LTL_f Synthesis on Probabilistic Systems

Andrew M. Wells
Rice University, Houston, Texas
andrew.wells@rice.edu

Morteza Lahijanian
University of Colorado, Boulder, Colorado
morteza.lahijanian@colorado.edu

Lydia E. Kavraki
Rice University, Houston, Texas
kavraki@rice.edu

Moshe Y. Vardi
Rice University, Houston, Texas
vardi@rice.edu

Many systems are naturally modeled as Markov Decision Processes (MDPs), combining probabilities and strategic actions. Given a model of a system as an MDP and some logical specification of system behavior, the goal of synthesis is to find a policy that maximizes the probability of achieving this behavior. A popular choice for defining behaviors is Linear Temporal Logic (LTL). Policy synthesis on MDPs for properties specified in LTL has been well studied. LTL, however, is defined over infinite traces, while many properties of interest are inherently finite. Linear Temporal Logic over finite traces (LTL_f) has been used to express such properties, but no tools exist to solve policy synthesis for MDP behaviors given finite-trace properties. We present two algorithms for solving this synthesis problem: the first via reduction of LTL_f to LTL and the second using native tools for LTL_f. We compare the scalability of these two approaches for synthesis and show that the native approach offers better scalability compared to existing automaton generation tools for LTL.

1 Introduction

Many real-world systems are stochastic in nature. They evolve in the world according to action (control) decisions and the uncertainty embedded in the execution of those actions, resulting in stochastic behavior. *Formal synthesis* studies how the system should choose actions so that it can increase the chances of achieving a desirable behavior. To allow such reasoning, *Markov Decision Processes* (MDPs) are typically used to model these systems since MDPs effectively capture sequential decision-making and probabilistic evolutions [5]. The desired behavior is expressed in a formal language, which yields expressive and unambiguous specifications. Temporal logics are a common choice for this language since they allow a combination of temporal and boolean reasoning over the behavior of the system. Most temporal logics are interpreted over behaviors with infinite time durations, but some behaviors are inherently finite and can be expressed more intuitively and more practically using a temporal logic with finite semantics [14, 13]. This work investigates policy synthesis on MDPs for specifications expressed in a formal language called *Linear Temporal Logic over finite traces* (LTL_f) [9], which reasons over system behaviors with finite horizons.

A popular specification language in formal verification is Linear Temporal Logic (LTL) [20]. LTL provides a natural description of temporal properties over an infinite trace. LTL can express properties such as order, e.g., "first *a* and next eventually *b*," and lack of starvation, e.g., "globally eventually resources are granted." While infinite-trace properties are essential for reasoning about many systems, other systems require finite-trace properties [14, 13], for example robot planning for finite behaviors. If a robot is to build an object, the interest is in the finite rather than the infinite traces that accomplish this task. In such cases LTL_f [9] is a suitable alternative to LTL, cf. [13].

Submitted to:
GandALF 2020

© Wells et al.
This work is licensed under the
Creative Commons Attribution License.

A number of recent studies has focused on developing efficient frameworks for solving LTL_f-reasoning problems, e.g., [9, 10, 25, 14, 18, 13]. De Giacomo and Vardi [9] presented a translation of LTL_f to LTL, implying that existing tools for LTL satisfiability and synthesis can be used for LTL_f with suitable transformations. In [10], the problem of reactive synthesis from LTL_f specifications that contain system (controllable) and environment (uncontrollable) variables is considered. The work shows that this problem reduces to strategy synthesis in a two-player automaton game, which is a 2EXPTIME-complete problem. To enable practical solutions to this problem, [25] introduces a symbolic approach to the LTL_f synthesis problem. The applications of those results in the field of robot planning are studied in [14, 12, 13]. In [18], the fundamental problem of LTL_f satisfiability checking is studied through a SAT-based approach. The results of those studies show that more scalable tools can be built by treating LTL_f *natively*, rather than reducing to LTL, even though LTL reasoning is a more mature and extensively studied domain. The underlying assumption in all those works is that the system is either purely deterministic or purely nondeterministic. Hence, it is natural to ask whether similar results can be obtained for probabilistic systems. That is, do native LTL_f techniques outperform reasoning by reduction to LTL?

Formal synthesis on probabilistic systems has been extensively studied in the formal-verification literature. In those studies, MDPs are a de facto modeling tool for the probabilistic system, and LTL and PCTL (probabilistic computation temporal logic) [5] are the specification languages of choice. For LTL synthesis, the approach is based on first translating the specification to a *deterministic Rabin automaton* (DRA) and then solving a stochastic shortest path problem [22, 8] on the product of the DRA with the MDP. Tools such as PRISM [17] can solve this LTL synthesis problem efficiently with their symbolic engine. LTL_f synthesis for probabilistic systems, however, has not been studied, and it is not clear whether the same tools can be extended for LTL_f reasoning with a similar efficiency.

Assigning rewards based on temporal goals has also been studied in the context of planning. In [21], a logic similar to LTL_f, \$FLTL, is considered; however, this logic cannot express properties such as "Eventually." In [3, 4], PastLTL is used to describe finite properties associated with rewards. PastLTL and LTL_f naturally express different properties (see [21]). In [7, 6], rewards can be attached to LTL_f or *linear dynamic logic on finite traces* (LDL_f) [9] formulas. In all of these MDP planning works, the goal is to maximize rewards rather than to study the behavior of the MDP. These approaches use approximations that give lower bounds and converging only in the limit. Thus, they cannot necessarily give a negative answer to a decision query about a synthesis problem (e.g., does a policy with at least 95% probability into success exist?).

In this work, we present the problem of LTL_f synthesis for MDPs. In approaching this problem, we specifically seek to answer the empirical question of which approach is more efficient; an approach based on the mature and well-studied LTL synthesis or an approach based the unique properties of LTL_f itself. The answer to this question can have a broader impact on the employment of formal methods for probabilistic systems. Hence, we introduce two solutions to this problem. The first one is based on a reduction to an LTL synthesis problem, and the second one is a native approach. For the first approach, we show a translation of the LTL_f specification to LTL and a corresponding augmentation of the MDP that allows us to use standard tools for LTL synthesis. For the second approach, we use specialized tools to obtain an automaton that we input to standard tools in order to solve the problem.

Even though both approaches have the same theoretical complexity bound [5, 10], we demonstrate that the native approach scales better than the translation to LTL through a series of benchmarking case studies. For a complete, online version of the paper, please see [24]. Our code and examples are available on GitHub [23].

2 Problem Formulation

Our goal is to synthesize a policy for an MDP such that the probability of satisfying a given LTL_f property is maximized. First we introduce the formalisms needed to define this problem. In Section 2.1, we introduce Markov processes and define the labeling of a path on these processes and the probability measure associated with a set of paths. Next, we introduce LTL and the finite-trace version LTL_f in Section 2.2. We then define when a path on an MDP to satisfies an LTL or LTL_f formula as well as the probability of satisfaction in Section 2.3. Finally, we give our formal problem definition in Section 2.4.

2.1 Markov Processes

Markov processes are frequently used to model systems that exhibit stochastic behavior. While this paper deals with MDPs, it is also important to define *Discrete-Time Markov Chains* (DTMCs) as they are needed to define probability measures.

Definition 1 (DTMC)**.** *A labeled* Discrete-Time Markov Chain *(DTMC) is a tuple* $\mathscr{D} = (S, P, s_{init}, AP, L)$*, where* S *is a countable set of states,* $s_{init} \in S$ *is the initial state,* AP *is a finite set of atomic propositions,* $L : S \to 2^{AP}$ *is a labeling function, and* $P : S \times S \to [0,1]$ *is a transition probability function where* $\sum_{s' \in S} P(s,s') = 1$ *for all* $s \in S$*.*

An execution of a DTMC is given by a path as defined below.

Definition 2 (Path)**.** *A* path w *through a DTMC is an infinite sequence of states:*

$$w = s_0 s_1 \dots s_n \dots, \qquad \textit{such that} \qquad P(s_i, s_{i+1}) > 0 \quad \forall i \geq 0.$$

$Paths(s)$ is the set of paths starting in s. A finite path is one with a last state s_n. $Paths_{fin}(s)$ is the set of all finite paths starting in s. For every path $w \in Paths(s)$, $pre(w)$ denotes the set of prefixes of w (similarly for $w \in Paths_{fin}(s)$).

