LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning

Alberto Camacho[1,2*], **Rodrigo Toro Icarte**[1,2*], **Toryn Q. Klassen**[1], **Richard Valenzano**[3] and **Sheila A. McIlraith**[1,2]

[1]Department of Computer Science, University of Toronto, Toronto, Canada

[2]Vector Institute, Toronto, Canada

[3]Element AI, Toronto, Canada

{acamacho, rntoro, toryn, sheila}@cs.toronto.edu, rick.valenzano@elementai.com

Abstract

In Reinforcement Learning (RL), 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 reflect complex reward-worthy behavior. We propose using reward machines (RMs), which are automata-based representations that expose reward function structure, as a normal form representation for reward functions. We show how specifications of reward in various formal languages, including LTL and other regular languages, can be automatically translated into RMs, easing the burden of complex reward function specification. We then show how the exposed structure of the reward function can be exploited by tailored q-learning algorithms and automated reward shaping techniques in order to improve the sample efficiency of reinforcement learning methods. Experiments show that these RM-tailored techniques significantly outperform state-of-the-art (deep) RL algorithms, solving problems that otherwise cannot reasonably be solved by existing approaches.

1 Introduction

In Reinforcement Learning (RL), an agent that is unaware of the dynamics of its environment or its reward model must act exploratorily and learn from the resulting experience in order to find effective policies that maximize the expected cumulative reward. RL is particularly useful in complex environments that are difficult to model and therefore cannot be solved using other sequential decision making techniques.

Two challenges that plague many RL systems are: (i) the difficulty of reward specification, and (ii) sample complexity – the need for large numbers of training episodes. 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, many nontrivial reward functions deliver reward sparsely, necessitating millions of exploratory episodes to converge to a 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 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 machine* [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 normal form for representing 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 lan-

*Contact Author

guages, 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 automata-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 normal form enables the composition of reward functions initially specified in multiple specification languages.

In the latter part of the paper, we propose a new reward-machine-tailored q-learning algorithm that significantly enhances an existing algorithm through the exploitation of reward shaping (e.g, [Ng *et al.*, 1999]). Experiments with tabular and Deep q-learning confirm the merits of reward-machine-tailored q-learning techniques in improving sample efficiency and providing high-quality solutions, solving problems that otherwise cannot reasonably be solved using state-of-the-art RL algorithms. The work presented here leverages and unifies previous work of the authors (e.g., [Camacho *et al.*, 2017; Toro Icarte *et al.*, 2018a; Toro Icarte *et al.*, 2018b; Camacho *et al.*, 2018]).

2 Background on Reinforcement Learning

In reinforcement learning an agent interacts with an unknown environment to learn how to act in a manner that maximizes its expected cumulative reward [Sutton and Barto, 2018]. 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

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

where $x \overset{\alpha}{\leftarrow} y$ 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.

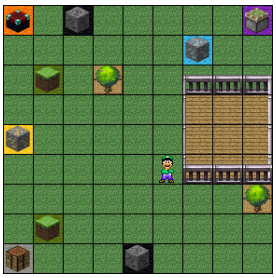

Figure 1: Minecraft-inspired domain containing natural resources, buildings where they can be processed, and perils such as zombies (adapted from [Andreas *et al.*, 2017]).

3 Specifying Reward Functions

An RL agent cannot inherently perceive reward from the environment. A programmer has to write the reward function (regardless of whether the environment behavior is manifest from a simulator or from the real world). Writing such reward functions is often challenging for the following reasons:

(a) The state representation may not be conducive to direct reward specification (for example, if it is pixels). It may not provide the appropriate level of abstraction – the appropriate building blocks – for the programmer.

(b) By definition, reward functions are Markovian. They typically map states, or states and actions, to a scalar reward value. Nevertheless, reward may actually be a consequence of some complex temporally extended behavior such as opening the freezer door, taking something out, and subsequently closing the freezer door. Such behavior may be difficult to capture directly with a Markovian reward function, depending on how much information is captured in a state. Reward is only conferred when the temporally extended behavior is completed.

