LTL$_f$/LDL$_f$ Non-Markovian Rewards

Ronen I. Brafman
Ben-Gurion University, Beer-Sheva, Israel
brafman@cs.bgu.ac.il

Giuseppe De Giacomo & Fabio Patrizi
Sapienza Università di Roma, Italy
degiacomo,patrizi}@dis.uniroma1.it

Abstract
In Markov Decision Processes (MDPs), the reward obtained in a state is Markovian, i.e., depends on the last state and action. This dependency makes it difficult to reward more interesting long-term behaviors, such as always closing a door after it has been opened, or providing coffee only following a request. Extending MDPs to handle non-Markovian reward functions was the subject of two previous lines of work. Both use LTL variants to specify the reward function and then compile the new model back into a Markovian model. Building on recent progress in temporal logics over finite traces, we adopt LTL$_f$/LDL$_f$ for specifying non-Markovian rewards and provide an elegant automata construction for building a Markovian model, which extends that of previous work and offers strong minimality and compositionality guarantees.

Introduction
Markov Decision Processes (MDPs) are a central model for sequential decision making under uncertainty. They are used to model and solve many real-world problems, and to address the problem of learning to behave well in unknown environments. The Markov assumption is a key element of this model. It states that the effects of an action depend only on the state it was executed in, and that a reward given at a state depends only on the previous action and state. It has long been observed (Bacchus, Boutilier, and Grove 1996; Thiébaux et al. 2006) that many performance criteria call for more sophisticated reward functions that do not depend on the last state only. For example, we may want to reward a robot that eventually delivers coffee each time it gets a request; or, to ensure it will access restricted areas only after having acquired the right permission. Such rewards are non-Markovian. Interestingly, Littman has advocated at IJ-CAI 2015 that it may actually be more convenient, from a design perspective, to assign rewards to the satisfaction of (non-Markovian) declarative temporal properties, rather than to states (Littman et al. 2017).

To extend MDPs with non-Markovian rewards we need a language for specifying such rewards. Markovian rewards are specified as a function $R$ from the previous state and action to the reals. $R$ can be specified using an explicit reward matrix, or implicitly, by associating a reward with properties of the last state and action. With non-Markovian rewards, an explicit representation is no longer possible, as the number of possible histories or futures of a state is infinite. Hence, we must use an implicit specification that can express properties of past (or future) states. To date, two specification languages have been proposed. (Bacchus, Boutilier, and Grove 1996) suggest using a temporal logic of the past. Whether a state satisfies such a past temporal formula depends on the entire sequence of states leading to the state. Thus, we can reward appropriate response to a “bring-coffee” command by associating a reward with the property: the “bring-coffee” command was issued in the past, and now I have coffee. A second proposal, by (Thiébaux et al. 2006), uses a temporal logic of the future with a special symbol to denote awarding a reward. At each step, one checks whether this symbol must be true in the current state, for the reward formula to be satisfied in the initial state. If that is the case, the current state is rewarded. This language is not often used in other areas, and its semantics is more involved.

Existing MDP solution methods, possibly with the exception of Monte-Carlo tree search algorithms (Kocsis and Szepesvári 2006), rely heavily on the Markov assumption, and cannot be applied directly with non-Markovian rewards. To address this, both proposals above transform the non-Markovian model to an equivalent Markovian one that can be solved using existing algorithms, by enriching the state with information about the past. For example, if we extend our state to record whether a “bring-coffee” command was issued earlier, a reward for bringing coffee in states indicating that “bring-coffee” was issued in the past, is now Markovian. It rewards the same behaviors as the non-Markovian reward on the original state. We call a model obtained by extending the state space of the original non-Markovian MDP, an extended MDP.

Using extended MDPs is a well-known idea. Since state space size affects the practical and theoretical complexity of most MDP solution algorithms, the main question is how to minimally enrich the state so as to make rewards Markovian. (Bacchus, Boutilier, and Grove 1996) provide algorithms for constructing an extended MDP that attempt to minimize size by reducing the amount of information about the past that is maintained. While their construction does not generate a minimal extended MDP, they allude to using automata minimization techniques to accomplish this. (Thiébaux et
al. 2006), instead, use a construction that works well with forward search based algorithms, such as LAO* (Hansen and Zilberstein 2001) and LRTDP (Bonet and Geffner 2003). Unlike classical dynamic programming methods that require the entire state space a-priori, these algorithms generate reachable states only. With a good heuristic function, they often generate only a fraction of the state space. So, while the augmented search space they obtain is not minimal, because states are constructed on the fly during forward search, their approach does not require a-priori enumeration of the state space, and never generates an unreachable state.

The aim of this paper is to bring to bear developments in the theory of temporal logic over finite traces to the problem of specifying and solving MDPs with non-Markovian rewards. With these tools, which were unavailable to earlier work, we are able to provide a cleaner, more elegant approach that builds on well understood semantics, much more expressive languages, and enjoys good algorithmic properties. We adopt LDL₁, a temporal logic of the future interpreted over finite traces, which extends LTL₁, the classical linear-time temporal logic over finite traces (De Giacomo and Vardi 2013). LDL₁ has the same computational features of LTL₁ but is more expressive, as it captures monadic second-order logic (MSO) on finite traces (i.e., inductively defined properties), instead of first-order logic (FO), as LTL₁.

A number of techniques based on automata manipulation have been developed for LDL₁, to address tasks such as satisfiability, model checking, reactive synthesis, and planning under full/partial observability (De Giacomo and Vardi 2013; 2015; De Giacomo and Vardi 2016; Torres and Baier 2015; Camacho et al. 2017c). We exploit such techniques to generate an extended MDP with good properties.