A trace or labeling of a path is the sequence of labels of the states in the path.

Definition 3 (Labeling of a Path)**.** *A* labeling *(also referred to as* valuation *or* trace*) of an infinite path* $w = s_0 s_1 \dots s_n \dots$ *is the sequence* $L(s_0)L(s_1)\dots L(s_n)\dots$ *The labeling of a finite path is defined analogously. We use* $L(w)$ *to denote the labeling of* w*.*

To reason over the paths of a DTMC probabilistically, we need to define a probability space with a probability measure over infinite paths. To this end, we first define cylinder sets that extend a finite path to a set of infinite paths.

Definition 4 (Cylinder Set for DTMC)**.** *The cylinder set for some finite path* $w \in Paths_{fin}(s_{init})$*, denoted by* $Cyl(w)$*, is the set of all infinite paths that share* w *as a prefix:*

$$Cyl(w) = \{w' \in Paths(s_{init}) \mid w \in pre(w')\}. \tag{1}$$

Definition 5 (Probability Measure over Paths of a DTMC)**.** *For the probability space* $(\Omega, \mathscr{E}, Pr)$*, where sample space* $\Omega = Paths(s_{init})$*, event space* $\mathscr{E}$ *is the smallest* σ*-algebra on* Ω *containing the cylinder sets of all finite paths (i.e.,* $Cyl(w) \in \mathscr{E}$ *for all* $w \in Paths_{fin}(s_{init})$*), the probability measure* Pr *is defined as:*

$$Pr(Cyl(s_0 \dots s_n)) = \prod_{0 \leq i < n} P(s_i, s_{i+1}), \tag{2}$$

where $s_0 = s_{init}$*. We define* $Pr(Cyl(s_{init})) = 1$*.*

Some systems exhibit not only probabilistic behavior but also non-deterministic behavior. These systems are typically modeled as Markov Decision Processes (MDPs). MDPs extend the definition of DTMCs by allowing choices of actions at each state.

Definition 6 (MDP)**.** *A labeled* Markov Decision Process *(MDP) is a tuple:* $\mathscr{M} = (S, A, P, s_{init}, AP, L)$*, where* s_{init}*, AP and L are as in Def. 1 and:*

- *S is a finite set of states;*
- *A is a finite set of actions, and* $A(s) \subseteq A$ *denotes the set of actions enabled at state* $s \in S$*;*
- $P : S \times A \times S \to [0,1]$ *is the transition probability function where* $\sum_{s' \in S} P(s,a,s') = 1$ *for all* $s \in S$ *and* $a \in A(s)$*.*

Example 1. An example of an MDP is show in Figure 1. Actions ($A = \{a_0, a_1\}$) and probabilities are shown as edge labels. State names are within each state and state labels are above each state.

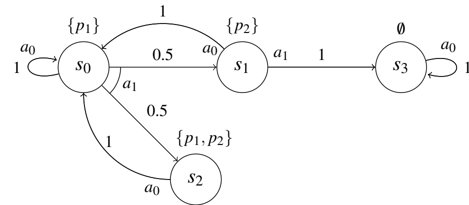


Figure 1: Example MDP.

The notion of path can be straightforwardly extended from DTMCs to MDPs.

Definition 7 (Path through MDP)**.** *A* path *w through an MDP is a state followed by a sequence of action-state pairs:*

$$w = s_0 \langle a_0, s_1 \rangle \langle a_1, s_2 \rangle \ldots \langle a_{n-1}, s_n \rangle \ldots$$

such that $s_0 = s_{init}$*,* $a_i \in A(s_i)$*, and* $P(s_i, a_i, s_{i+1}) > 0$ *for all* $i \geq 0$*.* $Paths(s)$*,* $Paths_{fin}(s)$ *and* $pre(w)$ *are defined analogously to the DTMC case.*

A *policy* (also known as an *adversary* or *strategy*) resolves the choice of actions. This induces a (possibly infinite) DTMC.

Definition 8 (Policy)**.** *A* policy $\pi : Paths_{fin} \to A$ *maps every finite path* $w = s_0 \langle a_0, s_1 \rangle \langle a_1, s_2 \rangle \ldots \langle a_{n-1}, s_n \rangle$ *to an element of* $A(s_n)$*, i.e.,* $\pi(w) \in A(s_n)$*.*[1]

The set of all policies is Π. A policy is *stationary* if $\pi(w)$ only depends on the most recent state s_n of w denoted by $last(w)$. Under policy π the action choices on $\mathscr{M}$ are determined. This gives us a DTMC whose states correspond to the finite paths of the MDP. We call this the DTMC induced on $\mathscr{M}$ by policy π and denote this by $\mathscr{D}^{\pi}$. $Paths^{\pi}(s)$ is shorthand for $Paths(s)$ of $\mathscr{D}^{\pi}$. Therefore, the probability of the paths of the MDP under π are defined according to $\mathscr{D}^{\pi}$, whose probability space includes a probability measure Pr^{π} over infinite paths via cylinder sets that extend a finite path to a set of infinite paths. See [5] for details.

2.2 Linear Temporal Logic

Linear temporal logic (LTL) is a popular formalism used to specify temporal properties. Here, we are interested in LTL interpreted over finite traces, but we must still define LTL as we use a reduction to LTL as one of our two approaches to synthesis.

Definition 9 (LTL Syntax)**.** *An* LTL *formula is built from a set of propositional symbols Prop and is closed under the boolean connectives as well as the "next" operator X and the "until" operator U:*

$$\varphi ::= \top \mid p \mid (\neg \varphi) \mid (\varphi_1 \wedge \varphi_2) \mid (X \varphi) \mid (\varphi_1 U \varphi_2),$$

where $p \in Prop$*.*

[1] Here, we focus on deterministic policies as they are sufficient for optimality of LTL (and LTL$_f$) properties on MDPs [5].

The common temporal operators "eventually" (F) and "globally" (G) are defined as: $F\,\varphi = \top\, U\, \varphi$ and $G\,\varphi = \neg F\,\neg\varphi$. The semantics of LTL are defined over infinite traces (for full definition, see [20]). An LTL formula φ defines an ω-regular language $\mathscr{L}(\varphi)$ over alphabet 2^{Prop}, i.e., $\mathscr{L}(\varphi) = \{\rho \in (2^{Prop})^\omega \mid \rho \models \varphi\}$.

Next we define LTL$_f$. To distinguish between the formulas of the two logics, we use φ for LTL and ϕ for LTL$_f$ formulas.

Definition 10 (LTL$_f$ Syntax & Semantics). *An* LTL$_f$ *formula has identical syntax to* LTL*, but the semantics is defined over finite traces. Let $|\rho|$ and ρ_i denote the length of trace ρ and the symbol in the ith position in ρ, respectively, and $\rho, i \models \phi$ is read as: "the ith step of trace ρ is a model of ϕ." Then,*

- $\rho, i \models \top$;
- $\rho, i \models p$ *iff* $p \in \rho_i$;
- $\rho, i \models \neg\phi$ *iff* $\rho, i \not\models \phi$;
- $\rho, i \models \phi_1 \wedge \phi_2$, *iff,* $\rho, i \models \phi_1$ *and* $\rho, i \models \phi_2$;
- $\rho, i \models X\phi$ *iff* $|\rho| > i+1$ *and* $\rho, i+1 \models \phi$;
- $\rho, i \models \phi_1 U \phi_2$ *iff* $\exists j$ *s.t.* $i \leq j < |\rho|$ *and* $\rho, j \models \phi_2$ *and* $\forall k,\ i \leq k < j, \quad \rho, k \models \phi_1$.

We say finite trace ρ satisfies formula ϕ, denoted by $\rho \models \phi$, iff $\rho, 0 \models \phi$. An LTL$_f$ formula ϕ defines a language $\mathscr{L}(\phi)$ over the alphabet 2^{Prop}. $\mathscr{L}(\phi)$ is a regular language, i.e., $\mathscr{L}(\phi) = \{\rho \in (2^{Prop})^* \mid \rho \models \phi\}$.

