s_n, a_n, s_{n+1})) \coloneqq \Sigma_{k=1}^{n} \rho^{(k)}(q_{n-1}^{(k)}, L(s_n, a_n, s_{n+1}))$. The desired result is obtained by observing that $\rho^{(k)}(q, \sigma)(s, a, s') \coloneqq r_k \cdot \mathbb{1}(\delta^{(k)}(q, \sigma) \in F^{(k)})$, and $q_n^{(k)} \in F^{(k)}$ iff $\pi \models_L \varphi_k$. □

Theorem 2 *Let $R = \{(r_1 : \varphi_1), \ldots, (r_n : \varphi_n)\}$ be a reward specification, and let $\mathcal{A}^{(i)} = \langle Q^{(i)}, \Sigma, q_0^{(i)}, \delta^{(i)}, F^{(i)}\rangle$, $1 \leq i \leq n$, be a set of DFA transformations, one for each φ_i. A RM $M_R = \langle Q, \boldsymbol{q_0}, \Sigma, \mathcal{R}, \delta, \rho\rangle$ that induces the same non-Markovian reward function as R can be constructed such that:*

- *Q has no more than $|Q^{(1)}| \times \cdots \times |Q^{(n)}|$ states*
- *$\mathcal{R}$ has no more than 2^n reward functions*

Proof sketch. It follows from the construction of R_M and Theorem 1. □

6 Using QRM with Reward Shaping

A major advantage of using reward machines as a canonical representation is that it allows researchers to focus on how to exploit the information inside the reward machine for learning policies faster while keeping the advantages of specifying reward using different formal languages. In particular, two previous works have successfully exploited automata-structured reward functions for learning policies faster while keeping convergence guarantees.

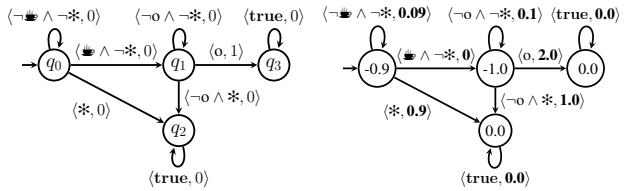
[Toro Icarte *et al.*, 2018b] proposed to decompose the problem using a reward machine specification of the reward function. Their approach, called *Q-Learning for RMs (QRM)*, learns one q-function $\tilde{q}_i(s, a)$ per state q_i from the reward machine. Then, given any experience state (s, a, s'), the RM is used to compute the reward $r_i = \delta_r(q_i, L(s, a, s'))$ that such experience would give the agent if performed from q_i. The next state in the reward machine can also be simulated $q_j = \delta_u(q_i, L(s, a, s'))$. This gives us all the information needed to perform a correct q-update for $\tilde{q}_i(s, a)$:

$$\tilde{q}_i(s, a) \xleftarrow{\alpha} r_i + \gamma \max_{a'} \tilde{q}_j(s', a')$$

This methodology was shown to be effective at learning policies in discrete and continuous environments.

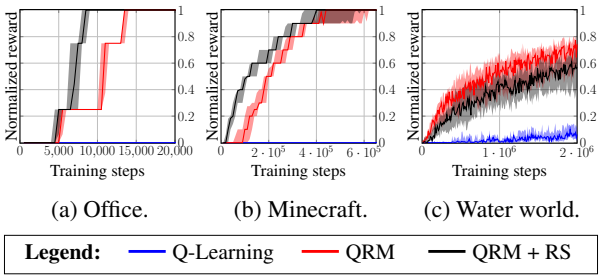
[Camacho *et al.*, 2017] proposed a different approach to exploit reward functions specified as LTL formulas based on potential-based reward shaping [Ng *et al.*, 1999]. The idea behind potential-based reward shaping is that some reward functions are easier to learn than others (even if they have the same optimal policy). Ng *et al.* proved that reward transformations based on potential functions over MDP states do not change the set of optimal policies. Formally, given any MDP $\mathcal{M} = \langle S, A, r, p, \gamma\rangle$ and potential function $\Phi : S \to \mathbb{R}$, changing the reward function of $\mathcal{M}$ by

$$r'(s, a, s') = r(s, a, s') + \gamma\Phi(s') - \Phi(s) \qquad (1)$$



(a) Before reward shaping. (b) After reward shaping

Figure 2: Reward shaping example.