Our formalism has three important advantages: 1. Enhanced expressive power. We move from linear-time temporal logics used by past authors to LDL₁, paying no additional (worst-case) complexity costs. LDL₁ can encode in polynomial time LTL₁, regular expressions (RE), the past LTL (PLTL) of (Bacchus, Boutilier, and Grove 1996), and all examples of (Thiébaux et al. 2006). Moreover, LDL₁ can naturally represent "procedural constraints" (Baier et al. 2008), i.e., sequencing constraints expressed as programs, using "if" and "while". In fact, future logics are more commonly used in the model checking community, as they are considered more natural for expressing desirable properties. This is especially true with complex properties that require the power of LDL₁.

2. Minimality and Compositionality. We generate a minimal equivalent extended MDP, exploiting existing techniques for constructing automata that track the satisfiability of an LDL₁ formula. This construction is relatively simple and compositional: if a new reward formula is added, we only need to optimize its automaton and add it to the current (extended) MDP. If the current MDP was minimal, the resulting (extended) MDP is minimal too.

3. Forward Construction via Progression. The automaton used to identify when a reward should be given can be constructed in a forward manner using progression. This ensures the generation of reachable states only, as in (Thiébaux et al. 2006). Moreover, we can combine progression in the space of MDPs with precomputed automata minimization to obtain the best of both worlds.

**Background**

MDPs. A Markov Decision Process (MDP) \( M = (S, A, Tr, R) \) contains a set \( S \) of states, a set \( A \) of actions, and a transition function \( Tr : S \times A \rightarrow \text{Prob}(S) \) that returns for every state \( s \) and action \( a \) a distribution over the next state. We can further restrict actions to be defined on a subset of \( S \) only, and use \( A(s) \) to denote the actions applicable in \( s \). The reward function, \( R : S \times A \rightarrow \mathbb{R} \), specifies the real-valued reward received by the agent when applying action \( a \) in state \( s \). In this paper, states in \( S \) are truth assignments to a set \( P \) of primitive propositions. Hence, if \( \phi \) is a propositional formula and \( s \) a state, we can check whether \( s \models \phi \).

A solution to an MDP, called a policy, assigns an action to each state, possibly conditioned on past states and actions. The value of policy \( \rho \) at \( s, \varphi \) is the expected sum of (possibly discounted) rewards when starting at \( s \) and selecting actions based on \( \rho \). Every MDP has an optimal policy, \( \rho^* \), i.e., one that maximizes the expected sum of rewards for every starting state \( s \in S \). When the horizon is infinite, there exists an optimal policy \( \rho^* : S \rightarrow A \) that is stationary and deterministic (i.e., \( \rho^* \) depends only on the current state) (Puterman 2005). There are diverse methods for computing an optimal policy. With the exception of online simulation-based methods, they rely on the Markov assumption, and their theoretical and practical complexity is strongly impacted by \(|S|\).

**LTL₁ and LDL₁.** LTL₁ is essentially the standard Linear-time Temporal Logic LTL (Pnueli 1977) interpreted over finite, instead of infinite, traces (De Giacomo and Vardi 2013). LTL₁ is as expressive as FO over finite traces and star-free regular expressions (RE), thus strictly less expressive than RE, which is true in turn are as expressive as MSO over finite traces. RE themselves are not convenient for expressing temporal specifications, since, e.g., they miss direct constructs for negation and conjunction. For this reason, (De Giacomo and Vardi 2013) introduced LDL₁ (linear dynamic logic on finite traces), which merges LTL₁ with RE, through the syntax of the well-known logic of programs PDL, propositional dynamic logic (Fischer and Ladner 1979; Harel, Kozen, and Tiuryn 2000; Vardi 2011), but interpreted over finite traces.

We consider a variant of LDL₁ that works also on empty traces. Formally, LDL₁ formulas \( \varphi \) are built as follows:

\[
\varphi ::= tt | \neg \varphi | \varphi_1 \land \varphi_2 | (\langle g \rangle \varphi)
\]

\[
g ::= \delta | \sigma | \varphi_1 \land \varphi_2 | (\langle g \rangle \varphi)
\]

where \( tt \) stands for logical true; \( \delta \) is a propositional formula over \( P \) (including true, not to be confused with \( tt \)); \( \sigma \) denotes path expressions, which are RE over propositional formulas \( \delta \) with the addition of the test construct \( \varphi \) typical of PDL. We use abbreviations \( [\varphi] \delta \varphi \equiv \neg \langle \delta \rangle \neg \varphi \) as in PDL, \( \langle \delta \rangle \equiv \neg tt \) for false, and \( \delta \langle \varphi \rangle tt \) to denote occurrence of propositional formula \( \varphi \).

Intuitively, \( \langle g \rangle \varphi \) states that, from the current step in the trace, there exists an execution satisfying the RE \( g \) such that its last step satisfies \( \varphi \), while \( [\varphi] g \) states that, from the current
step, all executions satisfying the RE ϕ are such that their last step satisfies ϕ. Tests are used to insert into the execution path checks for satisfaction of additional LDLf formulas.

The semantics of LDLf is given in terms of finite traces, i.e., finite sequences π = π₀, . . . , πₙ of elements from the alphabet 2². We define π(i) = πᵢ, length(π) = n + 1, and π(i, j) = πᵢ.πᵢ₊₁. . . . , πⱼ₋₁. When j > length(π), π(i, j) = π(i, length(π)).