2.3 Satisfaction of Temporal Logic Specification

Here, we define what it means for an MDP to satisfy an LTL or LTL$_f$ formula.

Definition 11 (Path satisfying LTL). *For a pair ($\mathscr{M}$, φ) of an MDP and an* LTL *formula where the atomic propositions of $\mathscr{M}$ match the propositions of φ (i.e., $Prop = AP$), we say that an infinite path w on $\mathscr{M}$ satisfies specification φ if the labeling of w is in the language of φ, i.e., $L(w) \in \mathscr{L}(\varphi)$.*

Following [25], we define finite satisfaction (of an LTL$_f$ formula) as follows.

Definition 12 (Path satisfying LTL$_f$). *For a pair ($\mathscr{M}$, ϕ) of an MDP and an* LTL$_f$ *formula where the atomic propositions of $\mathscr{M}$ match the propositions of ϕ (i.e., $Prop = AP$), we say that a (possibly finite) path w of $\mathscr{M}$ satisfies specification ϕ if at least one prefix of w is in the language of ϕ, i.e.,*

$$w \models \phi \quad \Leftrightarrow \quad \exists w' \in pre(w)\ s.t.\ L(w') \in \mathscr{L}(\phi). \tag{3}$$

Intuitively, this corresponds to a system that can declare its execution complete after satisfying its goals. LTL$_f$ is suitable for specifications that are to be completed in finite time.

Definition 13 (Probability of LTL$_f$ satisfaction). *The probability of satisfying an* LTL$_f$ *property ϕ in $\mathscr{M}$ under policy π is $Pr(\mathscr{M}^\pi \models \phi) = Pr^\pi(w \in Path^\pi(s_{init}) \mid w \models \phi)$.*

2.4 Problem Statement

We formalize the problem of LTL$_f$ synthesis on MDPs as:

Problem 1 (LTL$_f$ synthesis on MDPs). *Given MDP $\mathscr{M}$ and an* LTL$_f$ *formula ϕ, compute a policy π^* that maximizes the probability of satisfying ϕ, i.e.,*

$$\pi^* = \arg\max_{\pi \in \Pi} Pr(\mathscr{M}^\pi \models \phi),$$

as well as this probability, i.e., $Pr(\mathscr{M}^{\pi^*} \models \phi)$.

3 Synthesis Algorithms

We introduce two approaches to LTL$_f$ policy synthesis on MDPs. The first approach is based on reduction to classical LTL policy synthesis on MDPs. The second approach is through a translation of LTL$_f$ formulas to first order logic formulas, which can be translated to a symbolic deterministic automaton. In this section, we detail these two algorithms, and in Section 4, we show that the second approach scales better than the classical LTL approach.

3.1 Reduction to LTL Synthesis

We can reduce the problem of LTL$_f$ policy synthesis on MDPs to a classical LTL policy synthesis. The algorithm consists of two main steps: (1) construction of an MDP $\mathcal{M}'$ from $\mathcal{M}$ by augmenting it with an additional state and atomic proposition, and (2) translation of LTL$_f$ formula ϕ on the labels of $\mathcal{M}$ to its equivalent LTL formula φ on the labels of $\mathcal{M}'$.

3.1.1 Augmented MDP

Recall that the semantics of LTL$_f$ formulas is over finite traces whereas the interpretation of LTL formulas is over infinite traces. In order to reduce the LTL$_f$ synthesis problem to an LTL one, we need to be able to capture the finite paths (traces) of $\mathcal{M}$ that satisfy ϕ and extend them to infinite paths (traces). Specifically, we need those satisfying finite paths that contain no ϕ-satisfying prefixes, i.e., satisfy ϕ for the first time. To this end, we allow the environment (policy) to decide when to "terminate." We view the system to be "alive" until termination, at which point it is no longer alive. Then, we define an LTL formula that requires the system to be alive while it has not satisfied ϕ and to terminate after satisfying ϕ.

To this end, we augment MDP $\mathcal{M}$ with a terminal action and state and an atomic proposition *alive*. Formally, we construct MDP $\mathcal{M}' = (S', A', P', s'_{\text{init}}, AP', L')$, where $S' = S \cup \{s_{\text{term}}\}$, $A' = A \cup \{a_{\text{term}}\}$, $s'_{\text{init}} = s_{\text{init}}$, $AP' = AP \cup \{alive\}$,

$$A'(s) = \begin{cases} A(s) \cup \{a_{\text{term}}\} & \text{if } s \neq s_{\text{term}} \\ \{a_{\text{term}}\} & \text{if } s = s_{\text{term}} \end{cases}, \qquad L'(s) = \begin{cases} L(s) \cup \{alive\} & \text{if } s \neq s_{\text{term}} \\ \emptyset & \text{if } s = s_{\text{term}} \end{cases},$$

$$P'(s,a,s') = \begin{cases} P(s,a,s') & \text{if } s \in S,\, a \in A, s' \in S \\ 0 & \text{if } s \in S,\, a \in A, s' = s_{\text{term}} \\ 1 & \text{if } s \in S', a = a_{\text{term}}, s' = s_{\text{term}} \end{cases}.$$

In this MDP, the system can decide to terminate by taking action a_{term}, in which case it transitions to state s_{term} with probability one and remains there forever. The labeling of the corresponding path includes the atomic proposition *alive* at every time step until s_{term} is visited and is empty thereafter.

Example 2. Figure 2 illustrates the augmented MDP $\mathcal{M}'$ constructed from the example MDP $\mathcal{M}$ in Figure 1. State s_{term} along with the dashed edges and atomic proposition *alive* are added to $\mathcal{M}$. The dashed edges are enabled by action a_{term} and have transition probability of one. The label of s_{term} is the empty set, and the labels of the rest of the states contain *alive*.

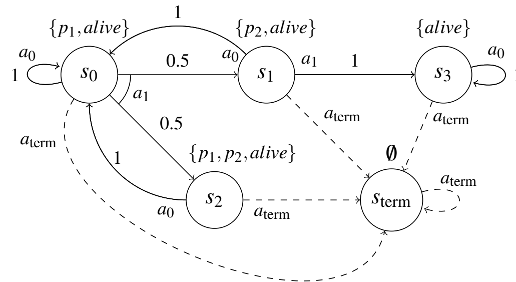


Figure 2: Augmented MDP constructed from $\mathcal{M}$ in Fig. 1.

With this augmentation, the system is able to terminate once a satisfying finite path is generated. Then, we can show that the probability of the set of infinite paths with satisfying prefixes on MDP $\mathcal{M}$ is equal to the probability of infinite paths of $\mathcal{M}'$ in the form $w_{fin}(s_{\text{term}})^{\omega}$, where w_{fin} is a satisfying prefix, and $(s_{\text{term}})^{\omega}$ is the suffix.

Let $Paths_{fin,\phi} \subseteq Paths^{\pi}_{fin}(s_{\text{init}})$ be a set of finite paths of interest of MDP $\mathcal{M}$ under policy $\pi \in \Pi$. Denote the probability of infinite paths with prefixes from $Paths_{fin,\phi}$ by

$$Pr(\mathcal{M}^{\pi} \models Paths_{fin,\phi}) = Pr^{\pi}(w \in Path^{\pi}(s_{\text{init}}) \mid pre(w) \cap Paths_{fin,\phi} \neq \emptyset). \tag{4}$$

Furthermore, define policy $\pi' \in \Pi'$ on the augmented MDP $\mathcal{M}'$ as

$$\pi'(w_{fin}) = \begin{cases} \pi(w_{fin}) & \text{if } w_{fin} \notin Paths_{fin,\phi} \ \wedge \ last(w_{fin}) \neq s_{\text{term}} \\ a_{\text{term}} & \text{otherwise} \end{cases} \tag{5}$$

Then, it can be shown (see Lemma 2) that: $Pr(\mathcal{M}^{\pi} \models Paths_{fin,\phi}) = Pr(\mathcal{M}'^{\pi'} \models Paths_{fin,\phi})$. Lemma 2 generalizes this result.