To address the first issue, (a) above, we require a suitable vocabulary for reward specification. In some cases, this may be drawn from the variables that define the set of states, S. In other cases, the vocabulary for constructing rewards may be a set of features or properties that are extracted from the state or experiences via property detectors. For the purposes of this paper, we assume the existence of some vocabulary in terms of a set of propositional symbols, and of a *labeling function* that relates the agent's experiences to this vocabulary. For the programmer, this vocabulary will have to be selected and, where necessary, the labeling function constructed.

Definition 1 (vocabulary and labeling function) *A* vocabulary *is a set $\mathcal{P}$ of propositional symbols. 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 $\mathcal{P}$.*

While $\mathcal{P}$ is defined as a set of propositional symbols, it can be written in an equivalent lifted or explicitly relational representation to further expose semantic structure and to allow for parsimonious representation of the reward function. Further, we have defined L slightly more generally than the labeling functions used by 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. A common class of labeling function just involves the current state, or the current state and the action.

Illustrative example. Consider a variant of a Minecraft-inspired domain originally proposed by Andreas *et al.* [2017], 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. We want to specify rewards in terms of a set $\mathcal{P}$ of propositional symbols including got_wood, $used\_factory$, and the like, while a labelling function would identify to the agent when those propositions were true. Here are some English-specified examples of reward-worthy behavior.
[E1] *"Make a bridge by collecting wood and iron in any order, and using the factory afterwards."*
[E2] *"If it's night time, stay in the shed until daylight."*
[E3] *"Always avoid zombies."*
[E4] *"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."*
Associated with each behavior is a scalar value that denotes the reward conferred for its satisfaction.

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.

To address this second issue, (b) at the outset of this section, we adapt the work of Bacchus *et al.* [1996] to define the notion of a Non-Markovian Reward Decision Process.

Definition 2 (NMRDP) *A* Non-Markovian Reward Decision Process (NMRDP) *is a tuple $\langle S, A, s_0, T, R, \gamma \rangle$, where S, A, s_0, T, γ are defined as in MDPs, and (unlike in MDPs) $R : (S \times A)^+ \times S \to \mathbb{R}$ 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 structure for representing non-Markovian reward functions via a Markovian decomposition. In Section 5 we argue that reward machines provide a normal form 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. We catalogue a diversity of compelling formal languages, whose formulas describe (some or all) regular languages. These formal languages can serve as specification 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 vocabulary of propositional symbols $\mathcal{P}$ denoting features or events of the concrete state of the environment. We assume that the agent is given, or can detect, the truth or falsity of the proposition represented by each such symbol at all times – i.e., there is a labelling function (see Definition 1) available.

Intuitively, an RM indicates what (Markovian) 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 Markovian reward functions.

Definition 3 (Mealy machine) *A* Mealy machine *is a tuple $\langle Q, q_0, \Sigma, \mathcal{R}, \delta, \rho \rangle$ where Q is a finite set of states, $q_0 \in Q$ 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 4 (RM) *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 5 (induced by an RM) *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 $q^{i+1} = \delta(q^i, L(s_i, a_i, s_{i+1}))$.*

4.1 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 is well-known that DFAs can recognize all and only regular languages. In the next section we consider various formal languages whose formulas can be transformed 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 transformation of formulas with associated rewards into RMs, by first transforming a formula into a DFA and then transforming the DFA into an RM by adding an appropriate output language and output function.

5 Formal Language Specification

In the previous section, we saw how to specify non-Markovian reward functions using RMs. RMs not only provide benefit as a specification language, but they also expose reward function structure in a manner that can be exploited by tailored q-learning algorithms, as we discuss in Section 6. While RMs have many virtues, it may not be the case that the syntax of RMs is well-suited to easy, programmer-friendly specification of every type of reward-worthy behavior. 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 diversity of well-designed high-level specification languages. We argue for the use of formal languages as a natural alternative to programming languages (and RMs) for reward function specification. They are compositional, some are declarative, and many support easy specification of temporally extended behavior.