In decision processes, traces are usually sequences of states and actions, i.e., they have the form: (s₀, a₁, s₁, . . . , sₙ₋₁, aₙ). These can still be represented as traces of the form π = π₀, . . . , πₙ by extending the set P to include one proposition pᵢ per action aᵢ and setting πᵢ = sᵢ ∪ {pᵢ | a = aᵢ+1}. In this way, πᵢ denotes the pair (sᵢ, aᵢ+1). We will always assume this form, even if referring to sequences of states and actions. Given a finite trace π, an LDLf formula ϕ, and a position i, we define when ϕ is true at step i, written π, i = ϕ, by (mutual) induction, as follows:

- π, i = tt;
- π, i = ¬ϕ iff π, i = ϕ;
- π, i = ϕ₁ ∧ ϕ₂ iff π, i = ϕ₁ and π, i = ϕ₂;
- π, i = (⟨ϕ⟩)ϕ iff there exists i ≤ j such that π(i, j) ∈ L(ϕ) and π, j = ϕ, where the relation π(i, j) ∈ L(ϕ) is as follows:
  - π(i, j) ∈ L(ϕ) if j = i+1, i < length(π), and π(i) = φ (ϕ propositional);
  - π(i, j) ∈ L(ϕ) if j = i and π, i = ϕ;
  - π(i, j) ∈ L(ϕ₀ + ϕ₂) if π(i, j) ∈ L(ϕ₀) or π(i, j) ∈ L(ϕ₂);
  - π(i, j) ∈ L(ϕ₁ + ϕ₂) if there exists k ∈ [i, j] such that π(i, k) ∈ L(ϕ₁), and π(k, j) ∈ L(ϕ₂);
  - π(i, j) ∈ L(ϕ*) if j = i or there exists k such that π(i, k) ∈ L(ϕ) and p(k, j) ∈ L(ϕ*)

Note that if i ≥ length(π), the above definitions still apply; though, ⟨ϕ⟩ϕ (ϕ propos.) and ⟨ϕ⟩ϕ become trivially false.

We say that a trace π satisfies an LDLf formula ϕ, written π = ϕ, if π, 0 = ϕ. Also, sometimes we denote by L(ϕ) the set of traces that satisfy ϕ: L(ϕ) = {π | π = ϕ}.

LDLf is as expressive as MSO over finite words. It captures LTf, by seeing next and until as the abbreviations φ + (true) (ϕ ∧ ¬end) and ϕ₁ U ϕ₂ + (true) (ϕ₁ ∧ ¬end) and any RE r, with the formula (r) end, where end = (true) eff expresses the trace has ended. Note that in addition to end we can also denote the last element of the trace as last = (true) end. Section Non-Markovian Rewards illustrates several examples of LDLf formulas in our context.

**Computing DFA for LDLf formulas**

As standard, an NFA is a tuple A = (Σ, Q₀, δ, F), where:
(i) Σ is the input alphabet; (ii) Q is the finite set of states; (iii) Q₀ ∈ Q is the initial state; (iv) δ ∈ Q × Σ × Q is the transition relation; (v) F ⊆ Q is the set of final states. A DFA is an NFA where δ is a function δ : Q × Σ → Q. By L(Δ) we mean the set of all traces over Σ accepted by Δ.

We can associate each LDLf formula ϕ with an (exponentially large) NFA Aϕ = (2², Q₀, δ, F) that accepts exactly the traces satisfying ϕ. A direct algorithm (LDLf2NFA) for computing the NFA given the LDLf formula, which is a variant of that in (De Giacomo and Vardi 2015), is reported below. Its correctness relies on the fact that (i) every LDLf formula ϕ can be associated with a polynomial alternating automaton on words (AFW) accepting exactly the traces that satisfy ϕ (De Giacomo and Vardi 2013), and (ii) every AFW can be transformed into an NFA, see, e.g., (De Giacomo and Vardi 2013).

The algorithm assumes the LDLf formula is in negation normal form (NNF), i.e., with negation symbols occurring only in front of propositions (any LDLf formula can be rewritten in NNF in linear time). Let Θ be the following auxiliary function, which takes as input an (implicitly quoted) LDLf formula ϕ (in NNF), extended with auxiliary constants F and T, and a propositional interpretation Θ for P, and returns a positive boolean formula whose atoms are (implicitly quoted) ϕ subformulas (not including F or T):

\[
\begin{align*}
\delta((t, θ)) &= true \\
\delta((f, θ)) &= false \\
\delta(ϕ, θ) &= δ((ϕ)t, θ) (ϕ prop.) \\
\δ(ϕ₁ ∧ ϕ₂, θ) &= δ(ϕ₁, θ) ∨ δ(ϕ₂, θ) \\
\δ(ϕ₁ ∨ ϕ₂, θ) &= δ(ϕ₁, θ) ∨ δ(ϕ₂, θ) \\
\δ(⟨ϕ⟩, ϕ, θ) &= E(ϕ) if θ = ϕ (ϕ prop.) \\
&= false if θ ≠ ϕ \\
\δ((ϕ₁ ∧ ϕ₂), ϕ, θ) &= δ(ϕ₁, θ) ∧ δ(ϕ₂, θ) \\
\δ(⟨ϕ₁ + ϕ₂⟩, ϕ, θ) &= δ(ϕ₁, θ) ∨ δ(ϕ₂, θ) \\
\δ(⟨ϕ₁⟩, ϕ, θ) &= δ(ϕ₁, θ) ∨ δ(ϕ₁F₁(ϕ₁), θ) \\
\δ(⟨ϕ⟩), ϕ, θ) &= δ(ϕ₁, θ) ∨ δ(ϕ₁F₁(ϕ₁), θ) \\
\δ(⟨ϕ₁⟩, ϕ, θ) &= δ(ϕ₁, θ) ∨ δ(ϕ₁F₁(ϕ₁), θ) \\
\δ(⟨ϕ₁⟩, ϕ, θ) &= δ(ϕ₁, θ) ∨ δ(ϕ₁F₁(ϕ₁), θ) \\
\delta(ϕ₁, θ) &= false \\
\δ(ϕ), θ) &= true (ϕ prop.)
\end{align*}
\]

where E(ϕ) recursively replaces ϕ all occurrences of atoms of the form T or F by E(ψ); and δ(ϕ, θ) is defined inductively exactly as above, except for the following base cases:

\[
\delta((ϕ₁ ∧ ϕ₂), θ) &= false \\
\delta((ϕ₁), θ) &= true (ϕ prop.)
\]

Note that δ(ϕ, θ) is always either true or false.