Lemma 2. *Let $Paths_{fin,\phi} \subseteq Paths_{fin}(s_{init})$ be a set of finite paths of interest of MDP $\mathcal{M}$. Further, let Π' be a set of all policies of augmented MDP $\mathcal{M}'$ such that if $\pi' \in \Pi'$, then*

$$\pi'(w_{fin}) \in \begin{cases} \{a_{term}\} & \textit{if } w_{fin} \in Paths_{fin,\phi}(s_{init}) \vee last(w_{fin}) = s_{term} \\ A(last(w_{fin})) & \textit{otherwise} \end{cases} \tag{6}$$

It holds that: $\max_{\pi \in \Pi} Pr(\mathcal{M}^{\pi} \models Paths_{fin,\phi}) = \max_{\pi' \in \Pi'} Pr(\mathcal{M}'^{\pi'} \models Paths_{fin,\phi})$.

Proofs of this lemma and the below theorem are available in the appendix.

An LTL_f formula on $\mathcal{M}$ can be translated to its equivalent LTL formula on $\mathcal{M}'$ following [9].

3.1.2 LTL_f to LTL

To translate an LTL_f formula on $\mathcal{M}$ to its equivalent LTL formula on $\mathcal{M}'$, we follow [9]. Let Φ_f be the set of LTL_f formulas ϕ defined over atomic propositions in AP and Φ be the set of LTL formulas φ defined over AP'. Then $g : \Phi_f \to \Phi$ is defined as:

$$g(\phi) = t(\phi) \wedge \big(alive\, U\, (G\neg alive)\big),$$

where $t : \Phi_f \to \Phi$ is inductively defined as:

- $t(p) = (p \wedge alive)$, where $p \in AP$;
- $t(\neg\phi) = \neg t(\phi)$;
- $t(\phi_1 \wedge \phi_2) = t(\phi_1) \wedge t(\phi_2)$;
- $t(X\,\phi) = X\,(alive \wedge t(\phi))$;
- $t(\phi_1\, U\, \phi_2) = t(\phi_1)\, U\, (alive \wedge t(\phi_2))$.

In this construction, mapping t ensures that *alive* is present in the last letter of every finite trace that satisfies ϕ. Then, g translates ϕ to φ by requiring *alive* to be true until ϕ is satisfied and false thereafter. In other words, the translated LTL formula $\varphi = g(\phi)$ requires the system to terminate after it satisfies ϕ.

Theorem 3. *Given an MDP $\mathscr{M}$ and an* LTL$_f$ *formula ϕ, the maximum probability of $\mathscr{M} \models \phi$ is given by* $\max_{\pi \in \Pi} Pr(\mathscr{M} \models \phi) = \max_{\pi' \in \Pi'} Pr(\mathscr{M}' \models g(\phi))$, *where $\mathscr{M}'$ is the augmented MDP, Π' is the set of all policies in $\mathscr{M}'$, and $g(\phi)$ is the* LTL *formula obtained from ϕ.*

A proof is available in the appendix.

A direct result of the above theorem is the reduction of LTL$_f$ policy synthesis to classical LTL policy synthesis on MDPs.

Corollary 4. *The policy synthesis to maximize the probability of satisfying* LTL$_f$ *formula ϕ on $\mathscr{M}$ can be reduced to the* LTL *maximal policy synthesis problem.*

Therefore, to solve Problem 1, we can use LTL synthesis on augmented MDP $\mathscr{M}'$ and property $\varphi = g(\phi)$. The general LTL synthesis algorithm is well-established [5] and follows the following steps: (1) translation of the LTL formula to a DRA, (2) composition of the DRA with the MDP, which results in another MDP called the product MDP, (3) identification of the maximal end-components on the product MDP that satisfy the accepting condition of DRA, and finally (4) solving the maximal reachability probability problem (stochastic shortest path problem) [8] on the product MDP with the accepting end-components as the target states. There exist many tools such as PRISM [17] that can solve the LTL synthesis problem. Specifically, PRISM has a symbolic implementation of this algorithm, enabling fast computations. In Section 4, however, we show that the native approach introduced below outperforms the LTL-reduction approach even by using PRISM's symbolic engine.

3.2 Native Approach

In the native approach, we first convert the LTL$_f$ formula into a *deterministic finite automaton* (DFA) using specialized tools. Then we take the product of this automaton with the MDP. Finally, we synthesize a strategy by solving the maximal reachability probability problem on this product MDP.

3.2.1 Translation to DFA

A Deterministic Finite Automaton (DFA) is a tuple: $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ the alphabet, δ the transition function, q_0 the initial state and F the set of accept states. A finite *run* of a DFA on a trace $\rho = \rho_0\rho_1 \ldots \rho_n$, where $\rho_i \in \Sigma$, is the sequence of states $q_0 q_1 \ldots q_{n+1}$ such that $q_{i+1} = \delta(q_i, \rho_i)$ for all $0 \leq i \leq n$. This run is accepting if $q_{n+1} \in F$.

Following [25], we translate LTL$_f$ to a DFA using MONA [15]. LTL$_f$ is expressively equivalent to First-order Logic on finite words, which in turn is a fragment of Weak Second-order Theory of One Successor. We use the translation given in [9] to convert LTL$_f$ formula ϕ to a First-order Logic formula. MONA offers translations from Weak Second-order Theory of One or Two Successors (WS1S/WS2S) to a DFA that accepts precisely the language of our LTL$_f$ formula ϕ. We denote this DFA by $\mathscr{A}_\phi$.

3.2.2 Product of DFA with MDP

Given DFA $\mathscr{A}_\phi$, we can take the product with the MDP $\mathscr{M}$ to achieve a new MDP $M \times \mathscr{A}_\phi$ as follows. The product of MDP $\mathscr{M} = (S, A, P, s_{\text{init}}, AP, L)$ and DFA $\mathscr{A}_\phi = (Q, \Sigma, q_0, \delta, F)$ is an MDP

$$\mathscr{M} \times \mathscr{A}_\phi = (S \times Q, A, P^{\mathscr{M} \times \mathscr{A}_\phi}, (s_{\text{init}}, q_{\text{init}})),$$

where $q_{\text{init}} = \delta(q_0, L(s_{\text{init}}))$, and

$$P^{\mathscr{M} \times \mathscr{A}_\phi}((s,q), a, (s', q')) = \begin{cases} P(s,a,s') & \text{if } q' = \delta(q, L(s)) \\ 0 & \text{otherwise} \end{cases}.$$

The paths of this product MDP have one-to-one correspondence to the paths of MDP $\mathcal{M}$ as well as the runs of $\mathcal{A}_\phi$. Therefore, the projection of the paths of $\mathcal{M} \times \mathcal{A}_\phi$ that reach state (s,q), where $q \in F$, on $\mathcal{A}_\phi$ are accepting runs and on $\mathcal{M}$ are ϕ-satisfy paths. Thus, we solve Problem 1 by synthesizing an optimal policy on this product MDP, using standard tools for the maximal reachability probability problem as discussed in Sec. 3.1. The resulting policy is stationary on the product MDP but history-dependent on $\mathcal{M}$.

Compared to the LTL-based approach, the direct translation to a DFA offers better runtime and memory usage. Additionally, it produces a minimal DFA while the LTL pipeline produces an ω-automaton which cannot be minimized effectively by existing tools. We show the benefits of the native approach experimentally in Sec. 4.

4 Evaluation

We evaluate the proposed synthesis approaches through a series of benchmarking case studies. Below, we provide details on our implementation, experimental scenarios, and obtained results. A version of our tool along with examples is available on GitHub [23].

4.1 Experimental Framework

We run our experiments using the PRISM framework [17]. PRISM uses a symbolic encoding of MDPs as well as an encoding of automata as a list of edges, where the labels of edges are encoded symbolically using BDDs. PRISM supports several tools for the LTL-to-automata translation. We tested PRISM's built-in translator as well as Rabinizer, LTL3DRA and SPOT [16, 1, 11]. Of these, PRISM's built in conversion and SPOT performed significantly better than the others, and were used for evaluation.