In this section, we propose to use RMs as a normal form representation for reward functions and as a *lingua franca* for reward specification that enables us to preserve the benefits of RMs in exposing reward function structure and providing for reward-function-tailored q-learning. To this end, we leverage the relationship between regular languages and automata, highlighting a selection of formal languages that can be used for reward function specification and that have established translations to automata. To facilitate the specification of reward functions, we are developing a tool to automatically generate reward machines from various formal languages.[1]

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]. As with RMs, each of these languages is defined using a vocabulary of propositional symbols $\mathcal{P}$ (or equivalent finite domain relational symbols) as described in Definition 1 (in Section 3). With perhaps the exception of $FLTL, any formula φ in these temporal logics can be transformed into a DFA that accepts all and only the traces that satisfy φ. Henceforth, we will refer to those DFAs as *transformations* of the formula. Later, we show how to exploit this property to construct 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 the temporal operators *next* ($\bigcirc$) and *until* ($\mathcal{U}$). The semantics of an LTL formula is evaluated over infinite-length state traces. Intuitively, $\bigcirc\alpha$ expresses that α needs to hold in the next timestep, and $\alpha \mathcal{U} \beta$ expresses 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 assert that a *bad prefix* never occurs. In contrast, co-safe properties assert that a *good prefix* will eventually occur. Safe and co-safe properties can be transformed into DFAs with absorbing accepting states that recognize good and bad prefixes, respectively. The transformation is double exponential [Kupferman and Vardi, 2001], but transformation tools that work well in practice exist (e.g. [Duret-Lutz *et al.*, 2016])[2].

Illustrative example (continued). Here we show how some of the previous examples of temporally extended behavior can be encoded as LTL formulae (cf. [Toro Icarte *et al.*, 2018a]), assuming the appropriate symbols within the vocabulary.

[E1] *"Make a bridge by collecting wood and iron in any order, and using the factory afterwards."*: $\Diamond(got\_wood \wedge \Diamond used\_factory) \wedge \Diamond(got\_iron \wedge \Diamond used\_factory)$

[E2] *"If it's night time, stay in the shed until daylight."*: $\Box(is\_night \rightarrow at\_shelter)$

[E3] *"Always avoid zombies."*: $\Box \neg near\_zombie$

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]).

[1] https://bitbucket.org/acamacho/fl2rm

[2] Spot software: https://spot.lrde.epita.fr

$\mathsf{LTL_f}$ is one of the recent examples. The syntax of $\mathsf{LTL_f}$ is the same as for LTL. For convenience, the macro $\mathsf{final} \coloneqq \neg\bigcirc\top$ is used to indicate the end of the trace. $\mathsf{LTL_f}$ can be transformed into DFA in double exponential time, and automated tools for this transformation are available (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* ($\mathcal{S}$), that have analogous semantics to the $\mathsf{LTL_f}$ operators *next* and *until*, except that they look back in time from the current state, rather than looking to the future. They are well-suited to specifying reward-worthy behavior [Bacchus *et al.*, 1996]. Similarly, PLTL has operators *always in the past* ($\boxminus$) and *sometime in the past* ($\diamondminus$), as well as the macro $\mathsf{start} \coloneqq \neg\ominus\top$. PLTL formulae 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 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 over 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 more 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 an alphabet $\mathcal{P}$ is constructed using constant symbols – the characters in $\mathcal{P}$, and special symbols that represent the empty language and the language that contains only the empty string – and the operations of concatenation, union, and the Kleene star. We note that it is also possible to define other operators like if-then-else and loops in terms of these operators. Regular expressions can be transformed into DFA with, e.g., Thompson's construction [Thompson, 1968].

