*ECAI 2024*
*U. Endriss et al. (Eds.)*


*doi:10.3233/FAIA240613*

# Maximally Permissive Reward Machines

**Giovanni Varricchione[a,*], Natasha Alechina[b,a], Mehdi Dastani[a] and Brian Logan[c,a]**

[a]Utrecht University
[b]Open University
[c]University of Aberdeen

**Abstract.** Reward machines allow the definition of rewards for temporally extended tasks and behaviors. Specifying "informative" reward machines can be challenging. One way to address this is to generate reward machines from a high-level abstract description of the learning environment, using techniques such as AI planning. However, previous planning-based approaches generate a reward machine based on a single (sequential or partial-order) plan, and do not allow maximum flexibility to the learning agent. In this paper we propose a new approach to synthesising reward machines which is based on the set of partial order plans for a goal. We prove that learning using such "*maximally permissive*" reward machines results in higher rewards than learning using RMs based on a single plan. We present experimental results which support our theoretical claims by showing that our approach obtains higher rewards than the single-plan approach in practice.

## 1 Introduction

Reward machines were introduced in [21] as a way of defining temporally extended (i.e., non-Markovian relative to the environment) tasks and behaviors. A *reward machine* (RM) is a Mealy machine where states represent abstract 'steps' or 'phases' in a task, and transitions correspond to observations of *high-level events* in the environment indicating that an abstract step/phase in the task has (or has not) been completed [2, 23]. The RM-based algorithm proposed in [2] has been shown to out-perform state-of-the-art RL algorithms, especially in tasks involving temporally extended behaviours. However, while learning with a reward machine is guaranteed to converge to an optimal policy *with respect to the reward machine*, in general RMs provide no guarantees that the resulting policy is optimal *with respect to the task* encoded by the reward machine. For example, a reward machine may specify that event $a$ should be observed before event $b$, while in some environment states, it may be more efficient to achieve $b$ before $a$. In general, for an RM-based policy to be optimal with respect to a task, the reward machine for the task must encode all possible ways the task can be achieved.

Another problem with reward machines is how to to generate them. While a declarative specification in terms of abstract steps or phases in a task is often easier to write than a conventional reward function, specifying a reward machine for a non-trivial task is challenging and prone to errors. Reward machines can be computed from task specifications expressed in a range of goal and property specification languages, including LTL and $\text{LTL}_f$, in a straightforward way [2]. However, reward machines generated from an abstract temporal formula may not expose significant task structure. Writing more "informative" specifications can be challenging, and, moreover, may inadvertently over-prescribe the order in which the steps are performed. One way to address this problem, is to generate a reward machine from a high-level abstract description of the learning environment, using techniques such as AI planning [11, 12], or (in a multi-agent setting) ATL model checking [24]. For example, Illanes et al. [11] consider a high-level model, in the form of a planning domain, of the environment in which the agent acts. They show how planning techniques can be used to synthesise a plan for a task, which is then used to generate a reward machine for that task. The reward machine is used to train a meta-controller for a hierarchical RL agent. The controller chooses which option (corresponding to an abstract action in the planning domain) to execute next. Their results indicate that an agent trained using a plan-based reward machine outperforms (is more sample efficient than) a standard HRL agent. They also show that reward machines based on partial-order plans outperform reward machines generated from sequential plans, arguing that this is because partial-order plans allow more ways of completing a task.

While the results presented by Illanes et al. are encouraging, their approach does not allow maximum flexibility to the agent, and thus cannot ensure learning an optimal policy for the task. The reward machine they generate is based on a single partial-order plan. In many cases, a goal may be achieved by different plans and each plan might be more appropriate in different circumstances, e.g., depending on the agent's location, or the resources available.

In this paper we propose a new approach to synthesising reward machines which is based on the *set of partial-order plans for a goal*. We present an algorithm which computes the set of partial-order plans for a planning task, and give a construction for synthesising a *maximally permissive* reward machine (MPRM) from a set of partial-order plans. We prove that the expected discounted future reward of optimal policies learned using an MPRM is greater than or equal to that obtained from optimal policies learned using a RM synthesised from any single partial-order plan. We introduce a notion of the *adequacy* of planning domain abstractions, which intuitively characterises when a planning domain captures all the relevant features of an MDP, and prove that the expected reward of an optimal policy learned using an MPRM synthesised from a *goal-adequate* planning domain is the same as that of an optimal policy for the underlying MDP. Finally, we evaluate MPRMs using three tasks in the CRAFTWORLD environment [1] used in [11, 12], and show that the agent obtains higher reward than with RMs based either on a single partial-order plan or on a single sequential plan.

* Corresponding Author. Email: g.varricchione@uu.nl

## 2 Preliminaries

In this section, we provide formal preliminaries for both reinforcement learning and planning.

### 2.1 Reinforcement Learning

In RL the model in which agents act and learn is generally assumed to be a *Markov Decision Process* (MDP) $M = \langle S, A, r, p, \gamma \rangle$, where $S$ is the set of states, $A$ is the set of actions, $r : S \times A \times S \to \mathbb{R}$ is the reward function, $p : S \times A \to \Delta(S)$ is the transition function, and $\gamma \in [0, 1]$ is the discount factor. It is assumed that the agent does not have access to the model in which it acts, i.e., $r$ and $p$ are hidden to it. The agent's goal in RL is to learn a *policy* $\rho : S \to \Delta(A)$, i.e., a map from each state of the MDP to a probability distribution over the set of actions. In particular, we are mostly interested in so-called "*optimal policies*", i.e., policies that maximise the expected discounted future reward from any state $s \in S$:

$$\rho^* = \arg\max_{\rho} \sum_{s \in S} v_\rho(s)$$

where $v_\rho(s)$ is the "*value function*", i.e., the expected discounted future reward obtained from state $s$ by following policy $\rho$:

$$v_\rho(s) = \mathbb{E}_\rho \left[ \sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right]$$

where $r_t$ is the reward obtained at timestep $t$.

As the MDP's dynamics and reward are hidden, the agent is supposed to learn a policy by trial and error. This is achieved by the agent taking an "exploratory" action $a$ in a state $s$, and observing which state $s'$ (sampled from $p(s, a)$) is reached and the reward $r' = r(s, a, s')$ that is obtained. By collecting these experiences $(s, a, s', r') \in S \times A \times S \times \mathbb{R}$, or "*samples*", the agent can learn a policy $\rho$ via RL algorithms, such as *Q-learning* [26].

### 2.2 Labelled MDPs

As in this work we assume the presence of planning domains and reward machines, we also assume that we are given a so-called "*labelling function*" $L$. This function will be the link between the low-level MDP, in which agents learn how to act, and the high-level planning domain and reward machine, which describe how agents can achieve a task using high-level symbols and actions.