In the implementation of the LTL-reduction approach, we augment the MDP and convert the LTL$_f$ formula into an equivalent LTL formula as described in Section 3.1. Then, we input both the LTL formula and the modified MDP into PRISM for synthesis. In the implementation of the native approach, we invoke PRISM on the original LTL$_f$ formula and MDP and use an external tool to convert the LTL$_f$ formula into a DFA using MONA [15] then convert from MONA's format to the HOA format [2]. Note that external tools (SPOT and our native approach) read from hard disk, whereas using PRISM's built-in conversion avoids this. Nevertheless, even including this time, the native approach (and sometimes SPOT) typically gives better performance as shown below. All experiments are run on a computer with an Intel i7-8550U and 16GB of RAM. PRISM is run using the default settings.

4.2 Experimental Scenarios

4.2.1 Test MDPs

We consider four types of MDPs: *Gridworld*, *Dining Philosophers*, *Nim*, and *Double Counter*. In Gridworld, an agent is given some goal as an LTL$_f$ formula and must maximize the probability of satisfaction. The agent has four actions: North, South, East, and West. Under each action, the probability of moving to the cell in the intended direction is 0.69, then 0.01 for its opposite cell, and finally 0.1 for each of the other directions and for remaining in the same cell. If the resulting movement would place the agent in collision with the boundary, the agent remains in its current cell. We model the motion of this agent as an MDP, where the states correspond to the cells of the grid and the set of actions is $\{a_{\text{north}}, a_{\text{south}}, a_{\text{east}}, a_{\text{west}}\}$.

We base the Dining-Philosophers domain on the tutorial from the PRISM website. We consider a ring of five philosophers with two different specifications. Note that typical Dining-Philosophers specifications of interest are infinite-trace; our tests are merely meant to show that our approach works on MDPs other than Gridworld. Both *Nim* and *Double Counter* are probabilistic versions of games presented in [19].

4.2.2 Test formulas

Unfortunately, there is no standard set of LTL$_f$ formulas that we can use for benchmarking. In [25], random LTL formulas are used as the basis of LTL$_f$ benchmarks; however, because we also consider an MDP, random formulas are frequently tautologies or non-realizable with respect to the MDP. For instance, consider a randomly generated formula $\phi = p_1 U p_2$, where p_1 and p_2 are randomly assigned to the labels of the states in the Gridworld MDP. There is likely no path of the MDP that can satisfy this formula. As a matter of fact, in a test of more than 70 randomly generated LTL$_f$ formulas and Gridworld MDPs, only one was “interesting” (yielding probability between zero and one) on the corresponding MDP. For benchmarking, the interest is in sets of formulas that make sense in a probabilistic setting.

Therefore, as the first set of test formulas for the *Gridworld* MDP, we considered a natural finite-trace specification where the agent has n goals to accomplish in any order, as well as a “safety” property where it must globally avoid some states. This is typical of e.g., a robotics domain [13]. We use **Fn** to denote this formula for a given n. For the second set of test formulas, we keep regions to visit and avoid but introduce some ordering and repeat visits. **OS** is a short formula introducing order and **OL** is a longer formula that also contains nested temporal operators. All formulas are given in the appendix.

For the *Dining Philosophers* domains, typical examples focus on infinite-run properties. We test several finite-run properties on these domains. These properties are meant to illustrate our tool on an MDP other than Gridworld, but are not representative of typical finite-trace properties. One version (**D5**) asserts that all philosophers must eat at least once. The other (**D5C**) is a more complex property involving orders of eating. Both formulas are available in the appendix of the online version [24].

For *Nim*, the game of Nim is played against a stochastic opponent. To increase difficulty, the specifications require that randomly chosen stack heights are to be reached or avoided by the system player. For *Double Counter*, two four-bit binary counters are used. One counter is controlled by a stochastic environment, and the other is controlled by the system. The aim of the system is to make its counter match that of the environment.

4.3 Experimental Results

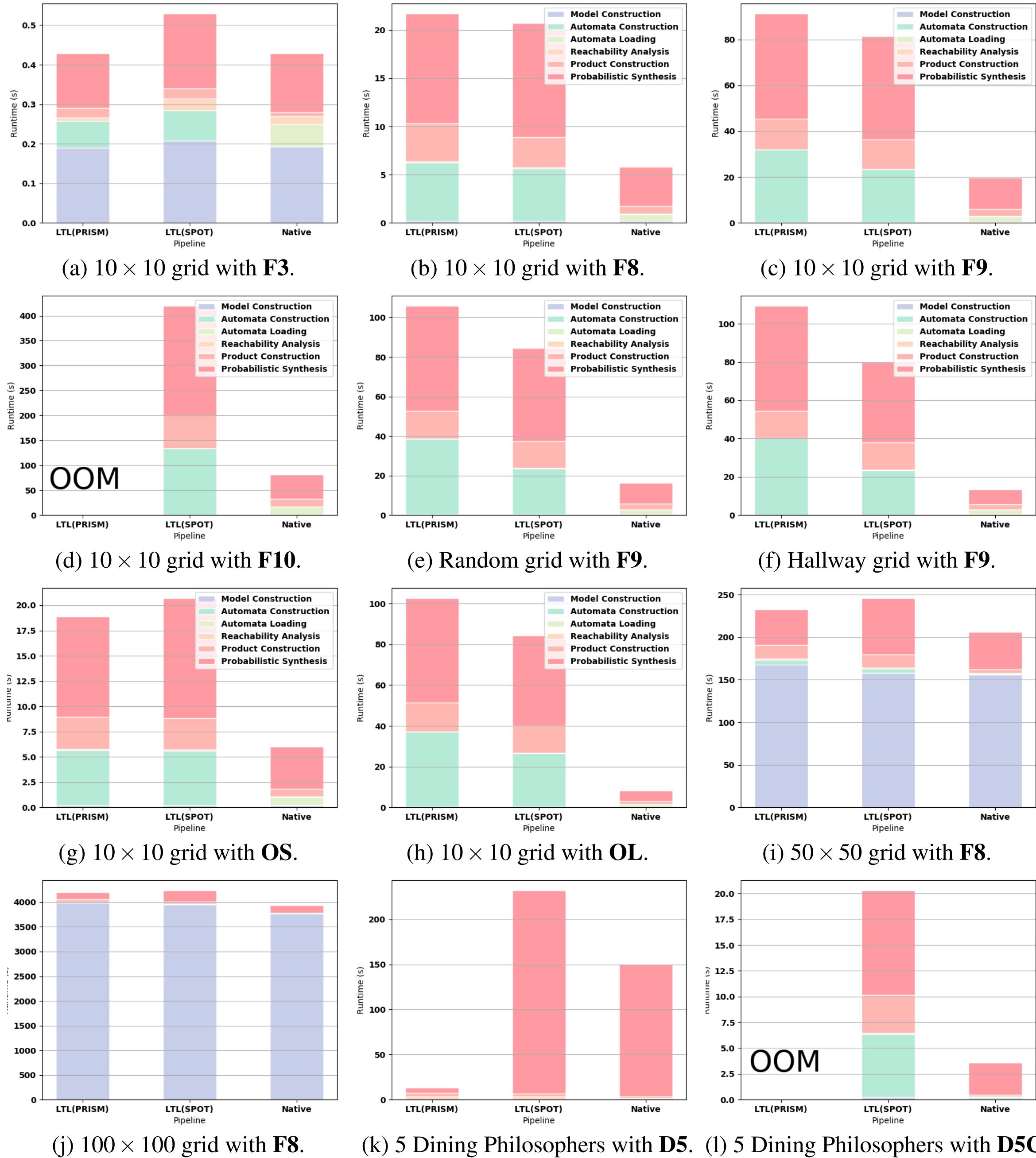
We provide experiments varying the complexity of the formulas and of the MDPs. Total runtime is shown in seconds. We refer to the translation to LTL as the “LTL pipeline” and the translation to a DFA via MONA as the “Native pipeline.” All plots are best viewed in color.