5.2 Specifying Rewards with Temporal Logics

Following previous work on NMRDPs, we use formal languages as a means to specify when to assign reward to the agent relative to their execution trace (Definition 6). Intuitively, a pair $(r : \varphi)$ denotes that reward r is given when the sequence of experiences along execution trace $\tau = (s_1, a_1) \cdots (s_n, a_n)s_{n+1}$ satisfies a property described by φ. Note that the formulas have alphabet $\mathcal{P}$. 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 6 (reward specification) *A* reward specification *is a set $R = \{(r_1 : \varphi_1), \ldots, (r_n : \varphi_n)\}$, where each $r_i \in \mathbb{R}$ and φ_i is a formula over propositional variables $\mathcal{P}$.*

As usual, let $\tau = (s_0, a_0)(s_1, a_1) \cdots (s_n, a_n)s_{n+1} \in (S \times A)^+ \times S$ be a sequence of states and actions representing the agent's execution history. The sequence of experiences received by the agent along τ is the sequence $\{(s_i, a_i, s_{i+1})\}_{0 \leq i \leq n}$ – note that the second state in an experience overlaps with the first state in the next experience. We say that the projection of the experiences of τ by L *entails* φ, and we write $\tau \models_L \varphi$, if $L(s_0, a_0, s_1) \cdots L(s_n, a_n, s_{n+1})$ entails φ.

We will use $\mathbb{1}(x)$ to denote the indicator function which evalutes to 1 if the condition described by x is true and 0 otherwise. Following Bacchus *et al.* [1996], we define how rewards should be assigned for a reward specification:

Definition 7 (induced by a specification) *For a setting $\langle S, A, \mathcal{P}, L\rangle$, the non-Markovian reward function* induced *by a reward specification $R = \{(r_1 : \varphi_1), \ldots, (r_n : \varphi_n)\}$ assigns reward $R(\tau) \coloneqq \sum_{k=1}^{n} r_k \cdot \mathbb{1}(\tau \models_L \varphi_k)$ to a trace $\tau = (s_0, a_0) \cdots (s_n, a_n)s_{n+1} \in (S \times A)^+ \times S$.*

5.3 Constructing Reward Machines

We show here that reward machines can be constructed from a formal reward specification, provided that the reward formulas can be transformed into DFAs – that is, represent regular languages. The rationale for transforming reward specifications into RMs is to adopt RMs as a normal form to express non-Markovian reward. By doing so, new techniques for reinforcement learning in NMRDPs, agnostic of the input language, can be developed and used off-the-shelf for various 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. $\mathsf{LTL_f}$). Without loss of generality, we assume all formulae share a common alphabet $\mathcal{P}$. The construction of an 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, LTL_f, and PLTL, the construction is worst-case 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 $M_R = \langle Q, \boldsymbol{q_0}, \Sigma, \mathcal{R}, \delta, \rho \rangle$ as the RM with components

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

where $\rho^{(k)}(q, \sigma)(s, a, s') := r_k \cdot \mathbb{1}(\delta^{(k)}(q, \sigma) \in F^{(k)})$. Recall that $\mathbb{1}(x)$ is the indicator function that evaluates to one if x is true, and zero otherwise. 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 $\tau = (s_0, a_0) \cdots (s_n, a_n) s_{n+1} \in (S \times A)^+ \times S$. We want to prove that the reward issued by the RM M_R after processing τ equals the reward induced by R, which is $R(\tau) := \sum_{k=1}^{n} r_k \cdot \mathbb{1}(\tau \models_L \varphi_k)$.