**Definition 2.1** (Labelled MDP)**.** Let $\mathcal{P}$ be a set of propositional symbols. Then, a *labelled MDP* is a tuple $\mathcal{M} = \langle S, A, r, p, \gamma, L \rangle$, where $S, a, r, p$ and $\gamma$ are as in an MDP, and $L : S \to 2^{\mathcal{P}}$ is the labelling function, mapping each state of the MDP to a set of propositional symbols.

### 2.3 Reward Machines

Reward machines [23] are a tool recently introduced in the RL literature to define non-Markovian reward functions via finite state automata. Let $\mathcal{M} = \langle S, A, r, p, \gamma, L \rangle$ be a labelled MDP for some set of propositional symbols $\mathcal{P}$.

**Definition 2.2** (Reward Machine)**.** A reward machine (RM) is a tuple $\mathcal{R} = \langle U, u_0, \Sigma, \delta_u, \delta_r \rangle$, where $U$ is the set of states of the RM, $u_0$ is the initial state, $\Sigma \subseteq 2^{\mathcal{P}}$ is the input alphabet, $\delta_u : U \times \Sigma \to U$ is the state transition function, and $\delta_r : U \times U \to \mathbb{R}$ is the reward transition function.

When using RMs, training is usually done over the so-called "*product*" between the labelled MDP and the RM, also known as a "*Markov Decision Process with a Reward Machine*" (MDPRM) [23].

**Definition 2.3** (MDPRM)**.** A *Markov Decision Process with a Reward Machine* (MDPRM) is a tuple $\mathcal{M} = \langle S, A, p, \gamma, L, U, u_0, \delta_u, \delta_r \rangle$, where $S, A, p, \gamma, L$ are as in the definition of a labelled MDP, and $U, u_0, \Sigma, \delta_u$ and $\delta_r$ are as in the definition of a reward machine.

At each timestep, the RM is in some state $u$. As the agent moves the MDP into state $s'$, the RM updates its internal state via the observation $L(s')$, i.e., the new RM state is $u' = \delta_u(u, L(s'))$. Accordingly, the RM also outputs the reward $\delta_r(u, u')$, which is the reward the agent obtains. As in "vanilla" MDPs, the agent learns a policy by taking exploratory actions and collecting rewards from the RM's reward function $\delta_r$. Thus, samples include also the states of the RM, i.e., each sample is a tuple $(s, u, a, s', u', r') \in S \times U \times \mathbb{R} \times S \times U$. For this reason, any RL algorithm that works with standard MDPs can also be used in MDPRMs. Moreover, algorithms exploiting access to the RM have also been proposed, e.g., CRM [23].

### 2.4 Symbolic Planning

A *planning domain* $\mathcal{D} = \langle \mathcal{F}, \mathcal{A} \rangle$, is a pair where $\mathcal{F} \subseteq \mathcal{P}$ is a set of *fluents* (propositions), and $\mathcal{A}$ is a set of *planning actions*. *Planning states* are subsets $\mathcal{S} \subseteq \mathcal{F}$, where a proposition is in $\mathcal{S}$ if and only if it is true in $\mathcal{S}$. Actions $a \in \mathcal{A}$ are tuples $a = \langle pre^+, pre^-, \mathit{eff}^+, \mathit{eff}^- \rangle$ such that each element of $a$ is a subset of $\mathcal{F}$, $pre^+ \cap pre^- = \emptyset$ and $\mathit{eff}^+ \cap \mathit{eff}^- = \emptyset$. The "*pre*" sets are the sets of "*preconditions*", whereas the "*eff*" are the sets of "*effects*", or "*postconditions*". $pre^+$ are the propositions that must be true to perform the action, whereas $pre^-$ those that must be false. Analogously, $\mathit{eff}^+$ are the propositions that are made true by the action, whereas $\mathit{eff}^-$ those that are made false. Thus, an action $a$ can be executed from a planning state $\mathcal{S}$ if and only if $pre^+ \subseteq \mathcal{S}$ and $pre^- \cap \mathcal{S} = \emptyset$. Executing action $a$ in state $\mathcal{S}$ results in the new state $\mathcal{S}' = \left(\mathcal{S} \setminus \mathit{eff}^-\right) \cup \mathit{eff}^+$. Given an MDP $\mathcal{M}$ and a planning domain $\mathcal{D}$, we assume that $A \cap \mathcal{A} = \emptyset$, i.e., the planning actions are not the same as the actions the agent can perform in the MDP. Intuitively, the planning actions can be seen as *high-level* or abstract actions which correspond to sequences of actions in the MDP. For example, in a Minecraft-like scenario where the agent can move up, down, left, and right on a grid, a planning action might be "*get wood*" corresponding to a sequence of movement actions ending in a cell containing wood.

As an example of a planning domain, consider the CRAFTWORLD environment [1] in which an agent moves in a grid and has to gather resources which can then be used to produce items.[1] For example, the agent can build a bridge in order to reach the gold ore. We assume the agent can build two different types of bridge: an iron bridge or a rope bridge. The iron bridge requires gathering wood and iron and then processing them in a factory. The rope bridge requires gathering grass and wood and processing them in a toolshed. The corresponding planning domain $\mathcal{D}$ can be formalised as:

$$\begin{aligned} \langle \mathcal{F} = \{&\texttt{has-wood}, \texttt{has-grass}, \\ &\texttt{has-iron}, \texttt{has-bridge}\}, \\ \mathcal{A} = \{&\texttt{get-wood}, \texttt{get-grass}, \texttt{get-iron}, \\ &\texttt{use-factory}, \texttt{use-toolshed}\}\rangle \end{aligned}$$

[1] CRAFTWORLD is based on the popular video game Minecraft, and was used as a test domain in [11, 12].

The get-x actions have no preconditions, and only one positive postcondition, i.e., that has-x is true. The use-factory action has the preconditions has-wood and has-iron, the positive postcondition has-bridge, and the negative postconditions, has-wood and has-iron, i.e., the use-factory action makes a bridge, "consuming" the resources collected by the agent in the process. The use-toolshed action has has-wood and has-grass as preconditions, the positive postcondition has-bridge, and the negative postconditions, has-wood and has-grass.

A *planning task* is a triple $\mathcal{T} = \langle \mathcal{D}, \mathcal{S}_I, \mathcal{G} \rangle$, where $\mathcal{D}$ is a planning domain, $\mathcal{S}_I$ is the initial planning state, and $\mathcal{G} = \langle \mathcal{G}^+, \mathcal{G}^- \rangle$ is a pair containing two subsets of $\mathcal{F}$ which are disjoint. Any planning state $\mathcal{S}$ such that $\mathcal{G}^+ \subseteq \mathcal{S}$ and $\mathcal{S} \cap \mathcal{G}^- = \emptyset$ is a *goal state*. For example, the planning task to build a bridge is given by the domain $\mathcal{D}$ we have previously defined, the initial state $\mathcal{S}_I = \emptyset$, and the goal $\mathcal{G} = \langle \{\texttt{has-bridge}\}, \emptyset \rangle$.