4.3.1 Automata Construction

First we measure how the length of the formula affects the synthesis computation time. For all of these experiments, we use a 10×10 Gridworld as the original MDP. For small formulas, the time needed to read the HOA file from hard disk outweighs the shorter construction time and smaller automata of the native approach as shown in Figure 3a. For formulas longer than **F3**, the native approach offers better computation time (e.g., see Figure 3b and Figure 3c).

PRISM successfully builds automata for formulas up to **F9**. In Figure 3d, we highlight the superiority of the native approach in constructing the automaton for **F10**. For this formula, PRISM runs out of memory when constructing the automaton according to the LTL pipeline. SPOT completes for **F10** but runs out of memory on **F11**. MONA works for formulas up to **F17**, which takes 8.586 seconds to compute, though writing the resulting file to disk is prohibitively expensive (the file is larger than 10GB). MONA runs out of memory constructing the automaton for **F18**.

We also considered a Gridworld with random obstacles (Figure 3e) and with hallways (Figure 3f), both with the formula **F9**. Finally, we consider two other formulas (**OS** and **OL**) in Figure 3g and Figure 3h to



(a) 10×10 grid with **F3**. (b) 10×10 grid with **F8**. (c) 10×10 grid with **F9**.

(d) 10×10 grid with **F10**. (e) Random grid with **F9**. (f) Hallway grid with **F9**.

(g) 10×10 grid with **OS**. (h) 10×10 grid with **OL**. (i) 50×50 grid with **F8**.

(j) 100×100 grid with **F8**. (k) 5 Dining Philosophers with **D5**. (l) 5 Dining Philosophers with **D5C**.

Figure 3: Runtime results for the Gridworld and Dining Philosophers MDPs with the LTL (PRISM and SPOT) and native pipelines.

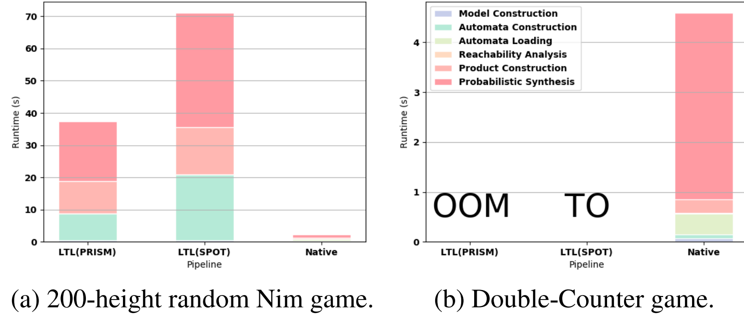
demonstrate that our improvement is not tied to the specific form of the specification.

For the Dining-Philosophers example, we consider five philosophers. Again, our results show that the native pipeline is faster and more memory efficient than the other approaches for automata construction (Figure 3l). However, the Dining-Philosophers example illustrates an issue where PRISM's built-in automata translation sometimes constructs automata such that computing the maximal accepting end-component using BDDs within PRISM is significantly faster (Figure 3k). The automata construction is

still slower than both SPOT and our native approach, and the automata returned have more states, but the BDD representation is more efficient. This issue affects some automata, not only for our tool, but also for other external tools we tested. This means not only automata construction and size, but also the BDD representation are important for overall runtime. However, PRISM's built-in translator's memory usage scales more poorly than SPOT or our native approach. On a more complex formula Figure 3l, PRISM's built-in construction runs out of memory, even though SPOT and the native approach can finish construction in less than ten seconds. Thus, for sufficiently complex formulas, PRISM's built-in translation is not viable. This suggests a need for automata construction methods that are not only more efficient but also produce automata that work well within PRISM.

In the Nim game, the native approach far outperforms both Prism and Prism with SPOT (Figure 4a). In the Double-Counter game, Prism runs out of memory and Prism with SPOT times out after more than 3 hours, whereas the native approach completes synthesis is less than 5 seconds.

Overall we observe that for large formulas the native pipeline offers significantly better scalability than the LTL pipeline (tested with PRISM's built-in translator, SPOT, Rabinizer, and LTL3DRA automata translators). With an implementation that does not access the hard disk, we expect even better performance of the native pipeline.



(a) 200-height random Nim game. (b) Double-Counter game.

Figure 4: Runtime results for the Nim and Double-Counter games.

4.3.2 Automata Size

The DFA generated by the native pipeline is minimal and typically much smaller than the LTL pipeline's DRA. SPOT and PRISM typically produce similarly sized DRAs. Because we take the product of the input MDP with these automata, we expect the size of the resulting product to be smaller in the native pipeline. Table 1 shows the automaton sizes for the various formulas. Table 2 shows the sizes of the product MDPs for the LTL and native pipelines respectively.

Interestingly, while the size of the DFA obtained from MONA is roughly half the size of the DRA constructed by PRISM, the final sizes of the products of the MDP and the automata are comparable for both approaches. The number of states, transitions and nondeterministic choices are all measured after reachability analysis is performed. We see time savings in the reachability analysis and product construction phases (both about two times faster), but the final products are similar in size. However, the product from the native pipeline has fewer nondeterministic transitions than the product from the LTL pipeline. The improvements from this are small relative to the time it takes to construct a large MDP.

Examples of this are shown in Figure 3i and Figure 3j. Note that the majority of the computation time is spent constructing the MDPs. For the LTL pipeline, this construction takes slightly longer time because of the augmented MDP. It is important to note that the native approach allows us to use larger formulas with these models whereas the LTL pipeline is limited to **F10** or smaller.

In summary, we observe that the native pipeline is the most efficient in terms of both runtime and memory. There are two possible drawbacks to the native pipeline. First, it requires reading from disk, which for small formulas negates the advantage in automata construction time. Second, PRISM's built-in automata translation sometimes constructs automata whose BDD representations work better within PRISM than automata from any external tools we tested.

Table 1: Sizes of the automata from LTL (DRA) and native (DFA) pipelines.

	F3	**F8**	**F9**	**F10**	**OS**	**OL**	**D5**	**D5C**	**Nim**	**Counter**
DRA states	19	512	1,027	NA	515	1,028	67	NA	701	NA
DFA states	10	258	514	1,026	258	259	33	29	67	130

Table 2: Sizes of the product MDP in the LTL pipeline and native pipelines.

	10x10 **F3**	10x10 **F8**	10x10 **F9**	50x50 **F8**	100x100 **F8**	10x10 **F10**	10x10 **OS**
States (LTL)	890	24,678	48,998	641,478	2,568,978	NA	24,678
States (Native)	881	24,421	48,485	641,221	2,568,721	96357	24,421
Transitions (LTL)	17,130	475,030	942,742	13,265,398	53,537,298	NA	475,030
Transitions (Native)	16,256	450,468	893,760	12,623,936	50,968,336	1775424	450,368
Choices (LTL)	4,410	122,358	242,934	3,206,358	12,843,858	NA	122,358
Choices (Native)	3,524	97,684	193,940	2,564,884	10,274,884	385428	97,684

	OL	10x10 rand	10x10 halls	5 Phil **D5**	5 Phil **D5C**	Nim	Counter
States (LTL)	24,935	44,253	19,476	4,548,220	NA	2115	NA
States (Native)	24,519	43,740	19,187	1,476,976	93,068	404	449
Transitions (LTL)	475,287	804,097	310,996	46,948,710	NA	8343	NA
Transitions (Native)	452,172	759,856	291,536	7,891,750	494,420	1205	908
Choices (LTL)	122,615	219,209	96,220	44,059,330	NA	6297	NA
Choices (Native)	98,076	174,960	76,748	6,895,580	437,050	808	460

5 Conclusion

We introduced the problem of LTL_f synthesis for probabilistic systems and presented two approaches. The first one is a reduction of LTL_f to LTL with a corresponding augmentation of the MDP. The second approach uses native tools to construct an automaton and then takes the product of this automaton with the MDP to construct a product MDP that can be used for synthesis through standard techniques. We showed that this native approach offers better scalability than the reduction to LTL. Our work opens the door to the use of LTL_f synthesis on practical domains such as robotics, cf. [13]. Our tool is on GitHub [23].