Let $\boldsymbol{q_{n-1}} = (q_{n-1}^{(1)}, \ldots, q_{n-1}^{(n)})$ be the state of M_R after processing all but one of the experiences, i.e., after processing $L(s_0, a_0, s_1) \cdots L(s_{n-1}, a_{n-1}, s_n)$. By construction of the DFA $\mathcal{A}^{(k)}$, 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 (i.e., iff $\tau \models_L \varphi_k$, using the notation from Section 5.2). The reward issued by M_R after processing all the experiences is $\rho(\boldsymbol{q_{n-1}}, L(s_n, a_n, s_{n+1})) := \sum_{k=1}^{n} \rho^{(k)}(q_{n-1}^{(k)}, L(s_n, a_n, s_{n+1}))$. Finally, the desired result is obtained by observing that $\rho^{(k)}(q, \sigma)(s, a, s') := r_k \cdot \mathbb{1}(\delta^{(k)}(q, \sigma) \in F^{(k)})$, and $q_n^{(k)} \in F^{(k)}$ iff $\tau \models_L \varphi_k$. □

Theorem 2 *Let $R = \{(r_1 : \varphi_1), \ldots, (r_n : \varphi_n)\}$ be a reward specification, where for $1 \leq i \leq n$, φ_i can be transformed into a DFA with m_i states. An 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 $m_1 \times \cdots \times m_n$ states*
- *$\mathcal{R}$ has no more than 2^n reward functions*

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

6 Using QRM with Reward Shaping

Reinforcement learning algorithms typically treat the reward function as a black box that returns a scalar reward given a state or state-action pair. Representing a reward function as a reward machine exposes the reward function structure in a *normal form*, allowing for general learning algorithms that tailor to the problem-specific reward function. With automated translation of a multitude of specification languages to the RM normal form, it allows for specification in any of these languages while benefiting from powerful reward-function-tailored learning techniques.

In previous work, Toro Icarte *et al.* [2018a] proposed an algorithm for reward-function-tailored q-learning with LTL specifications of reward functions, called *LPOPL*. They subsequently developed *QRM*, a q-learning algorithm tailored to RM specifications of reward functions [Toro Icarte *et al.*, 2018b]. In both cases, reward-function-tailored q-learning significantly outperformed state-of-the-art (deep) q-learning algorithms while preserving convergence guarantees. In this section, we review *QRM* and propose a means of further exploiting the RM normal form structure via *reward shaping*. We show experimentally that reward shaping can be effectively combined with QRM.

Q-Learning for RMs (QRM) was introduced as a way to decompose and exploit a reward machine [Toro Icarte *et al.*, 2018b]. QRM learns one q-function $\tilde{q}_i(s, a)$ per RM state q_i. Then, given any experience (s, a, s'), the RM is used to compute the reward $r_i = \rho(q_i, L(s, a, s'))$ that the agent would receive for this experience, if it occurred while the RM was in state q_i. The next state in the reward machine can also be computed as $q_j = \delta(q_i, L(s, a, s'))$. This gives us all the information needed to perform a q-update for $\tilde{q}_i(s, a)$:

$$\tilde{q}_i(s, a) \overset{\alpha}{\leftarrow} r_i + \gamma \max_{a'} \tilde{q}_j(s', a') \tag{1}$$

for all $q_i \in Q$ simultaneously. The resulting method is guaranteed to converge to an optimal solution in the tabular case.

We propose to augment QRM with *reward shaping* [Ng *et al.*, 1999] over the RM. The intuition behind reward shaping is that some reward functions are easier to learn policies for than others, even if those functions have the same optimal policy. To that end, Ng *et al.* [1999] proved that given any MDP $\mathcal{M} = \langle S, A, s_0, T, r, \gamma \rangle$ and function $\Phi : S \to \mathbb{R}$, changing the reward function of $\mathcal{M}$ to

$$r'(s, a, s') = r(s, a, s') + \gamma \Phi(s') - \Phi(s) \tag{2}$$

will not change the set of optimal policies. Thus, if we find a function Φ – referred to as a *potential function* – that also allows us to learn optimal policies more quickly, we are guaranteed that the found policies are still optimal with respect to the original reward function.