(a) Office. (b) Minecraft. (c) Water world.

Figure 3: Results on the two tabular and one deep learning domain. Note the poor performance of Q-Learning.

will not change the set of optimal policies. In particular, [Camacho *et al.*, 2017] showed that effective potential functions Φ can be directly extracted from the DFA representation of LTL specifications of the reward function. More precisely, the authors define potential functions $\Phi(s, q)$, where s is the MDP state and q the automata state. Different potential functions were proposed that can be extracted from the automata structure, e.g. based on the distance between q and any accepting state in the automata – but without further lookahead. The construction of RMs from DFA presented in Section 5.3 suggests that similar potential functions can be used to perform reward shaping in QRM. We propose here a more clever way to compute potentials that are not merely based on the distance between q and a rewarding RM transition, but that approximate the cumulative discounted reward.

We propose a generalization of Camacho *et al.*'s approach that also works for RMs. We compute a potential-based function by running value iteration over the RM states. This is equivalent to transforming the RM $\langle Q, q_0, \Sigma, \mathcal{R}, \delta, \rho \rangle$ into a deterministic MDP $\mathcal{M} = \langle S, A, T, r, \gamma \rangle$, where $S = Q$, $A = 2^{\mathcal{P}}$, $T(q'|q, \sigma) = 1$ if $q' = \delta(q, \sigma)$ (zero otherwise), $r(q, \sigma, q') = \rho(q, \sigma)$, and $\gamma < 1$. The outcome of value iteration is the optimal value function per state in $\mathcal{M}$. This is $v^*(q) = \max_\sigma q^*(q, \sigma)$ where q^* is the optimal q-function and q is an state of $\mathcal{M}$. Then, we defined $\Phi(s, q) = -v^*(q)$ for every environment state s and RM state q. Figure 2 shows an example. The original reward machine from Figure 2a gives a reward of 1 to the agent when it gets to the office (represented by proposition o) after picking the coffee (represented by proposition ☕) as long as the agent does not break any furniture (represented by proposition ✻). In this pictorial representation of a reward machine, the nodes represent the states of the reward machine and the transitions are labelled by a pair $\langle c, r \rangle$ where c is a logical condition to transit from one state to the other and r is the reward that the agent receives. Figure 2b is the resulting reward machine after applying our automated version of reward shaping using $\gamma = 0.9$. We show the potential values on each node. The rewards on each transition was updated using (1). Intuitively, this reward machine gives larger rewards per step as the agent moves closer to the terminal state q_3. This will hopefully encourage the agent to learn policies that reaches q_3 faster.

We evaluated our approach on the same three environments proposed in [Toro Icarte *et al.*, 2018b]. The office and Minecraft domains are grid problems that are solved using tabular RL. The water domain is a continuous space problem that is solved using Deep RL. We used the exact same parameters and reward machines proposed in that paper and compare to a q-learning baselines and standard QRM without reward shaping. From these three domains, our approach to automatically compute potential-based reward shaping helped in 2 out of 3 domains. It is remarkable that by just including a small preprocesing step to adjust the rewards in the RM produces such large performance improvements. We also find it interesting that reward shaping did not help on the water world domain which is our only deep RL environment. This conterintuitive result opens possibilities for future research. To this end, we will release our code upon publication.

7 Discussion and Future Work

We examined two important challenges of Reinforcement Learning: (i) the difficulty of reward specification, and (ii) sample efficiency. We proposed the use of *Reward Machines* (RMs) as a canonical structure to represent reward functions. We provided a formal characterization of RMs in terms of Mealy machines and noted the association with automata, thereby providing a means to translate a diversity of goal and temporal property specification languages, augmented with scalar rewards, to RMs. We also presented an algorithm to realize the translation between automata and RMs. RMs expose the structure of the reward function to the learning agent, which when used in conjunction with RM-customized q-learning, results in a marked improvement in sample efficiency, and thus in faster realization of high-quality policies. The translation of all these formal languages to RMs obviates the need for numerous tailored q-learning algorithms, while supporting reward specification in a diversity of compelling languages. We proposed the use of reward shaping to enhance existing RM-tailored q-learning by computing a potential function that approximates the expected cumulative discounted reward based on the RM structure. Results were impressive in the tabular case, significantly outperforming QRM, the incumbent RM-tailored q-learning algorithm. In the Deep learning domain the use of reward shaping showed no added benefit over the already highly effective RM-tailored q-learning algorithm.