The NFA Aϕ for an LDLf formula ϕ is then built in a forward fashion as shown in Figure 1, where: states of Aϕ are sets of atoms (each atom is a quoted ϕ subformula) to be interpreted as conjunctions; the empty conjunction ∅ stands for true; q' is a set of quoted subformulas of ϕ denoting a minimal interpretation such that q' = L(ϕ); notice that we trivially have (ϕ, a, ω) ∈ δ for every a ∈ 2².

**Theorem 1.** (De Giacomo and Vardi 2015) Algorithm LDLf2NFA is correct, i.e., for every finite trace π: π = ϕ iff π ∈ L(A). Moreover, it terminates in at most an exponential number of steps, and generates a set of states S whose size is at most exponential in the size of the formula ϕ.

The NFA Aϕ can be transformed into a DFA, in exponential time, following the standard procedure, and then possibly put in (the unique) minimal form, in polynomial time (Rabin
and Scott 1959). Thus, we can transform any LDLₐ formula into a DFA of double exponential size. While this is a worst-case complexity, in most cases the size of the DFA is actually manageable (Tabakov and Vardi 2005).

**Computing the DFA on-the-fly.** All operations above can be performed on-the-fly, without the need for constructing $A_\varphi$. To do so, we progress all possible states that the NFA can be in, after consuming the next trace symbol, and accept the trace iff, once it has been completely read, the set of possible states contains a final state. More formally, call a set of possible states for the NFA a macro state, let $Q = \{q_1, \ldots, q_n\}$ be the current macro state (initially $Q = Q_0 = \{q_0\} = \{\varphi\}$), and let $\Theta$ be the next trace symbol. Then, the successor macro state is the set $Q' = \{q' \mid \exists q \in Q \text{ s.t. } q' = \lambda_{(\varphi, q)} \partial(\psi, \Theta)\}$. Given an input trace $\pi$, we decide whether $\pi \models \varphi$ by iterating the above procedure, starting from the initial state $Q = Q_0$, and accepting $\pi$ iff the last state obtained includes $\{true\}$.

The following observations are in order. Firstly, to compute $Q'$, only function $\partial$ is needed. Neither the set of states $Q$ nor the transition relation $\delta$ of the NFA are required. In other words, the on-the-fly progression does not require the construction of $A_\varphi$. Secondly, the progression step produces only one successor macro state $Q'$, thus transitions are deterministic. Indeed, it is immediate to see that the on-the-fly progression includes the determinization of $A_\varphi$. As it turns out, the on-the-fly approach performs, in fact, the execution and the determinization at once and in a lazy way, i.e., avoiding the construction of the entire resulting DFA.

### Non-Markovian Rewards

In this section we extend MDPs with LDLₐ-based reward functions resulting in a non-Markovian-reward decision process (NMMDP). Specifically, a NMMDP is a tuple $M = (S, A, Tr, R)$, where $S, A$ and $Tr$ are as in an MDP, and $R$ is redefined as $R : (S \times A)^* \rightarrow \mathbb{R}$. The reward is now a real-valued function over finite state-action sequences. Given a (possibly infinite) trace $\pi$, the value of $\pi$ is:

$$v(\pi) = \sum_{i=1}^{\infty} \gamma^{i-1} R(\pi(1), \pi(2), \ldots, \pi(i)),$$

where $0 < \gamma \leq 1$ is the discount factor and $\pi(i)$ denotes the pair $(s_{i-1}, a_i)$. Since every policy $\rho : S^* \rightarrow A$ induces a distribution over the set of possible infinite traces, we can now define the value of a policy $\rho$ given an initial state $s_0$ as:

$$v^\rho(s) = E_{\pi \sim \mathcal{M}, \rho, s_0} v(\pi).$$

That is, $v^\rho(s)$ is the expected value of infinite traces, where the distribution over traces is defined by the initial state $s_0$, the transition function $Tr$, and the policy $\rho$.

Specifying a non-Markovian reward function explicitly is cumbersome and unintuitive, even if we only want to reward a finite number of traces. But, typically, we want to reward behaviors that correspond to various patterns. LDLₐ provides an intuitive and convenient language for specifying $R$ implicitly, using a set of pairs $\{(\varphi_1, r_1), \ldots, (\varphi_n, r_n)\}$. Intuitively, if the current (partial) trace is $\pi = (s_0, a_1, \ldots, s_{n-1}, a_n)$, the agent receives at $s_n$ a reward $r_i$ for every formula $\varphi_i$ satisfied by $\pi$. Formally:

$$R(\pi) = \sum_{1 \leq i \leq n, \varphi_i(\pi)} r_i$$

From now on, we assume $R$ is thus specified. Note that we use LDLₐ to reward partial traces. Sometimes, we may want to reward complete traces only, this is studied later.

**Examples.** To illustrate the power of LDLₐ as a mechanism to specify non-Markovian rewards, we show some examples.

The properties mentioned in the introduction are LDLₐ-expressible: [true]\{request$p$ \rightarrow [true]\coffee$p$\} (all coffee requests from person $p$ will eventually be served); [\neg restr$\alpha$ \rightarrow restr$\alpha$ \rightarrow restr$\alpha$ \rightarrow restr$\alpha$\} (before entering restricted area $a$ the robot must have permission for $a$). Note that the first formula can be easily rewritten in LTLₐ, as $\Box(request_p \rightarrow \diamond coffee_p)$, but not the second one.