In previous work, Camacho *et al.* [2017; 2018] proposed *reward shaping* over automata in service of finding a policy for a fully specified MDP with LTL-specified reward function. The potential functions considered took the form $\Phi(s, q)$, where s is the MDP state and q is the state in a DFA representation of the LTL formula. They proposed several methods for extracting potential functions from a DFA. One such function was based on the minimum number of transitions between q and any accepting state of the automata.

6.1 Reward Shaping with Value Iteration

The construction of RMs from a DFA presented in Section 5.3 suggests that potential functions could similarly be extracted

Algorithm 1 Value Iteration for Automatic Reward Shaping

Input: Q, δ, ρ, γ
for $q_i \in Q$ **do**
  $v(q_i) \leftarrow 0$ {initializing v-values}
$e \leftarrow 1$
while $e > 0$ **do**
  $e \leftarrow 0$
  for $q_i \in Q$ **do**
    $v' \leftarrow \max_{\delta(q_i,\sigma)=q_j} \rho(q_i,\sigma) + \gamma v(q_j)$
    $e = \max\{e, |v(q_i) - v'|\}$
    $v(q_i) \leftarrow v'$
return v

from a reward machine. In this section, we consider the use of *value iteration* over the RM states as a way to compute a potential function. Intuitively, the idea is to approximate the cumulative discounted reward of being in any RM state by treating the RM itself as an MDP.

Formally, given RM $\langle Q, q_0, \Sigma, \mathcal{R}, \delta, \rho \rangle$, we construct MDP $\mathcal{M} = \langle S, A, \emptyset, T, r, \gamma \rangle$, 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 $\gamma < 1$. Intuitively, this is a MDP where every transition in the RM corresponds to a deterministic action with the same reward. We note that $\mathcal{M}$ has no start state, since we have to compute the value of being in every state in $\mathcal{M}$ when using an optimal policy. That is, we compute $v^*(q) = \max_\sigma q^*(q,\sigma)$ where q^* is the optimal q-function and q is a state of $\mathcal{M}$. We do so using the well-known *value iteration* algorithm. The overall process of computing v^* using value iteration given Q, δ, ρ, and γ, is shown in Algorithm 1.

Once we have computed v^*, we then define the potential function as $\Phi(s,q) = -v^*(q)$ for every environment state s and RM state q. As we will see below, the use of negation encourages the agent to transition towards RM states that correspond to task completion. By construction, shaping using this potential function will still guarantee we converge to the optimal solution in the tabular case.

To make this approach more clear, consider an example task in what we refer to as the *office world* environment. In this grid world, the agent can move in the four cardinal directions. At certain locations, the agent can find coffee, mail, an office, and decorations. The task for this example is to bring the coffee (represented by proposition ☕), to the office (represented by proposition o), without stepping on any locations with decorations (represented by proposition ✻). This task can be written using the $\mathsf{LTL_f}$ formula $\varphi = \Box(\neg ✻) \wedge \Diamond(☕ \wedge \bigcirc\Diamond(o)) \wedge \Box(☕ \rightarrow \bigcirc\Box(o \leftrightarrow \mathsf{final}))$. That is, the agent should always avoid the decorations, eventually get the coffee and then eventually reach the office, and – any point after getting coffee – only reach the office at the final time (the last point is to ensure that the agent doesn't get rewarded a second time for being in the office after getting the coffee). The same behaviour can be captured in PLTL with the formula $\varphi' = \boxminus(\neg ✻) \wedge o \wedge \ominus(\diamondminus(☕)) \wedge \ominus\boxminus(o \rightarrow (\mathsf{start} \vee \neg\ominus\diamondminus(☕)))$. That is, the agent should have avoided the decorations at all times, be at the office, have been at the coffee, and (so reward is only given once) at any prior time when the agent was at the office it should not have earlier got the coffee.