A *sequential plan* $\pi = [a_0, \ldots, a_n]$ for a planning task $\mathcal{T}$ is a sequence of planning actions $a_i \in \mathcal{A}$ such that: (i) it is possible to execute them sequentially starting from $\mathcal{S}_I$, and (ii) by doing so, the planning domain reaches a goal state. For example, the following sequential plan allows the agent to produce a rope bridge:

$$[\texttt{get-wood}, \texttt{get-grass}, \texttt{use-toolshed}]$$

A *partial-order plan* (POP) $\overline{\pi} = \langle \mathcal{A}', \prec \rangle$ is a pair where $\mathcal{A}'$ is a multiset of actions from $\mathcal{A}$ and $\prec$ is a partial order over $\mathcal{A}'$ [3, 16]. We write $a \prec a'$ to denote $(a, a') \in \prec$, meaning that action $a$ must be performed before action $a'$. For example, the following partial-order plan allows the agent to produce an iron bridge:

$$\overline{\pi}_{\texttt{iron-bridge}} = \langle \{\texttt{get-wood}, \texttt{get-iron}, \texttt{use-factory}\}, \{\texttt{get-wood} \prec \texttt{use-factory}, \texttt{get-iron} \prec \texttt{use-factory}\} \rangle$$

Sequential plans are a special case of partial-order plans where $\prec$ is a total order. In general, a partial-order plan corresponds to a *set* of sequential plans, i.e., the set of all sequential plans that can be obtained by extending the partial order $\prec$ to a total order (referred to as a "*linearisation*" of the partial-order plan). Compared to sequential plans, partial-order plans allow the agent greater flexibility in choosing the order in which actions are executed. While a sequential plan constrains the agent to follow the total order of the plan, with a partial-order plan the agent can perform any action $a$, so long as all actions $a'$ such that $a' \prec a$ have already been executed.

Typically, given a planning task, a partial-order planner, e.g., [18], returns a single partial-order plan $\overline{\pi} = \langle \mathcal{A}', \prec \rangle$. However, in general, a planning task can be achieved using multiple partial-order plans, i.e., plans $\overline{\pi}'$ where the actions in $\mathcal{A}'$ are ordered differently, or which use different multisets of actions.

**Definition 2.4** (Set of all partial-order plans)**.** The *set of all partial-order plans for a planning task* $\langle \mathcal{D}, \mathcal{S}_I, \mathcal{G} \rangle$, $\overline{\Pi}$, is the set of plans $\langle \mathcal{A}', \prec \rangle$ where $\mathcal{A}' \subseteq \mathcal{A}$ and any linearisation $[a_0, \ldots, a_n]$ of $\mathcal{A}'$ consistent with $\prec$ results in a goal state $\mathcal{S}$, i.e., $\mathcal{G}^+ \subseteq \mathcal{S}$ and $\mathcal{G}^- \cap \mathcal{S} = \emptyset$.

It is straightforward to give an algorithm that returns the set of all partial-order plans $\overline{\Pi}$ for a planning task, see Algorithm 1. We assume the following definitions. $steps(\overline{\pi})$ is the multiset of actions in the plan $\overline{\pi}$ and $ord(\overline{\pi})$ is the set of ordering constraints. In addition,

**Algorithm 1** Compute the set of all partial-order plans

1: $\overline{\pi} \leftarrow \langle \{start, finish\}, \{start \prec finish\} \rangle$
2: $\overline{\Pi} \leftarrow \emptyset$
3: **procedure** POP-PLAN($\overline{\pi}$)
4: $\quad open \leftarrow$ open preconditions $\in steps(\overline{\pi})$
5: $\quad$ **if** $open = \emptyset$ **then**
6: $\quad\quad \overline{\Pi} \leftarrow \overline{\Pi} \cup \{\langle steps(\overline{\pi}) \setminus \{start, finish\}, ord(\overline{\pi}) \setminus \{start \prec finish\} \rangle\}$
7: $\quad$ **else**
8: $\quad\quad$ **for** $a \in steps(\overline{\pi})$ s.t. $p \in pre(a) \wedge p \in open$ **do**
9: $\quad\quad\quad$ **for** $a' \in \mathcal{A} \cup steps(\overline{\pi})$ s.t. $p \in \mathit{eff}(a')$ **do**
10: $\quad\quad\quad\quad$ **if** $a'$ is new **then**
11: $\quad\quad\quad\quad\quad \overline{\pi} \leftarrow \langle steps(\overline{\pi}) \cup \{a'\}, ord(\overline{\pi}) \cup \{start \prec a' \prec finish\} \rangle$
12: $\quad\quad\quad\quad ord(\overline{\pi}) \leftarrow ord(\overline{\pi}) \cup \{a' \prec a\}$
13: $\quad\quad\quad\quad links(\overline{\pi}) \leftarrow links(\overline{\pi}) \cup \{(a', p, a)\}$
14: $\quad\quad\quad\quad$ ORDER($\overline{\pi}, a', p, a$)
15: **procedure** ORDER($\overline{\pi}, a', p, a$)
16: $\quad threats \leftarrow \{(a_i, a_j) \mid (a_i, \neg p, a_j) \in links(\overline{\pi})\}$
17: $\quad$ **if** $threats \neq \emptyset$ **then**
18: $\quad\quad cons \leftarrow \{\{o_1, \ldots, o_n\} \mid (a_j, a_k)_i \in threats \wedge o_i = a \prec a_j \text{ or } o_i = a_k \prec a'\}$
19: $\quad\quad$ **for** $c \in cons$ **do**
20: $\quad\quad\quad$ **if** $ord(\overline{\pi}) \cup c$ is consistent **then**
21: $\quad\quad\quad\quad ord(\overline{\pi}) \leftarrow ord(\overline{\pi}) \cup c$
22: $\quad\quad\quad\quad$ POP-PLAN($\overline{\pi}$)
23: $\quad$ **else**
24: $\quad\quad$ POP-PLAN($\overline{\pi}$)

the algorithm maintains a set $links(\overline{\pi})$ of *causal links* of the form $(a', p, a)$ where $a'$ and $a$ are steps and $p$ is a literal in the postcondition of $a'$ and in the precondition of $a$. Causal links record the reason for adding step $a'$ to the plan (in order to establish precondition of $a$), and are used to generate ordering constraints. A step $a''$ *threatens* a causal link $(a', p, a)$ if $a''$ makes $p$ false. To resolve the threat $a''$ should be placed either before $a'$ in the order, or after $a$. An ordering is *consistent* if it is transitive and does not contain cycles, i.e., there is no $a_i, a_j$ such that $(a_i \prec a_j), (a_j \prec a_i) \in ord$. Given a planning action $a$, $pre(a)$ is the set containing the positive and negative literals of the propositional symbols appearing in the sets $pre^+$ and $pre^-$ of $a$, and $\mathit{eff}(a)$ is the set of positive and negative literals in $\mathit{eff}^+$ and $\mathit{eff}^-$. A precondition $p$ of a step $a$ is termed *open* if there is no causal link $(a', p, a) \in links(\overline{\pi})$ establishing $p$. A plan is *complete* if it has no open preconditions. Initially, the set of plans is empty, and $\overline{\pi}$ is initialised to a plan consisting of two steps: *start* and *finish*: *start* has no preconditions and the initial state $\mathcal{S}_I$ as a postcondition; *finish* has no postconditions and the goal $\mathcal{G}$ as a precondition. $ord$ contains the single ordering constraint $\{start \prec finish\}$, and $links$ is empty.