Also the formulas from (Bacchus, Boutilier, and Grove 1996) and (Thiébaux et al. 2006) are LDLₐ-expressible: 1. $\neg g \neg g$ end, reward offered only at the first state where $g$ holds; 2. $true \neg g true$ end, reward offered at every state that follows $g$ (included); 3. $\neg g \neg g \neg g \neg g \neg g$ end, achievement of $g$ is rewarded periodically, at most once every $k$ steps; 4. $true \neg g \neg g \neg g \neg g \neg g$ end, achievement of $g$ is rewarded whenever it occurs within $k$ steps ($k \geq 1$) of a state where $\neg g$ holds; 5. $true \neg g \neg g \neg g \neg g$ end, reward issued whenever $g$ is achieved and followed immediately by $g$ and then by $g$; 6. $true \neg g \neg g \neg g \neg g$ end, achievement of $g$ is rewarded whenever it follows $c$; 7. $true \neg g \neg g \neg g \neg g$ end, only the first achievement of $g$ that follows $c$ is rewarded; 8. $true \neg g \neg g \neg g$ end, $g$ is rewarded whenever it follows $c$ immediately; 9. $true \neg g \neg g \neg g \neg g$ end, only the first achievement of $g$ occurring within $k$ steps ($k \geq 1$) of $c$ is rewarded; 10. $true \neg g \neg g \neg g \neg g$ end, only the first achievement of $g$ occurring within $k$ steps ($k \geq 1$) of $c$ is rewarded; 11. $\neg g$ end, reward issued if $g$ has always been true; 12. $\neg g$ end, the holding of $c$ until $g$ is rewarded.

To appreciate the power of LDLₐ, consider the following example. Suppose one prefers policies for a physician with the following structure (on top of whatever other requirements/rewards exist): the physician should work in clear phases: check patient ($a$) then update patient record ($b$) repeatedly. Occasionally, and always in the end, after treating
When the rewards are Markovian, one can compute \( LTL_f \) formula would be:

\[
\begin{align*}
& (a \wedge W a) \\
& \Box (b \rightarrow \Box (c \wedge \Box (d))) \\
& \Box (c \wedge \Box (d)) \\
& \Box (d) \\
& \Box (ac) \\
& \Box (a \wedge b)
\end{align*}
\]

If \( a, b, c \) are non exclusive formulas (they can be true simultaneously), the equivalent \( LTL_f \) formula can be much more complex (depending on the actual formulas \( a, b, c \)), and in some cases it may not exist at all (as in the limit case where \( a = b = c = \text{true} \)).

\( LTL_f \), differently from \( LTL_f \), can also easily express procedural constraints (De Giacomo and Vardi 2013; Fritz and McIlraith 2007; Baier et al. 2008), so one can reward the traces satisfying such constraints. To see this, consider a sort of propositional variant of Golog (Levesque et al. 1997):

\[
\begin{align*}
\varrho & := A | \phi ? | \varrho_1 + \varrho_2 | \varrho_2 | \varrho^* | (A \text{ prop}) \\
& \text{if } \phi \text{ then } \varrho_1 \text{ else } \varrho_2 \text{ while } \phi \text{ do } \varrho.
\end{align*}
\]

These programs correspond to \( LTL_f \) path expressions, with if and while abbreviations (Fischer and Ladner 1979):

\[
\begin{align*}
& \text{if } \phi \text{ then } \varrho_1 \text{ else } \varrho_2 \equiv (\phi^*; \varrho_1) + (\neg \phi^*; \varrho_2) \\
& \text{while } \phi \text{ do } \varrho \equiv (\phi^*; \varrho)^*; \neg \phi
\end{align*}
\]

We can reward the traces that follow such (nondeterministic) programs. E.g.: At every point, if it is hot then, if the air-conditioning is off, turn it on, else do not turn it off:

\[
[\text{true}^*] \text{if (hot) then if (not on) then turn on air else turn off air [true] end}
\]

As another example: Alternate the following two instructions: while it is hot, if the air-conditioning is off then turn it on, else do not turn it off; do something for one step:

\[
\langle (\text{while (hot) do if (not on) then turn on air else turn off air [true]) end} \rangle
\]

**Building an Equivalent Markovian Model**

When the rewards are Markovian, one can compute \( \rho^* \) (for stationary \( \rho \)) and an optimal policy \( \rho^* \) using Bellman’s dynamic programming equations (Puterman 2005). However, this is not the case when the reward is non-Markovian, and thus the optimal policy may not be stationary. The standard solution is to formulate an extended MDP (with Markovian rewards) that is equivalent to the original NMRDP (Bacchus, Boutilier, and Grove 1996; Thiébaux et al. 2006).

**Definition 1** (Bacchus, Boutilier, and Grove 1996). An NMRDP \( M = \langle S, A, Tr, R \rangle \) is equivalent to an extended MDP \( M' = \langle S', A, Tr', R' \rangle \) if there exist two functions \( \tau : S' \rightarrow S \) and \( \sigma : S \rightarrow S' \) such that

1. \( \forall s \in S : \tau(\sigma(s)) = s \);
2. \( \forall s_1, s_2 \in S \) and \( s'_1 \in S' \): if \( Tr(s_1, a, s_2) > 0 \) and \( \tau(s'_1) = s_1 \), there exists a unique \( s'_2 \in S' \) such that \( \tau(s'_2) = s_2 \) and \( Tr(s'_1, a, s'_2) = Tr(s_1, a, s_2) \);
3. For any feasible trajectory \( \langle s_0, a_1, \ldots, s_{n-1}, a_n \rangle \) of \( M \) and \( \langle s'_0, a_1, \ldots, s'_{n-1}, a_n \rangle \) of \( M' \), such that \( \tau(s'_0) = s_0 \) and \( \sigma(s'_0) = s'_0 \), we have \( R((s_0, a_1, \ldots, s_{n-1}, a_n)) = R'((s'_0, a_1, \ldots, s'_{n-1}, a_n)) \).