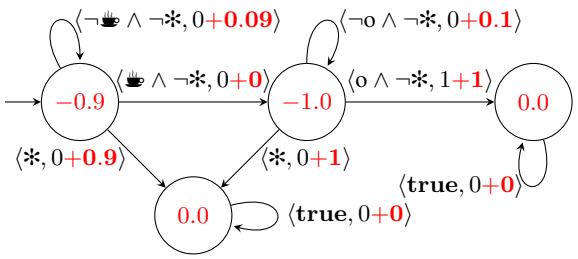

Figure 2: Reward shaping example with $\gamma = 0.9$.

Figure 2 shows an RM with the reward specification $R = \{(1 : \varphi)\}$ that gives a reward of 1 to the agent for satisfying the task φ. Nodes represent RM states, and each transition is labelled by a pair $\langle c, r + rs \rangle$, where c is a logical condition to transition between the states, r is the reward that the agent receives for the transition according to R, and rs is reward shaping applied to that transition. Applying our value iteration approach with $\gamma = 0.9$ gives the RM states the potential values indicated within each node. The reward shaping for each transition is calculated using equation (2). Notice that with reward shaping the agent is given positive reward for stepping on a decoration (i.e., transitioning to the bottom node). However, it only gets immediate reward, and is then unable to get further reward. In contrast, the agent can accumulate reward by avoiding the decorations. Moreover, the reward for completing the task (i.e., reaching the right-most node) is increased by reward shaping, thus ensuring that the optimal policy still completes the task as quickly as possible.

6.2 Experiments with RM-Based Reward Shaping

We tested the effectiveness of reward shaping using value iteration over a reward machine. In particular, we evaluated three approaches: a q-learning agent which takes the environment state and the current RM state as input, a QRM-based agent, and a QRM-based agent that uses reward shaping. We experimented using the same three environments, tasks, and reward machine specifications as Toro Icarte *et al.* [2018b]. We also used the same network architecture and setup when using Deep RL methods. Code used is publicly available.[3]

Our evaluation is in the *multi-task* setting, meaning that for each test environment, the agent is given a set of tasks to solve. For all tasks, the agent gets a reward of 1 only upon the task's completion, and receives a reward of 0 on all other steps. During experiments on an environment, we alternate between the tasks on an episode-by-episode basis.

The q-learning agent learns a separate q-function for each task. In contrast, QRM can exploit the RMs for the different tasks in the multi-task setting by updating the q-functions for all the states of all the RMs, not just the states of the current task's RM [Toro Icarte *et al.*, 2018b]. To do so, the RMs are used to simulate the state transition and reward seen for experience (s, a, s') from every state in every RM. We can then update the q-function for each of these RM states, as done using equation 1 in the case of a single RM.

[3] https://bitbucket.org/RToroIcarte/qrm

Experiments involved 3 test environments. The first was the office world described above. For this environment, we used four tasks including delivering the coffee and/or mail, and patrolling between several locations. In all tasks, the agent had to avoid stepping on the decorations.

The second environment was the *Minecraft* domain described above. In this grid world domain, the agent can move in the cardinal directions and pick up different raw materials in order to build objects. For our tests, we used variants of the ten object building tasks introduced by Andreas *et al.* [2017], with any unnecessary strict ordering of subtasks removed.

The third environment was a variant of the continuous *WaterWorld* domain [Karpathy, 2015]. Here, the agent moves around a continuous two-dimensional box, by changing its velocity in one of the four cardinal directions on every step. Different coloured balls are also moving around the environment. The ten tasks used all corresponded to touching different coloured balls, sometimes in a specific sequence. The environment is fully observable, unlike the original.