The procedure POP-PLAN takes a partial-order plan $\overline{\pi}$ as input. If $\overline{\pi}$ has no open preconditions, i.e., the plan is complete, then we remove the steps *start* and *finish*, add it to the set of plans, and POP-PLAN returns (lines 5-6). Otherwise, we iterate over each open precondition in the set of open preconditions, $open$ (lines 8-14). For each open precondition $p$, an action $a'$ from the set of actions $\mathcal{A}$ of the planning domain is chosen which establishes $p$ (line 9; if there are no actions which establish $p$, i.e., the plan cannot be extended to a complete plan, this branch of the computation terminates and $\overline{\pi}$ is discarded). The procedure ORDER is then called (line 14) to resolve any threats introduced by the addition of $a'$. If there are no threats, then POP-

PLAN is called again with the updated $\overline{\pi}$ containing $a'$ (lines 23-24). Instead, if there exists at least a threat, the set of sets of ordering constraints $cons$ (line 18) contains all possible ways of safeguarding each threatened link $(a_i, \neg p, a_j)$. For each such set of ordering constraints $c$, if $c$ is consistent with the current ordering constraints in $ord(\overline{\pi})$, they are added to $ord(\overline{\pi})$, and POP-PLAN is called to extend the plan for each possible ordering of actions (lines 19-22). When a plan is found, we backtrack and continue from the 'closest' enclosing **for** loop (which may be iterating over sets of ordering constraints in ORDER, or actions $a'$ and open preconditions $p$ in POP-PLAN) to search for alternative ways of extending the incomplete plan $\overline{\pi}$. Algorithm 1 runs in EXPSPACE, as the set of partial order plans is in the worst case exponential in the number of actions in $\mathcal{A}$. In practice, this is often not an issue: the planning domain is an abstraction of the underlying MDP, and the number of actions is typically small.

For the bridge task, the algorithm would produce another partial-order plan, in which the agent builds a rope bridge using grass:

$$\begin{aligned}&\overline{\pi}_{\texttt{rope-bridge}} = \\ &\langle \mathcal{A} = \{\texttt{get-wood}, \texttt{get-grass}, \texttt{use-toolshed}\}, \\ &\quad \prec = \{\texttt{get-wood} \prec \texttt{use-toolshed}, \\ &\qquad \texttt{get-grass} \prec \texttt{use-toolshed}\}\rangle\end{aligned}$$

Thus giving us the set of all partial-order plans for the bridge task: $\overline{\Pi}_{\texttt{bridge}} = \{\overline{\pi}_{\texttt{iron-bridge}}, \overline{\pi}_{\texttt{rope-bridge}}\}$. In the Appendix[2], we also provide all sequential plans that can be obtained by linearising the POPs in $\overline{\Pi}_{\texttt{bridge}}$.

## 3 Maximally Permissive Reward Machines

In this section, we show how the set of all partial-order plans, $\overline{\Pi}$, for a planning task $\mathcal{T} = \langle \mathcal{D}, \mathcal{S}_I, \mathcal{G} \rangle$, can be used to synthesise a reward machine $\mathcal{R}_{\overline{\Pi}}$ that is *maximally permissive*, i.e., which allows the agent maximum flexibility in learning a policy.

Let $\Pi$ be the set of all linearisations $\pi$ of all the partial-order plans in $\overline{\Pi}$. We denote by $pref(\pi)$ the set of all proper prefixes (of arbitrary length) of $\pi \in \Pi$. Note that the prefixes are finite, as the set of actions in $\overline{\Pi}$ is finite. Then, let $states(pref(\pi))$ be the set of sequences of planning states that is induced by the prefixes in $pref(\pi)$, assuming that the initial planning state is $\mathcal{S}_I$. We denote with $steps(\pi)$ the set of actions in a sequential plan, and with $post(\mathcal{A}')$ the set containing $post(a)$, as defined in Section 2.4, for each planning action $a \in \mathcal{A}'$. For an arbitrary sequence of planning states $u$, we denote with $last(u)$ the last element of the sequence. For sets of literals $P$, we, respectively, denote with $P^+$ and $P^-$ the sets of propositional symbols with positive and negative literals in $P$.

**Construction 1** (Maximally Permissive Reward Machine (MPRM))**.** *Fix the set of all partial-order plans $\overline{\Pi} = \{\overline{\pi}_1, \ldots, \overline{\pi}_n\}$ for some planning task $\mathcal{T} = \langle \mathcal{D}, \mathcal{S}_I, \mathcal{G} \rangle$. Then, $\mathcal{R}_{\overline{\Pi}}$, the maximally permissive RM corresponding to $\overline{\Pi}$, is defined as follows:*

- $U = \left[\bigcup_{\pi \in \Pi} states(pref(\pi))\right] \cup \{u_g\}$;
- $u_0 = [\mathcal{S}_I]$;
- $\Sigma = \bigcup_{\pi \in \Pi} post(steps(\pi))$;
- $\delta_u(u, P) = u\mathcal{S}$, where $\mathcal{S} = \left(last(u) \setminus P^-\right) \cup P^+$ and $u\mathcal{S} \in states(pref(\pi))$ for some linearisation $\pi \in \Pi$, or $= u_g$ if $\mathcal{G}^+ \subseteq \mathcal{S}$ and $\mathcal{G}^- \cap \mathcal{S} = \emptyset$;
- $\delta_r(u, u') = \begin{cases} 0 & \text{if } u' = u_g \\ -1 & \text{otherwise.} \end{cases}$

[2] A version of this paper including the appendix can be found on arXiv [25].