As in previous work, we restrict our attention to extended MDPs such that \( S' = Q \times S \), for some set \( Q \).

Given an NMRDP \( M = \langle S, A, Tr, R \rangle \), we now show how to construct an equivalent extended MDP. First, using the methods described earlier, construct for each new formula \( \varphi \) its corresponding (minimal) DFA, \( A_{\varphi} = (2^P Q_1, q_0, \delta_1, F_1) \) (notice that \( S \subseteq 2^P \) and \( \delta_1 \) is total).

Then, define the equivalent extended MDP \( M' = \langle S', A', Tr', R' \rangle \) where:

- \( S' = Q_1 \times \cdots \times Q_m \times S \) is the set of states;
- \( A' = A \);
- \( Tr' : S' \times A' \times S' \rightarrow [0, 1] \) is defined as follows:

\[
Tr'(q_1, \ldots, q_m, s, a, q'_1, \ldots, q'_m) = \\
\begin{cases} 
Tr(s, a, s') & \text{if } \forall i : \delta_1(q_i, s) = q'_i \\
0 & \text{otherwise;}
\end{cases}
\]

That is, the state space is a product of the states of the original MDP and the various automata. The action set is the same. Given action \( a \), the \( S \)-component of the state progresses according to the original MDP dynamics, and the other components progress according to the transition function of the corresponding automaton. Finally, in every state, and for every \( 1 \leq i \leq m \), the agent receives the reward associated with \( \varphi_i \) if the DFA \( A_{\varphi_i} \) reached a final state.

**Theorem 2.** The NMRDP \( M = \langle S, A, Tr, R \rangle \) is equivalent to the extended MDP \( M' = \langle S', A', Tr', R' \rangle \).

**Proof.** Recall that every \( s' \in S' \) has the form \( \langle q_1, \ldots, q_m, s \rangle \). Define the state \( \delta(q_1, \ldots, q_m, s) = s \). Define the state \( \delta(s) = \langle q_0, q_0, q_0, \ldots, q_0, q_0 \rangle \). We have \( \delta(s) = s \). Condition 2 of Def. 1 is easily verifiable by inspection. For condition 3, consider a possible trace \( \pi = \langle s_0, a_1, \ldots, s_{m-1}, a_m \rangle \). We use \( \delta \) to obtain \( s'_0 = \delta(s_0) \) and given \( s_i \), we define \( s'_i \) (for \( 1 \leq i \leq n \)) to be the unique state \( \langle q_{j_1}, \ldots, q_{j_m}, s_i \rangle \) such that \( q_{j_i} = \delta(q_{j_i-1}, a_i) \) for all \( 1 \leq i \leq m \). We now have a corresponding possible trace of \( M' \), i.e., \( \pi' = \langle s'_0, a'_1, s'_1, \ldots, s'_{m-1}, a'_m \rangle \). This is the only feasible trajectory of \( M' \) that satisfies Condition 3. The reward at \( \pi' = \langle s'_0, a'_1, s'_1, \ldots, s'_{m-1}, a'_m \rangle \) depends only on whether or not each formula \( \varphi_i \) is satisfied. However, by construction of the automaton \( A_{\varphi_i} \) and the transition function \( Tr' \), \( \pi' = \varphi_i \iff s'_i = (q_1, \ldots, q_m, s_i) \) and \( q_i \in F_i \).
Theorem 3 ((Bacchus, Boutilier, and Grove 1996)). Given an NMRDP \( \mathcal{M} \), let \( \rho' \) be an optimal policy for an equivalent MDP \( \mathcal{M}' \). Then, policy \( \rho \) for \( \mathcal{M} \) that is equivalent to \( \rho' \) is optimal for \( \mathcal{M} \).

Minimality and Compositionality

One of the main aims of previous work on NMRDP specification methods was to help minimize the size of the resulting MDP. We now explain the attractive minimality properties supported by our construction. Note that the minimization discussed below is w.r.t. the extended MDP. It is quite possible that the original NMRDP can be minimized by removing duplicate states. Its minimization is orthogonal to the issue of minimizing the extended MDP, and hence, we assume that the NMRDP itself is already minimal.

Constructing a Minimal MDP. The Markovian model is obtained by taking the synchronous product of the original MDP and a DFA that is itself the synchronous product of smaller DFA’s, one for each formula. We can apply the simple, standard automaton minimization algorithm to obtain a minimal automaton, thus obtaining a minimal MDP. But even better, as we show below, it is enough to ensure that each DFA \( A_{\varphi_i} \) in the above construction is minimal to ensure the overall minimality of the extended MDP.

Theorem 4. If every automaton \( A_{\varphi_i} \) (1 \( \leq i \leq m \)) is minimal then the extended MDP defined above is minimal.

Proof. Let \( A_s \) be the synchronous product of \( A_{\varphi_i} \) (1 \( \leq i \leq m \)). We show that no two distinct states of the synchronous product \( A_s \) are equivalent, and therefore, all of them are needed, hence the thesis.

Suppose two distinct states of the synchronous product \( A_s \) are equivalent. Then, being \( A_s \) a DFA, such two states are bisimilar. Two states of \( A_s \) are bisimilar (denoted by \( \sim \)) iff:

- for all \( i \), \( q_i \in F_i \) iff \( t_i \in F_i \);
- for all \( a \), \( \delta_s(q_1, \ldots, q_m) = (q'_1, \ldots, q'_m) \) implies \( \delta_s(t_1, \ldots, t_m, a) = (t'_1, \ldots, t'_m) \) and \( (q'_1, \ldots, q'_m) \sim (t'_1, \ldots, t'_m) \);
- for all \( a \), \( \delta_s(q_1, \ldots, q_m, a) = (q'_1, \ldots, q'_m) \) and \( (q'_1, \ldots, q'_m) \sim (t'_1, \ldots, t'_m) \).