Formal languages present important advantages over traditional programming languages utilized for reward specification: they are compositional, some are declarative, and many support easy specification of temporally extended behavior. Our methods enable the use of a myriad of different languages to specify reward, while enjoying the advantage of language-tailored q-learning.

References

[Andreas *et al.*, 2017] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *ICML*, pages 166–175, 2017.

[Bacchus *et al.*, 1996] Fahiem Bacchus, Craig Boutilier, and Adam J. Grove. Rewarding behaviors. In *AAAI*, pages 1160–1167, 1996.

[Bacchus *et al.*, 1997] Fahiem Bacchus, Craig Boutilier, and Adam J. Grove. Structured solution methods for non-Markovian decision processes. In *AAAI*, pages 112–117, 1997.

[Baier and McIlraith, 2006] Jorge A. Baier and Sheila A. McIlraith. Planning with temporally extended goals using heuristic search. In *ICAPS*, pages 342–345, 2006.

[Baier *et al.*, 2007] Jorge A. Baier, Christian Fritz, and Sheila A. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In *ICAPS*, pages 26–33, 2007.

[Baier *et al.*, 2008] Jorge A. Baier, Christian Fritz, Meghyn Bienvenu, and Sheila McIlraith. Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In *AAAI, Nectar Track*, pages 1509–1512, 2008.

[Brafman *et al.*, 2018] Ronen I. Brafman, Giuseppe De Giacomo, and Fabio Patrizi. LTLf/LDLf non-Markovian rewards. In *AAAI*, pages 1771–1778, 2018.

[Camacho *et al.*, 2017] Alberto Camacho, Oscar Chen, Scott Sanner, and Sheila A. McIlraith. Non-Markovian rewards expressed in LTL: guiding search via reward shaping. In *SOCS*, pages 159–160, 2017.

[De Giacomo and Vardi, 2013] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, pages 854–860, 2013.

[Duret-Lutz *et al.*, 2016] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *ATVA*, volume 9938 of *LNCS*, pages 122–129. Springer, October 2016.

[Emerson, 1990] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. 1990.

[Hopcroft and Ullman, 1979] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Kupferman and Vardi, 2001] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[Lamport, 2002] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lu *et al.*, 2018] Tyler Lu, Dale Schuurmans, and Craig Boutilier. Non-delusional q-learning and value-iteration. In *NIPS*, pages 9971–9981, 2018.

[Mealy, 1955] George H Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, Sep. 1955.

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[Ng *et al.*, 1999] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 3, pages 278–287, 1999.

[Schaul *et al.*, 2015] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[Sohrabi *et al.*, 2011] Shirin Sohrabi, Jorge A. Baier, and Sheila A. McIlraith. Preferred explanations: Theory and generation via planning. In *AAAI-11*, pages 261–267, 2011.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[Thiébaux *et al.*, 2006] Sylvie Thiébaux, Charles Gretton, John K. Slaney, David Price, and Froduald Kabanza. Decision-theoretic planning with non-Markovian rewards. *Journal of Artificial Intelligence Research*, 25:17–74, 2006.

[Thompson, 1968] Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[Toro Icarte *et al.*, 2018a] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. Teaching multiple tasks to an RL agent using LTL. In *AAMAS*, pages 452–461, 2018.

[Toro Icarte *et al.*, 2018b] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *ICML*, pages 2112–2121, 2018.

[Triantafillou *et al.*, 2015] Eleni Triantafillou, Jorge A. Baier, and Sheila A. McIlraith. A unifying framework for planning with LTL and regular expressions. In *Proceedings of the 2nd Workshop on Model Checking and Automated Planning (MOCHAP-15)*, pages 23–31, 2015.

[Van Hasselt *et al.*, 2016] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[Watkins and Dayan, 1992] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[Zhu *et al.*, 2017] Shufang Zhu, Lucas M. Tabajara, Jianwen Li, Geguang Pu, and Moshe Y. Vardi. Symbolic LTLf synthesis. In *IJCAI*, pages 1362–1369, 2017.