In the construction of the MPRM, the set of states correspond to the set of all possible prefixes of planning states across all POPs in the set used to build the reward machine. Then, the RM transitions from a state $u$ to a state $u' = u\mathcal{S}$ when it observes the set of propositional symbols $P$ which are exactly the conditions such that $\mathcal{S} = \left(last(u) \setminus P^-\right) \cup P^+$ and, most importantly, $u\mathcal{S}$ is a prefix of some linearisation of a POP in $\overline{\Pi}$. As soon as the RM "reaches" a sequence of states such that the last state is a goal state for the task (meaning also that a linearisation has been "completed"), it gives a reward of 0 to the agent and terminates in state $u_g$, while for all other transitions the agent gets a reward of $-1$. Note that if $u\mathcal{S}$ is not the prefix of any linearisation of a POP in $\overline{\Pi}$, or $\mathcal{S}$ is not a goal state, then $\delta_u(u, P) = u$. Figure 1 shows the MPRM synthesised from the set $\overline{\Pi}_{\texttt{bridge}}$ of all partial-order plans for the bridge example we gave in Section 2.4.

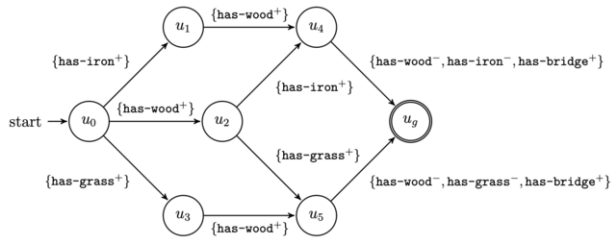


**Figure 1.** MPRM for the bridge task. Positive and negative postconditions are respectively denoted with a superscript $+$ and $-$.

In the remainder of this section, we provide a theoretical analysis linking the optimal policies that can be learned by an agent depending on the kind of reward machine it is equipped with. We consider RMs that can be built from the set of all partial-order plans (RM-$\overline{\Pi}$), a single partial-order plan (RM-$\overline{\pi}$), and a single sequential plan (RM-$\pi$), over the same planning domain $\mathcal{D}$. All RMs issue a non-negative reward only in the final state. We denote the optimal policy learnt using the set of all partial-order plans by $\rho^*_{\text{RM-}\overline{\Pi}}$, using a single partial-order plan by $\rho^*_{\text{RM-}\overline{\pi}}$, and using a single sequential plan by $\rho^*_{\text{RM-}\pi}$.

**Theorem 3.1.** *Let $\mathcal{M}$ be a labelled MDP, $\mathcal{D}$ a planning domain over $\mathcal{M}$, and RM-$\overline{\Pi}$, RM-$\overline{\pi}$ and RM-$\pi$ final state reward machines generated from $\mathcal{D}$ for the same task. Then,*

$$\rho^*_{\text{RM-}\overline{\Pi}} \geq \rho^*_{\text{RM-}\overline{\pi}} \geq \rho^*_{\text{RM-}\pi}$$

*where $\rho_1 \geq \rho_2$ if and only if $v(\rho_1(s)) \geq v(\rho_2(s))$ for all states $s \in S$ of $\mathcal{M}$.*

*Proof.* The proof follows from the fact that any policy that can be learned using an RM synthesised from a single sequential plan can also be learned using an RM synthesised from a partial-order plan which has the sequential plan as its linearisation (if it remains an optimal policy). Similarly, a policy learned using an RM synthesised from a single partial-order plan can also be learned using an RM that is synthesised from the set of all partial-order plans. □

Theorem 3.1 shows that MPRMs allow an agent to learn an optimal policy with respect to the planning domain and the planning task. A natural question to ask is whether it learns a *goal-optimal* policy $\rho^*$, i.e., a policy that achieves the goal using the smallest number of actions in the underlying MDP. For example, an agent using Q-learning is guaranteed to learn a goal-optimal policy on an MDP where the agent is always given the same negative reward and the discount factor $\gamma$ is exactly 1.

An agent using an MPRM will learn a goal-optimal policy if the planning domain and labelling are "*adequate*" for the goal.

**Definition 3.2.** Given a labelled MDP, planning domain $\mathcal{D}$, and goal $\mathcal{G}$, we say that $\mathcal{D}$ is *adequate* for $\mathcal{G}$ if, and only if:

- $\mathcal{G}$ corresponds to a set of planning domain fluents, i.e., $\mathcal{G} \subseteq \mathcal{F}$;
- a goal-optimal policy encounters all the state labels in some plan $\overline{\pi} \in \overline{\Pi}$ for $\mathcal{G}$, in the order consistent with the order in $\overline{\pi}$.

For example, if any policy to build a bridge has to encounter labels corresponding to getting wood, getting iron and using a factory, a planning domain and labelling containing only these fluents is adequate for the goal of having a bridge. However, if there is an alternative way of building a bridge that involves getting grass, and this label is missing in the planning domain, then the domain is not adequate for the goal of building a bridge.

**Theorem 3.3.** *$\rho^* = \rho^*_{RM\text{-}\overline{\Pi}}$ if RM-$\overline{\Pi}$ is synthesized from a goal-adequate planning domain.*

*Proof.* From the definition of a planning domain adequate for the goal, any goal-optimal policy has to go through the way-points encoded in the reward machine. □

## 4 Empirical Evaluation

In this section, we evaluate maximally permissive reward machines in three tasks in the CRAFTWORLD environment, and show that the agent obtains higher reward with an MPRM than with RMs based on a single partial-order plan or a single sequential plan. In the first task, the agent has to build a bridge, as in the example in Section 2.4 For the second task, the agent has to collect gold. In the third task, the agent has to collect gold or a gem, and the task is considered achieved when the agent collects at least one of the two items. For the gold-or-gem task we have to slightly modify the definition of goal states in planning tasks: the goal is the pair $\mathcal{G} = \langle \mathcal{G}^+ = \{\texttt{has-gold}, \texttt{has-gem}\}, \mathcal{G}^- = \emptyset \rangle$, and a planning state $\mathcal{S}$ is a *goal* state if and only if $\mathcal{G}^+ \cap \mathcal{S} \neq \emptyset$ and $\mathcal{G}^- \cap \mathcal{S} = \emptyset$. The gold and the gem are collected as described in [1]: gold is collected by using a(ny) bridge, whereas the gem is collected using an axe. To produce an axe, the agent must combine a stick, which can be obtained by processing wood at the workbench, with iron at the toolshed. We refer to these, respectively, as the "*bridge task*", "*gold task*", and "*gold-or-gem task*". In the planning domain, we add the following fluents:

- For the gold task: `has-gold`;
- For the gold-or-gem task: `has-gold`, `has-stick`, `has-axe`, `has-gem`;

and the following planning actions:

- For the gold task:
  - `get-gold`, with one positive precondition, `has-bridge`, and one positive postcondition, `has-gold`;
- For the gold-or-gem task:
  - `use-workbench`, with one positive precondition, `has-wood`, one positive postcondition, `has-stick`, and one negative postcondition, `has-wood`;
  - `use-toolshed-for-axe`, with positive preconditions, `has-stick` and `has-iron`, one positive postcondition, `has-axe`, and two negative postcondition, `has-stick` and `has-iron`;
  - `get-gem`, with one positive precondition, `has-axe`, and one positive postcondition, `has-gem`;

The set of partial-order plans for the bridge task $\overline{\Pi}_{\texttt{bridge}} = \{\overline{\pi}_{\texttt{iron-bridge}}, \overline{\pi}_{\texttt{rope-bridge}}\}$ is given in Section 2.4. For the gold task, we extend $\overline{\pi}_{\texttt{iron-bridge}}$ and $\overline{\pi}_{\texttt{rope-bridge}}$ by adding the `get-gold` action, and by having `use-factory` $\prec$ `get-gold` and `use-toolshed` $\prec$ `get-gold`. For the gold-or-gem task, the set of partial-order plans consists of $\overline{\pi}_{\texttt{iron-bridge}}$, $\overline{\pi}_{\texttt{rope-bridge}}$ and $\overline{\pi}_{\texttt{gem}}$ in which the agent makes an axe and uses it to mine the gem. $\overline{\pi}_{\texttt{gem}}$ is defined as follows:

$$
\begin{aligned}
&\overline{\pi}_{\texttt{gem}} = \\
&\langle \mathcal{A} = \{\texttt{get-wood}, \texttt{get-iron}, \texttt{use-workbench}, \\
&\qquad \texttt{use-toolshed-for-axe}, \texttt{get-gem}\}, \\
&\prec = \{\texttt{get-wood} \prec \texttt{use-workbench}, \\
&\qquad \texttt{get-iron} \prec \texttt{use-toolshed-for-axe}, \\
&\qquad \texttt{use-workbench} \prec \\
&\qquad\quad \texttt{use-toolshed-for-axe}, \\
&\qquad \texttt{use-toolshed-for-axe} \prec \texttt{get-gem}\}\rangle
\end{aligned}
$$

For each task, we also generate all sequential plans that can be obtained by linearising the POPs that can be used to achieve the task. Thus, for both the bridge and gold tasks, there are a total of 4 sequential plans and 2 partial-order plans. For the gold-or-gem task, there are a total of 7 sequential plans and 3 partial-order plans. In the Appendix, we provide formal definitions of the planning domains, and give also the plans for the gold and gold-or-gem tasks.

### 4.1 Experimental Setup

The maximally permissive RMs for each task were synthesised using the construction given in Section 3. The RMs for each partial-order and sequential plan were generated using the approach presented in [11]. Training is carried out by using Q-learning over the resulting MDPRMs [23].[3]

For each task we generated 10 different maps of size 41 by 41 cells. The maps and initial locations were chosen so that from some locations a task can be completed more quickly by following a particular sequential plan. For example, in the first map for the bridge task, if the agent starts from a location in the upper half of the map (i.e., in the first 20 rows) it is more convenient to build an iron bridge, while in the lower half of the map it is more convenient to build a rope bridge. The MDP reward function $r$ returns $-1$ for each step taken by the agent, until it achieves the task or the episode terminates. When the task is completed, the map and agent are "re-initialised": the agent is placed on a random starting cell and its "inventory" is emptied, i.e., it contains no items. For each set of plans and single partial-order/sequential plan for a task, and for each of the 10 maps for the task, an agent was trained with the corresponding RM for 10,000,000 training steps. Training was carried out in episodes, lasting at most 1,000 steps, after which the environment was re-initialised regardless of whether the agent has achieved the task or not. Every 10,000 training steps the agent was evaluated on the same map used for training from 5 (predetermined) starting positions. We set the learning rate $\alpha = 0.95$, the discount rate $\gamma = 1$, and the exploration rate $\varepsilon = 0.1$.

[3] Note that we do not provide results for a baseline that does not employ reward machines (e.g., Q-learning): as shown in [23], CRAFTWORLD is a complex environment with sparse rewards, making it infeasible for an agent to learn an effective policy without having access to a reward machine.

Our implementation, largely based off of that of [11], is available on GitHub at github.com/giovannivarr/MPRM-ECAI24.

### 4.2 Results

For each approach, we plot the median and the 25$^{th}$ and the 75$^{th}$ percentiles (shaded areas) of the rewards obtained across all maps by the agents in the evaluations during training for each task. To make the plots more readable, we have "aggregated" results for the sequential and partial-order plan-based RMs: for each kind of plan we plot the median and the 25$^{th}$ and the 75$^{th}$ percentiles of all agents trained with an RM generated using that type of plan. In the plots, "`QRM-MPRM`" denotes the performance of the agent trained with a maximally permissive RM, while "`Aggregated-QRM-Seq`" and "`Aggregated-QRM-POP`" are, respectively, the aggregated performance of agents trained with sequential plan RMs and partial-order plan RMs. On the $x$-axis we plot the number of steps (in millions), while on the $y$-axis we plot the performance obtained during the evaluations run at the corresponding timestep.

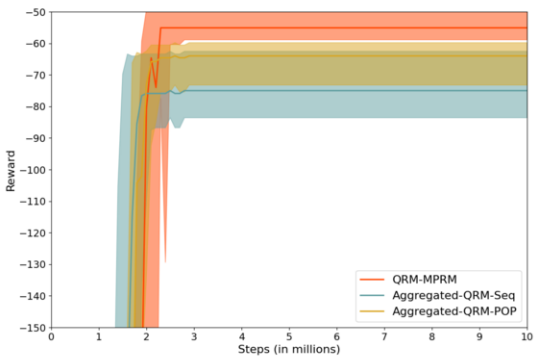


**Figure 2.** Results for the bridge task.

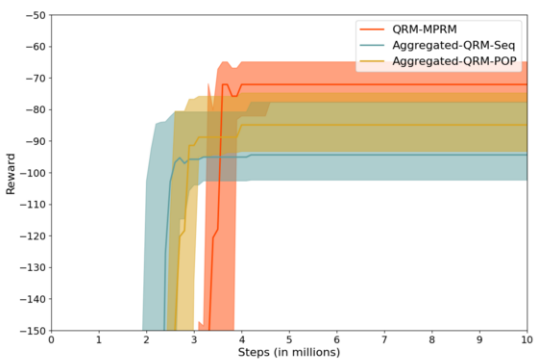


**Figure 3.** Results for the gold task.

Figure 2 shows the results for the bridge task, Figure 3 shows the results for the gold task, and Figure 4 shows the results for the gold-or-gem task. As can be seen, in all tasks the agent trained with the MPRM outperforms the aggregated results for the agents trained using RMs based on a single partial-order or sequential plan. This is as expected given Theorem 3.1. In addition, the agent trained using an RM based on a single partial-order plan outperforms the agent trained using an RM based on a single sequential plan. Again, this