The results are shown in Figure 3. For these experiments, we used a discount factor γ of 0.9 and an exploration constant ϵ of 0.1. For the two grid world environments, we used a step size α of 1. In the WaterWorld, the step size was set to 0.00001. Each method was also run 30 times per environment. In the grid world environments, we used tabular RL, while we used neural networks to approximate the q-functions in the WaterWorld domain. To generate the charts, we paused the learning procedure every 100 steps for the grid worlds and every $1,000$ steps for the WaterWorld, and then tested the effectiveness of the current policy on every task. The plot shows the average performance over the 30 runs – as well as the 25th and 75th percentile – after having normalized the reward per task using the maximum discounted reward possible on that task. For example, an average value of 1 means that the agent has learned to perform optimally on all tasks in that environment over all 30 runs.

Figure 3 illustrates the powerful advantage that QRM-based approaches have over standard q-learning, as first observed by Toro Icarte *et al.* [2018b]. It further shows that combining reward shaping with QRM (denoted by "QRM + RS") leads to significant improvements in two of the three domains. We reiterate that this improvement is achieved with a simple, completely automatic preprocessing step that modifies the reward function while still maintaining optimality. We find it interesting that reward shaping did not help in the WaterWorld environment, the only domain in which we used deep RL. This counter-intuitive result opens possibilities for future research. Finally, our results encourage exploration of other learning algorithms that exploit the structure exposed by RMs (or other normal form representations).

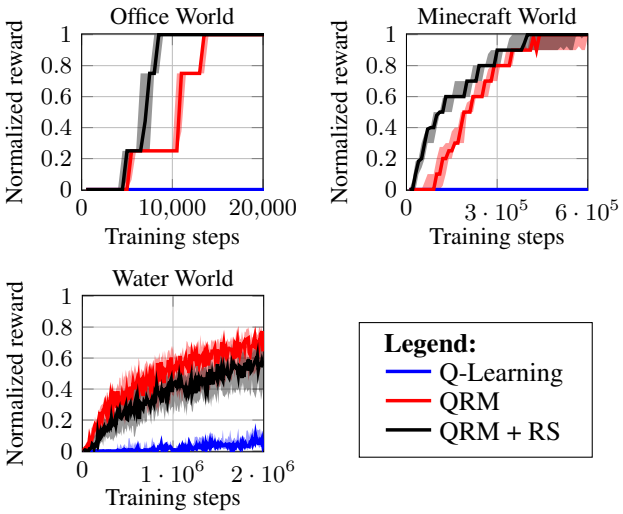

Figure 3: Results on two tabular and one deep learning domain.

7 Concluding Remarks

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 normal form to represent reward functions and as a *lingua franca* for reward functions initially specified in other languages. 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. Finally, we presented an algorithm to realize the translation between automata and RMs, providing a link to a tool under development for automatically generating RMs from various formal languages.

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. A link to the code used for experimenting with this method has been provided. Results were impressive in the tabular case, significantly outperforming QRM, the incumbent RM-tailored q-learning algorithm. In the Deep learning case, the use of reward shaping showed no added benefit over this already highly effective RM-tailored q-learning algorithm, though further investigation is needed.

Formal languages present important advantages over traditional programming languages used 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 reward-function-tailored q-learning.

Acknowledgements

We gratefully acknowledge funding from CONICYT (Becas Chile), the Natural Sciences and Engineering Research Council of Canada (NSERC), and Microsoft Research.

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.

[Camacho *et al.*, 2018] Alberto Camacho, Oscar Chen, Scott Sanner, and Sheila A. McIlraith. Non-markovian rewards expressed in LTL: Guiding search via reward shaping (extended version). In *GoalsRL, a workshop collocated with ICML/IJCAI/AAMAS*, 2018.

[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. Elsevier, 1990.

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

[Karpathy, 2015] Andrej Karpathy. REINFORCEjs: WaterWorld demo. Retrieved from http://cs.stanford.edu/people/karpathy/reinforcejs/waterworld.html, 2015.

[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 *NeurIPS*, 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, 2018] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT Press, second edition, 2018.

[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 *MOCHAP, a workshop collocated with ICAPS*, 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.