For future work we would like to expand our results to include probability minimization (c.f. Theorem 3). Our native approach extends easily, but there are subtleties that prevent applying the LTL pipeline to minimization. We are also interested in a fully symbolic methodology, where the automaton is represented symbolically and the product is also taken symbolically. We would like to further investigate the issue we discovered where external tools quickly find automata with fewer states than PRISM's built-in method, but for which computing maximal accepting end-components in the product MDP takes much longer. We are also interested in applying the work to robotics domains.

References

[1] Tomáš Babiak, František Blahoudek, Mojmír Křetínskỳ & Jan Strejček (2013): *Effective translation of LTL to deterministic Rabin automata: Beyond the (F, G)-fragment*. In: Automated Technology for Verification and Analysis, Springer, pp. 24–39.

[2] Tomáš Babiak et al. (2015): *The Hanoi Omega-Automata Format*. In Daniel Kroening & Corina S. Păsăreanu, editors: Computer Aided Verification, Springer Intl. Publishing, Cham, pp. 479–486.

[3] Fahiem Bacchus, Craig Boutilier & Adam Grove (1996): *Rewarding Behaviors*. In: Proc. of the Thirteenth Natl. Conf. on Artificial Intelligence - Volume 2, AAAI96, AAAI Press, p. 11601167.

[4] Fahiem Bacchus, Craig Boutilier & Adam Grove (1997): *Structured Solution Methods for Non-Markovian Decision Processes*. In: Proc. of the (AAAI-97) and (IAAI-97), AAAI97/IAAI97, AAAI Press, p. 112117.

[5] Christel Baier & Joost-Pieter Katoen (2008): *Principles of Model Checking*. The MIT Press.

[6] Ronen I. Brafman, Giuseppe De Giacomo & Fabio Patrizi (2018): *LTLf/LDLf Non-Markovian Rewards*. In Sheila A. McIlraith & Kilian Q. Weinberger, editors: *Proc. of (AAAI-18), (IAAI-18), and (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, AAAI Press, pp. 1771–1778.

[7] Alberto Camacho, Oscar Chen, Scott Sanner & Sheila A. McIlraith (2017): *Non-Markovian Rewards Expressed in LTL: Guiding Search Via Reward Shaping*. In: *SOCS*.

[8] Luca De Alfaro (1997): *Formal verification of probabilistic systems*. 1601, Citeseer.

[9] Giuseppe De Giacomo & Moshe Y Vardi (2013): *Linear Temporal Logic and Linear Dynamic Logic on Finite Traces*. In: *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, 13, pp. 854–860.

[10] Giuseppe De Giacomo & Moshe Y Vardi (2015): *Synthesis for LTL and LDL on Finite Traces*. In: *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, 15, pp. 1558–1564.

[11] Alexandre Duret-Lutz et al. (2016): *Spot 2.0 — a framework for LTL and ω-automata manipulation*. In: *Proc. of the 14th Intl. Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, *Lecture Notes in Computer Science* 9938, Springer, pp. 122–129.

[12] Keliang He, Morteza Lahijanian, Lydia E Kavraki & Moshe Y Vardi (2018): *Automated Abstraction of Manipulation Domains for Cost-Based Reactive Synthesis*. *IEEE Robotics and Automation Letters* 4(2), pp. 285–292.

[13] Keliang He, Andrew M Wells, Lydia E Kavraki & Moshe Y Vardi (2019): *Efficient Symbolic Reactive Synthesis for Finite-Horizon Tasks*. In: *2019 Intl. Conf. on Robotics and Automation (ICRA)*, IEEE, pp. 8993–8999.

[14] Keling He, Morteza Lahijanian, Lydia E. Kavraki & Moshe Y. Vardi (2017): *Reactive Synthesis for Finite Tasks Under Resource Constraints*. In: *Int. Conf. on Intelligent Robots and Systems (IROS)*, IEEE, Vancouver, BC, Canada, pp. 5326–5332.

[15] Jesper G Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe & Anders Sandholm (1995): *Mona: Monadic second-order logic in practice*. In: *Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 89–110.

[16] Zuzana Komárková & Jan Křetínský (2014): *Rabinizer 3: Safraless Translation of LTL to Small Deterministic Automata*. In Franck Cassez & Jean-François Raskin, editors: *Automated Technology for Verification and Analysis*, Springer Intl. Publishing, Cham, pp. 235–241.

[17] M. Kwiatkowska, G. Norman & D. Parker (2011): *PRISM 4.0: Verification of Probabilistic Real-time Systems*. In G. Gopalakrishnan & S. Qadeer, editors: *Proc. 23rd Intl. Conf. on Computer Aided Verification (CAV'11)*, *LNCS* 6806, Springer, pp. 585–591.

[18] Jianwen Li, Kristin Y Rozier, Geguang Pu, Yueling Zhang & Moshe Y Vardi (2019): *SAT-Based Explicit LTLf Satisfiability Checking*. In: *Proc. of the AAAI Conf. on Artificial Intelligence*, 33, pp. 2946–2953.

[19] Lucas Martinelli Tabajara & Moshe Y. Vardi (2019): *Partitioning Techniques in LTLf Synthesis*. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, International Joint Conferences on Artificial Intelligence Organization, pp. 5599–5606, doi:10.24963/ijcai.2019/777. Available at `https://doi.org/10.24963/ijcai.2019/777`.

[20] Amir Pnueli (1977): *The temporal logic of programs*. In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*, IEEE, pp. 46–57.

[21] Sylvie Thiébaux, Charles Gretton, John Slaney, David Price & Froduald Kabanza (2006): *Decision-Theoretic Planning with Non-Markovian Rewards*. *J. Artif. Int. Res.* 25(1), p. 1774.

[22] M.Y. Vardi (1985): *Automatic Verification of Probabilistic Concurrent Finite-State Programs*. pp. 327–338.

[23] Andrew M. Wells: *LTLf for PRISM*. Available at `https://github.com/andrewmw94/ltlf_prism`.

[24] Andrew M Wells, Morteza Lahijanian, Lydia E Kavraki & Moshe Y Vardi: *LTLf Synthesis on Probabilistic Systems (Online version)*. Available at `http://www.andrewmwells.com/ltlf-synthesis-on-probabilistic-systems/`.

[25] Shufang Zhu, Lucas M Tabajara, Jianwen Li, Geguang Pu & Moshe Y Vardi (2017): *Symbolic LTLf synthesis*. In: *Proc. of the 26th Intl. Joint Conf. on Artificial Intelligence*, AAAI Press, pp. 1362–1369.

F

Proof of lemma 2.

Proof. We prove by contradiction. Recall that $\mathcal{M}^{\pi}$ and $\mathcal{M}'^{\pi'}$ are both DTMCs, and the probability measure is defined over infinite paths using cylinder sets in 5. Assume to the contrary that

$$\max_{\pi\in\Pi} Pr(\mathcal{M}^{\pi} \models Paths_{fin,\phi}) > \max_{\pi'\in\Pi'} Pr(\mathcal{M}'^{\pi'} \models Paths_{fin,\phi})$$

Then for an optimal policy π^* all paths $w = s_0 \dots s_n \in Paths_{fin,\phi}$ we have probability measure given by the cylinder set $Pr(Cyl(s_0 \dots s_n)) = \prod_{0\leq i<n} P(s_i, \pi^*(s_i), s_{i+1})$. But because of (6), we know that there exists an equivalent policy π'^* (with paths w') where the actions can be chosen such that the probabilities are the same $\pi'^*(w'_{fin}) = \pi^*(w_{fin})$ if $w_{fin} \notin Paths_{fin,\phi} \wedge last(w_{fin}) \neq s_{\text{term}}$. Thus the probability measure from the cylinder sets of these paths will be equivalent. Furthermore the probability for a_{term} is always 1, which does not change the product. Thus there must be a π'^* yielding the same probability. This presents a contradiction. So:

$$\max_{\pi\in\Pi} Pr(\mathcal{M}^{\pi} \models Paths_{fin,\phi}) \leq \max_{\pi'\in\Pi'} Pr(\mathcal{M}'^{\pi'} \models Paths_{fin,\phi})$$