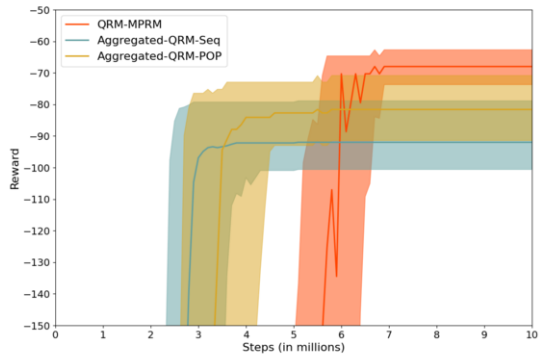


**Figure 4.** Results for the gold-or-gem task.

is consistent with Theorem 3.1 and the results in [11]. However, in all experiments, the agent trained with the maximally permissive RM converges more slowly than the other agents (particularly in the gold-or-gem task). Intuitively, increasing the flexibility of the RM trades solution quality for sample complexity. Note that, as the planning domain used is *adequate* in the sense defined in Section 3 for all the tasks, the MPRM agent can learn an optimal policy for each task.

Figure 5 illustrates the behaviour of the agent trained using the MPRM and the agents trained using the RMs generated from each of the partial-order plans on a map for the gold-or-gem task. For readability, we have decided to not include agents trained with a sequential plan-based RM: note that none of them achieved the task in fewer steps than the agents shown in the figure. In the supplementary material we provide, for each agent, a file containing its trajectory (i.e., the ordered sequence of coordinates of the cells it visited) in the test, also for agents trained with a sequential plan-based RM. Given the initial position of the agent, the optimal plan for the task is to collect a gem (POP-2). As can be seen, both the agent trained using the POP-2-based RM and the MPRM agent achieve the goal by collecting a gem and complete the task in 60 steps. The agent trained using the RM based on the POP to collect gold using a rope bridge (POP-1) is also able to achieve the task, but in 63 steps. However, the agent trained using the RM based on the POP to collect gold using an iron bridge (POP-0) is unable to complete the task after 10,000,000 timesteps. This illustrates that the MPRM agent is able to choose the "correct" plan for the agent's position, and the inherent problems of training agents using an RM based on a single plan.

## 5 Related Work

Reward machines have been used both in single-agent RL [21, 6, 4, 7] and multi-agent RL [17, 10]. As mentioned in the Introduction, approaches to synthesise reward machines from high-level specifications have also been proposed; however, to the best of our knowledge, only [11, 12] and our work generate reward machines from plans.

Another line of research focuses on learning reward machines from experience. In [22] an approach is proposed that uses Tabu search to update the RM hypothesis, by searching through the trace data generated by the agent exploring the environment. [28] presented an approach where the RM hypothesis is updated whenever traces that are inconsistent with the current one are detected. In [9] RM-learning is reduced to SAT solving, and solved using DPLL. While these approaches do not require an abstract model of the environment in the form of a planning domain, they focus on learning a reward machine for a single task. In [22] and [28] the agent learns a

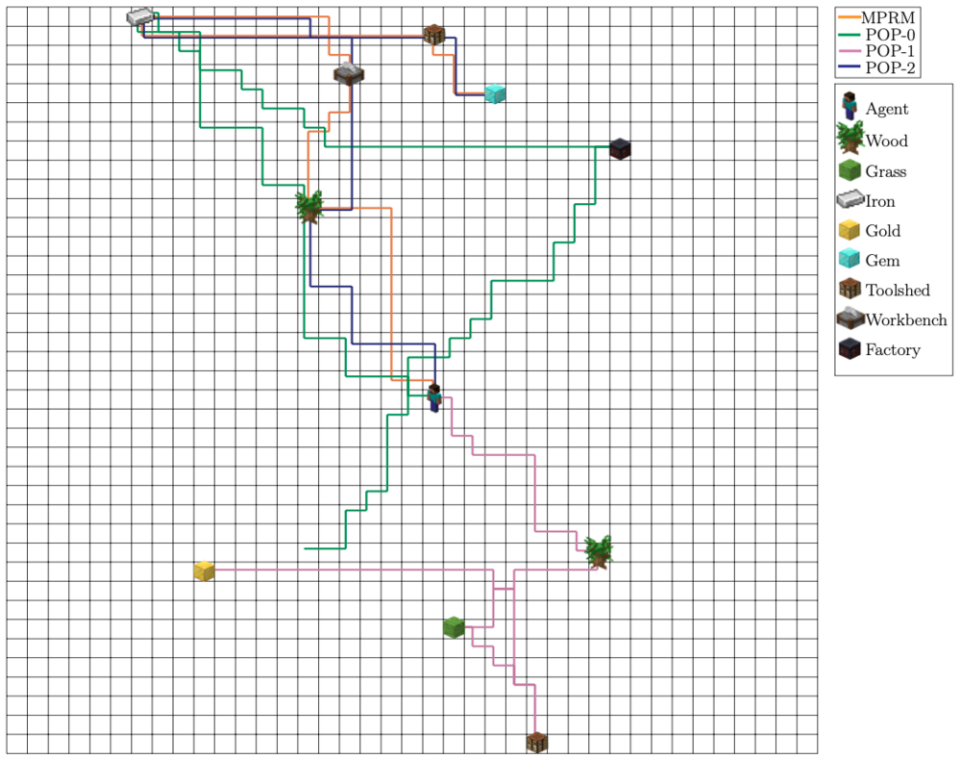


**Figure 5.** Illustration of the behaviour of the MPRM and POP-trained agents on the gold-or-gem task.

policy for each state of the RM hypothesis. However, when the latter is updated, it has to re-learn such policies from scratch ([28] tries to mitigate this issue by transferring a subset of the policies, but this is not always possible). Moreover, all these approaches assume that the agent is able to generate "positive" traces, i.e., traces in which the task is achieved. While in simple environments this is a reasonable assumption, for more complex environments with sparse rewards it may be difficult to generate positive traces.

Planning has been applied to reinforcement learning since at least [8], which combined Q-learning with STRIPS planning. More recently, [29] proposed an approach integrating planning with options [20, 5]. In [15] a framework is introduced that exploits planning to improve sample efficiency in deep RL. In both of these approaches the RL experience is then used to improve the planning domain, similarly to what happens in model-based RL. Then, the new plan obtained using the updated domain is used to train again the RL agent. In [19, 13, 27] abstract models for the low-level MDP and/or its actions are learned so that planning can be leveraged to improve learning. However, all of these approaches assume that learning is guided by a single (sequential) plan.