Now we show that \( (q_1, \ldots, q_m) \sim (t_1, \ldots, t_m) \) implies \( q_i = t_i \), for all \( i \). To check this we show that the relation “project on \( i \)”, \( \Pi_i((q_1, \ldots, q_m) \sim (t_1, \ldots, t_m)) \) extracting the \( i \)-th component on the left and on the right of \( \sim \) is a bisimulation for states in \( A_s \). Indeed it is immediate to verify that \( \Pi_i((q_1, \ldots, q_m) \sim (t_1, \ldots, t_m)) \) implies

- \( q_i \in F_i \) iff \( t_i \in F_i \);
- for all \( a \), \( \delta_i(q_i, a) = q'_i \) implies \( \delta_i(t_i, a) = t'_i \) and \( \Pi_i((q'_1, \ldots, q'_m) \sim (t'_1, \ldots, t'_m)) \);
- for all \( a \), \( \delta_i(t_i, a) = t'_i \) implies \( \delta_i(q_i, a) = q_i \) and \( \Pi_i((q'_1, \ldots, q'_m) \sim (t'_1, \ldots, t'_m)) \).

Hence if there are two distinct states \( (q_1, \ldots, q_m) \sim (t_1, \ldots, t_m) \) then at least for one \( i \) it must be the case that \( q_i \) and \( t_i \) are distinct and bisimilar and hence equivalent. But this is impossible since each DFA \( A_{\varphi_i} \) is minimal. \( \square \)

In general, the synchronous product of minimal DFA’s can be minimized further. The theorem shows that no further minimization is possible if the final states of the automata are kept distinct, to assign proper rewards. This is not required if one does not need to distinguish between multiple reward formulas (in which case, final states can be conjoined).

This theorem also implies that the construction is compositional and, hence, incremental: If a new formula is added, we need not change the MDP, but simply extend it with one additional component. If the original MDP was minimal and the new component is minimal, so is the resulting MDP.

Generating Reachable States Only. Minimizing the state space of the extended MDP is important if one wants to apply classical dynamic programming algorithms such as value iteration (Bellman 1957) and policy iteration (Howard 1960). However, these methods typically do not scale up as the size of \( S \) increases. Instead, search-based algorithms, such as LAO* (Hansen and Zilberstein 2001), LRTDP (Bonet and Geffner 2003), or MCTS (Kocsis and Szepesvári 2006) are preferred. Their main advantage is that they explore only a subset of the reachable states. These states are generated by progressing the initial state with a sequence of actions. For this reason, (Thiébaux et al. 2006) developed an approach in which progression can be applied to generate the states of the extended MDP, as well. That is, one does not need to enumerate the entire set \( S \) of the original NMRDP and the entire set \( S' \) of the extended MDP a priori.

We, too, support progression, and in a manner that is simpler to understand and analyze. Recall that the automaton that tracks the satisfaction of a reward formula can be constructed by progression, and that the states of the extended MDP are simply vectors that represent the state of the NMRDP and the state of the automaton for each reward formula. Hence, we can trivially support progression by simply progressing each component of this vector in the standard manner.

As observed by (Thiébaux et al. 2006), when progression is used, the constructed states may not be minimal. That is, one may generate two extended states that have an identical underlying MDP state but a different extended part, yet both states are equivalent in the sense that we get the same reward behavior from both. This issue is easy to understand with our automata-based construction: It is simply a result of the fact that the automaton constructed by progression is not minimal. Our construction provides the user with a clear set of options:

1. Apply simple progression, possibly generating a non-minimal automaton for some of the formulas. This is essentially the approach of (Thiébaux et al. 2006). (But see the discussion in Section .)
2. Build the automata for the reward formulas and minimize them off-line before starting search. Then, apply progression using the minimal automata. This approach is attractive because the reward formulas are typically much smaller than the MDP and so is the size of their automata.
Hence the effort of constructing and minimizing them will not be significant. Yet, by minimizing them a-priori, we can reduce the size of the combined state space (which is a product of the two) significantly.

3. Apply a more complex progression algorithm that generates a minimal automaton on-the-fly in time quadratic in the size of the minimal automata for the reward formulas (Lee and Yannakakis 1992).

**Getting rewards for complete traces only.** We may want to reward an agent for its entire behavior rather than for each prefix of it. This means that the value of a sequence \( \pi = (s_0, a_1, s_1 \ldots, s_{n-1}, a_n) \) is defined as follows:

\[
v(\pi) = \sum_{i:\pi_i=\varphi_i} r_i
\]

Behaviors optimal w.r.t. this definition may differ from those optimal w.r.t. the original definition in which rewards are collected following each action. Now, an agent must attempt to make as many formulas true at once, as it does not get any “credit” for having achieved them in the past.

Given an NMRDP \( \mathcal{M} \) with the above reward semantics, we can easily generate an equivalent MDP using the above construction, preceded by the following steps:

1. Add a special action \textit{stop} to \( A \).
2. Add a new proposition \textit{done} to \( S \).
3. No action is applicable in a state in which \textit{done} is true.
4. The only effect of the \textit{stop} action is to make \textit{done} be true.
5. Convert every reward formula \( \varphi_i \) to \textit{done} \& \( \varphi_i \).

Interestingly, when focussing on complete traces, our framework becomes an extension of Goal MDP planning that handles temporally extended goals, see, e.g., Chapter 6 and Chapter 4 of (Geffner and Bonet 2013).