The second case is analogous. Assume to the contrary that

$$\max_{\pi\in\Pi} Pr(\mathcal{M}^{\pi} \models Paths_{fin,\phi}) < \max_{\pi'\in\Pi'} Pr(\mathcal{M}'^{\pi'} \models Paths_{fin,\phi})$$

Then for an optimal policy π'^* all paths $w' = s_0 \dots s_n \in Paths_{fin,\phi}$ we have probability measure given by the cylinder set $Pr(Cyl(s_0 \dots s_n)) = \prod_{0\leq i<n} P(s_i, \pi'^*(s_i), s_{i+1})$. But because of (6), we know that there exists an equivalent policy π^* (with paths w) where the actions can be chosen such that the probabilities are the same $\pi^*(w_{fin}) = \pi'^*(w'_{fin})$ if $w_{fin} \notin Paths_{fin,\phi} \wedge last(w_{fin}) \neq s_{\text{term}}$. And $last(w_{fin}) = s_{\text{term}}$ only after a_{term} has been taken and no other actions are available in s_{term}. The probability for a_{term} is always 1, which does not change the product. Thus the probability measure from the cylinder sets of these paths will be equivalent. Thus there must be a π^* yielding the same probability. This presents a contradiction. So:

$$\max_{\pi\in\Pi} Pr(\mathcal{M}^{\pi} \models Paths_{fin,\phi}) \geq \max_{\pi'\in\Pi'} Pr(\mathcal{M}'^{\pi'} \models Paths_{fin,\phi})$$

□ □

Proof of Theorem 3

Proof. This follows through an application of Lemma 2 and a proof that the finite paths on $\mathcal{M}$ satisfying ϕ correspond to the infinite paths on on $\mathcal{M}'$ satisfying $\varphi = g(\phi)$. We consider both directions:

Let w be a finite path satisfying ϕ. Then from [9], it follows that paths satisfying ϕ will also satisfy φ assuming the *alive* proposition of φ is set precisely when the path w ends. (5) ensures that this is the case.

Let w' be an infinite path satisfying φ. Then from [9], it follows that paths satisfying φ will also satisfy ϕ assuming the *alive* proposition of φ is set to symbolize an end of the execution of w' on $\mathcal{M}'$. (5) ensures that this is the case.

□ □

G

We use the following LTL$_f$ formulae:

F3:

```
(F "loca") & (G !"bad")
```

F8:

```
(F "loca") & (F "locb") & (F "locc") & (F "locd")
 & (F "locaa") & (F "locab") & (F "locac")
  & (F "locad") & (G !"bad")
```

F9:

```
(F "loca") & (F "locb") & (F "locc") & (F "locd")
 & (F "loce") & (F "locf") & (F "locg")
  & (F "loch") & (F "loci") & (G !"bad")
```

F10:

```
(F "loca") & (F "locb") & (F "locc") & (F "locd")
 & (F "loce") & (F "locf") & (F "locg")
  & (F "loch") & (F "loci") & (F "locj") & (G !"bad")
```

OS:

```
(F "loca") & (F "locb" & (F "locc")) & (F "locc" & (F "locb"))
 & (F "locd") & (F "loce") & (F "locf")
  & (F "locg") & (F "loch") & (G !"zbad")
```

OL:

```
(F "loca") & (F "locb" & (F "locc")) & (F "locc" & (F "locb"))
 & (F "locd" & (F "loce")) & (F "loce" & (F "locd"))
  & (F "locf" & (F "locg"))  & (F "locg" & (F "locf"))
   & (F "loch") & (G F "loci") & (G !"zbad")
```

D5:

```
(F "eat1") & (F "eat2") & (F "eat3") & (F "eat4") & (F "eat5")
```

D5C:

```
((F "eat1") & (F "eat2")) |
((((("eat1" U "eat2") U "eat3") U "eat4") U "eat5") |
("eat1" U ("eat2" U ("eat3" U ("eat4" U "eat5")))))
```

Nim:

```
(F(G !"robotwin")) & (F("x46")) | (X (!("x120"))) &
(F("x6")) | (F("x165")) | (G(!("x111"))) &
(F("x127")) & (X (!("x12"))) | (X (!("x36"))) &
(G(!("x102")))
```

Double Counter:

```
(G((!( (X !"p0") & (X !"p1") & (X !"p2") & (X !"p3") ) |
( ((!"p0") & (!"p1") & (!"p2") & (!"p3") ) | ((!"p0") &
(!"p1") & (!"p2") & ("p3") ))) & (!( (X !"p0") & (X !"p1") &
(X !"p2") & (X "p3") ) | ( ((!"p0") & (!"p1") & (!"p2") & ("p3") ) |
((!"p0") & (!"p1") & ("p2") & (!"p3") ))) & (!( (X !"p0") & (X !"p1") &
(X "p2") & (X !"p3") ) | ( ((!"p0") & (!"p1") & ("p2") & (!"p3") ) |
((!"p0") & (!"p1") & ("p2") & ("p3") ))) & (!( (X !"p0") & (X !"p1") &
(X "p2") & (X "p3") ) | ( ((!"p0") & (!"p1") & ("p2") & ("p3") ) |
((!"p0") & ("p1") & (!"p2") & (!"p3") ))) & (!( (X !"p0") & (X "p1") &
(X !"p2") & (X !"p3") ) | ( ((!"p0") & ("p1") & (!"p2") & (!"p3") ) |
((!"p0") & ("p1") & (!"p2") & ("p3") ))) & (!( (X !"p0") & (X "p1") &
(X !"p2") & (X "p3") ) | ( ((!"p0") & ("p1") & (!"p2") & ("p3") ) |
((!"p0") & ("p1") & ("p2") & (!"p3") ))) & (!( (X !"p0") & (X "p1") &
(X "p2") & (X !"p3") ) | ( ((!"p0") & ("p1") & ("p2") & (!"p3") ) |
((!"p0") & ("p1") & ("p2") & ("p3") ))) & (!( (X !"p0") & (X "p1") &
(X "p2") & (X "p3") ) | ( ((!"p0") & ("p1") & ("p2") & ("p3") ) |
(("p0") & (!"p1") & (!"p2") & (!"p3") ))) & (!( (X "p0") & (X !"p1") &
(X !"p2") & (X !"p3") ) | ( (("p0") & (!"p1") & (!"p2") & (!"p3") ) |
(("p0") & (!"p1") & (!"p2") & ("p3") ))) & (!( (X "p0") & (X !"p1") &
(X !"p2") & (X "p3") ) | ( (("p0") & (!"p1") & (!"p2") & ("p3") ) |
(("p0") & (!"p1") & ("p2") & (!"p3") ))) & (!( (X "p0") & (X !"p1") &
(X "p2") & (X !"p3") ) | ( (("p0") & (!"p1") & ("p2") & (!"p3") ) |
(("p0") & (!"p1") & ("p2") & ("p3") ))) & (!( (X "p0") & (X !"p1") &
(X "p2") & (X "p3") ) | ( (("p0") & (!"p1") & ("p2") & ("p3") ) |
(("p0") & ("p1") & (!"p2") & (!"p3") ))) & (!( (X "p0") & (X "p1") &
(X !"p2") & (X !"p3") ) | ( (("p0") & ("p1") & (!"p2") & (!"p3") ) |
(("p0") & ("p1") & (!"p2") & ("p3") ))) & (!( (X "p0") & (X "p1") &
(X !"p2") & (X "p3") ) | ( (("p0") & ("p1") & (!"p2") & ("p3") ) |
(("p0") & ("p1") & ("p2") & (!"p3") ))) & (!( (X "p0") & (X "p1") &
(X "p2") & (X !"p3") ) | ( (("p0") & ("p1") & ("p2") & (!"p3") ) |
(("p0") & ("p1") & ("p2") & ("p3") ))) & (!( (X "p0") & (X "p1") &
(X "p2") & (X "p3") ) | ( (("p0") & ("p1") & ("p2") & ("p3") ) |
((!"p0") & (!"p1") & (!"p2") & (!"p3") ))))) &
(F("robotturnwin" | "humanturnwin"))
```