## 6 Conclusions

We have proposed a new planning-based approach to synthesising maximally permissive reward machines which uses the set of partial-order plans for a goal rather than a single sequential or partial-order plan as in previous work. Planning-based approaches have the advantage that it is straightforward to train agents to achieve new tasks — given a planning domain, we can automatically generate a reward machine for a new task. We have provided theoretical results showing how agents trained using maximally permissive reward machines learn policies that are at least as good as those learned by agents trained with a reward machine built from an individual sequential or partial-order plan, and the expected reward of an optimal policy learned using an MPRM synthesised from a *goal-adequate* planning domain is the same as that of an optimal policy for the underlying MDP. Experimental results from three different tasks in the CRAFTWORLD environment suggest that these theoretical results apply in practice. However, our results also show that agents trained with maximally permissive RMs converge more slowly than agents trained using RMs based on a single plan. We believe this is because the increased flexibility of maximally permissive RMs trades solution quality for sample complexity. Our approach is therefore most useful when the quality of the resulting policy is paramount.

A limitation of our approach is that, in the worst case, the set of all partial order plans for a task may be exponential in the number of actions in the planning domain. In future work we would like to investigate the use of *top-k* planning techniques, e.g., [14], to sample a diverse subset of the set of all plans. Intuitively, such an approach could allow the quality of the resulting policy to be traded off against the number of plans in the sample.

Another line of future work is to investigate option-based approaches to learning [20, 5] as in [12], where each abstract action in a plan is "implemented" as an option. We expect results similar to the ones in this paper, where the agent trained with all partial-order plans is able to achieve a better policy but converging slower.

Finally, the experiments in Section 4 are limited to discrete environments. However, our approach is applicable to environments with continuous action and state spaces. Reward machines have previously been successfully applied in such environments [23, 7], and planning domains, which form the basis our approach, are agnostic about the underlying environment, as they are defined in terms of states resulting from (sequences of) MDP actions rather than the actions themselves. Nevertheless, learning in continuous environments is more challenging than learning in discrete ones, and evaluating the benefits of our approach in such environments is future work.

## References


[1] J. Andreas, D. Klein, and S. Levine. Modular multitask reinforcement learning with policy sketches. In *Proceedings of the 30th International Conference on Machine Learning (ICML 2017)*, pages 166–175, 2017.

[2] A. Camacho, R. Toro Icarte, T. Klassen, R. Valenzano, and S. McIlraith. LTL and beyond: Formal languages for reward function specification in reinforcement learning. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, (IJCAI 2019)*, pages 6065–6073, 2019.

[3] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32: 333–377, 1987.

[4] J. Corazza, I. Gavran, and D. Neider. Reinforcement learning with stochastic reward machines. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI 2022))*, pages 6429–6436, 2022.

[5] T. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, 13: 227–303, 2000.

[6] T. Dohmen, N. Topper, G. Atia, A. Beckus, A. Trivedi, and A. Velasquez. Inferring probabilistic reward machines from non-markovian reward signals for reinforcement learning. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS 2022)*, pages 574–582, 2022.

[7] D. Furelos-Blanco, M. Law, A. Jonsson, K. Broda, and A. Russo. Hierarchies of reward machines. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*, pages 10494–10541, 2023.

[8] M. Grounds and D. Kudenko. Combining reinforcement learning with symbolic planning. In *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*, pages 75–86, 2005.

[9] M. Hasanbeig, N. Y. Jeppu, A. Abate, T. Melham, and D. Kroening. Deepsynth: Automata synthesis for automatic task segmentation in deep reinforcement learning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)*, pages 7647–7656, 2021.

[10] J. Hu, Z. Xu, W. Wang, G. Qu, Y. Pang, and Y. Liu. Decentralized graph-based multi-agent reinforcement learning using reward machines. *Neurocomputing*, 564:126974, 2024.

[11] L. Illanes, X. Yan, R. Toro Icarte, and S. McIlraith. Symbolic planning and model-free reinforcement learning: Training taskable agents. In *Proceedings of 4th Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM)*, pages 191–195, 2019.

[12] L. Illanes, X. Yan, R. Toro Icarte, and S. McIlraith. Symbolic plans as high-level instructions for reinforcement learning. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS 2020)*, pages 540–550, 2020.

[13] M. Jin, Z. Ma, K. Jin, H. H. Zhuo, C. Chen, and C. Yu. Creativity of ai: Automatic symbolic option discovery for facilitating deep reinforcement learning. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI 2022)*, pages 7042–7050, 2022.

[14] M. Katz and J. Lee. $K_*$ search over orbit space for top-k planning. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence (IJCAI 2023)*, pages 5368–5376, 2023.

[15] D. Lyu, F. Yang, B. Liu, and S. Gustafson. Sdrl: Interpretable and data-efficient deep reinforcement learning leveraging symbolic planning. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, pages 2970–2977, 2019.

[16] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI 1991)*, pages 634–639, 1991.

[17] C. Neary, Z. Xu, B. Wu, and U. Topcu. Reward machines for cooperative multi-agent reinforcement learning. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2021)*, page 934–942, 2021.

[18] X. Nguyen and S. Kambhampati. Reviving partial order planning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 459–466, 2001.

[19] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588:604–609, 2020.

[20] R. S. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112:181–211, 1999.

[21] R. Toro Icarte, T. Klassen, R. Valenzano, and S. McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, pages 2107–2116, 2018.

[22] R. Toro Icarte, E. Waldie, T. Klassen, R. Valenzano, M. Castro, and S. McIlraith. Learning reward machines for partially observable reinforcement learning. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, 2019.

[23] R. Toro Icarte, T. Klassen, R. Valenzano, and S. McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 73:173–208, 2022.

[24] G. Varricchione, N. Alechina, M. Dastani, and B. Logan. Synthesising reward machines for cooperative multi-agent reinforcement learning. In *Proceedings of the 20th European Conference on Multi-Agent Systems (EUMAS 2023)*, pages 328–344, 2023.

[25] G. Varricchione, N. Alechina, M. Dastani, and B. Logan. Maximally permissive reward machines, 2024. URL https://arxiv.org/abs/2408.08059. arXiv preprint arXiv: 2408.08059, 2024. Full version of this paper.

[26] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[27] Z. Wu, C. Yu, C. Chen, J. Hao, and H. H. Zhuo. Plan to predict: Learning an uncertainty-foreseeing model for model-based reinforcement learning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS 2022)*, pages 15849–15861, 2022.

[28] Z. Xu, I. Gavran, Y. Ahmad, R. Majumdar, D. Neider, U. Topcu, and B. Wu. Joint inference of reward machines and policies for reinforcement learning. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS 2020)*, pages 590–598, 2020.

[29] F. Yang, D. Lyu, B. Liu, and S. Gustafson. PEORL: Integrating symbolic planning and hierarchical reinforcement learning for robust decision-making. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, pages 4860–4866, 2018.