**Comparison with previous proposals**

**Capturing PLTL rewards.** Our proposal can be seen as extending (Bacchus, Boutilier, and Grove 1996). There, rewards are assigned to partial traces whenever the last state of the trace satisfies past-LTL (PLTL) formulas. Without introducing explicitly PLTL, but given a partial trace \( \pi_0, \ldots, \pi_n \) we reverse it into \( \pi_n, \ldots, \pi_0 \) and evaluate it over the LTL\(_f \) formula \( \varphi \) obtained from the PLTL formula by simply replacing the past operators with the corresponding future operators (e.g., replace \textit{since} with \textit{eventually}). Then, the setting remains analogous to the one shown above. In particular, we can construct the NFA \( A_{\varphi} \) associated with \( \varphi \) and, instead of reversing the partial traces, reverse \( A_{\varphi} \), thus getting an NFA \( A_{\varphi}^{-} \), by simply reversing the edge directions and switching initial and final states. This can be done in linear time. If we now determinize (and minimize) \( A_{\varphi}^{-} \), getting the (minimal) DFA \( A_{\varphi} \), we can proceed exactly as above.

Given this essential equivalence of PLTL and LTL\(_f \), and the fact that LTL\(_f \) is strictly more expressive than LTL\(_f \), we conclude that our setting is strictly more expressive than the one in (Bacchus, Boutilier, and Grove 1996). In addition, unlike (Bacchus, Boutilier, and Grove 1996), our automata construction algorithm is based on progression, allowing us to use information about the initial state to prevent the generation of unreachable states.

**Comparing with SFLTL rewards.** In (Thiébautx, Kabanza, and Slaney 2002; Gretton, Price, and Thiébautx 2003; Thiébautx et al. 2006) a sophisticated temporal logic, called SFLTL is introduced, which is able to specify explicitly when a partial (finite) trace gets a reward. In fact, many formulas in this logic cannot naturally be interpreted as specifying reasonable rewards. Thus, the results of (Thiébautx et al. 2006) focus on a class of formulas called reward-normal. This class has been further studied in (Slaney 2005) and in (Gretton 2014). In this work it is first shown that a notable fragment of reward-perfect formulas is equivalent in expressive power to star-free RE, and hence equivalent to LTL\(_f \). Then, reward-normal formulas are shown to be expressible as RE, and hence expressible in LDL\(_f \). Moreover, in (Gretton 2014), a variant of reward-normal formulas is introduced with exactly the same expressive power as RE\(_f \), and hence equivalent to LDL\(_f \). Interestingly, the reduction to RE is based on an actual translation into finite state automata.

Besides the useful ability to build the extended MDP by progression, (Thiébautx et al. 2006) discuss a minimality notion called blind minimality where no two states in the extended MDP lead to the same reward behavior under all conceivable futures (where a “conceivable future” is any future sequence of states and actions, including ones that are not reachable). Observe that if one considers an NMRDP with a single state, this amounts to being able to generate the minimal automata for the reward formulas by progression. We are unaware of any general method for providing such a construction in linear time, which further hints that the set of formulas for which this result holds is rather limited. Given the above, we cannot claim to provide blind minimality. We can only speculate that given the similarity between their progression algorithm and the standard progression algorithm in cases where they are both defined, we should be able to offer similar guarantees. Thus, the formalism developed here is simpler syntactically and semantically, most likely more expressive, can support minimization without requiring the construction of the NMRDP, but supports progression as well, and is compositional.

**Conclusion**

We presented a new language for specifying non-Markovian rewards in MDPs. Our language is based on LTL\(_f \)/LDL\(_f \) and is more expressive than previous proposals and being based on a standard temporal logic of the future, is likely to be more intuitive to use. We showed how to construct a minimal equivalent MDP, and since we rely on general methods for tracking temporal formulas, the construction is cleaner. Being based on progression, it can use information about the initial state to prune unreachable states.

In future work we intend to examine the use of monitoring notions developed for LTL\(_f \) and LDL\(_f \) (Bauer, Leucker, and Schallhart 2010; De Giacomo et al. 2014; Maggi et al. 2011). Using such monitors one could extract early rewards that guide the process to get full rewards later. Another important direction for future work is exploiting non-Markovian
rewards in reinforcement learning (RL) to provide better guidance to the learning agent, as well as extending inverse RL methods to learn to assign non-Markovian rewards in a state.

Finally, we observe that LDL₁ can capture all co-safe LTL formulas, i.e., LTL formulas whose automaton is a DFA (instead of a Büchi automaton). Co-safe LTL formulas were used in Robotics to select traces of an MDP that represent behaviors of interest (Lacerda, Parker, and Hawes 2014; 2015). Thus, instead of co-safe LTL, we can use LDL₁ to gain expressivity in trace specification at essentially no cost.

Acknowledgements: The first author is supported in part by ISF grant 933/13 and the Lynn and William Frankel Center for Computer Science. The work was partially supported by the Sapienza project “Immersive Cognitive Environments”.

References
Camacho, A.; Triantafillou, E.; Muise, C.; Baier, J. A.; and McIlraith, S. 2017c. Non-deterministic planning with temporally extended goals: LTL over finite and infinite traces. In AAAI.
De Giacomo, G., and Vardi, M. Y. 2013. Linear temporal logic and linear dynamic logic on finite traces. In IJCAI.
De Giacomo, G., and Vardi, M. Y. 2016. LTL₁ and LDL₁ synthesis under partial observability. In IJCAI.
Lacerda, B.; Parker, D.; and Hawes, N. 2014. Optimal and dynamic planning for Markov decision processes with co-safe LTL specifications. In IROS.
Pnueli, A. 1977. The temporal logic of programs. In FOCS.
Torres, J., and Baier, J. A. 2015. Polynomial-time reformulation of LTL temporally extended goals into final-state goals. In IJCAI.
Vardi, M. Y. 2011. The rise and fall of linear time logic. In GandALF.