Fuel in Markov Decision Processes (FiMDP): A Practical Approach to Consumption

František Blahoudek[1], Murat Cubuktepe[2], Petr Novotný[3], Melkior Ornik[4], Pranay Thangeda[4(✉)], and Ufuk Topcu[2]

[1] Brno University of Technology, Brno, Czech Republic
[2] The University of Texas at Austin, Austin, USA
{mcubuktepe,utopcu}@utexas.edu
[3] Masaryk University, Brno, Czech Republic
petr.novotny@fi.muni.cz
[4] University of Illinois Urbana-Champaign, Urbana, USA
{mornik,pranayt2}@illinois.edu

Abstract. Consumption Markov Decision Processes (CMDPs) are probabilistic decision-making models of resource-constrained systems. We introduce FiMDP, a tool for controller synthesis in CMDPs with LTL objectives expressible by deterministic Büchi automata. The tool implements the recent algorithm for polynomial-time controller synthesis in CMDPs, but extends it with many additional features. On the conceptual level, the tool implements heuristics for improving the expected reachability times of accepting states, and a support for multi-agent task allocation. On the practical level, the tool offers (among other features) a new strategy simulation framework, integration with the Storm model checker, and FiMDPEnv - a new set of CMDPs that model real-world resource-constrained systems. We also present an evaluation of FiMDP on these real-world scenarios.

1 Introduction

Planning the motion of an agent operating in a stochastic environment described by a discrete-time *Markov decision process* (MDP) is a classical problem of stochastic control [3,20]. In each step, the agent selects an action to perform, the outcome of which is given by a probability distribution over successor states depending on the current state and the selected action. Such a framework finds immediate use in planning for robots and autonomous vehicles operating on complex terrain, as well as operations on a financial market [7,12,16].

This work was partially supported by NASA under Early Stage Innovations grant No. 80NSSC19K0209, and by DARPA under grant No. HR001120C0065. Petr Novotný was supported by the Czech Science Foundation grant No. GJ19-15134Y and František Blahoudek was supported by the Czech Ministry of Education, Youth and Sports ERC CZ project LL1908.

M. Huisman et al. (Eds.): FM 2021, LNCS 13047, pp. 640–656, 2021.
https://doi.org/10.1007/978-3-030-90870-6_34

Constructing control strategies that enable the agent to reach a particular target state with maximum probability has been the focus of substantial previous work on MDPs [4,14,22]. Moving forward from simple reachability tasks, linear temporal logic (LTL) [19] serves as a convenient way to describe a wide class of missions for such an agent [2,11]. LTL specifications can describe the desired spatiotemporal features of the agent's path like "visit area A before visiting either area B or C, while never visiting area D". The synthesis of optimal control strategies for such specifications has also been studied previous work (e.g. [2,22]); a standard method is to monitor the agent's progress at its mission by a deterministic automaton, thus posing mission success as producing an accepting path in the synchronous product of the automaton and the MDP.

The motivation for this paper is to practically enable planning for resource-constrained agents. Such agents maintain a limited amount of some critical resource (e.g. energy needed to operate) at every time step. The resource is consumed in each time step (with the amount depending on the current state and action) and can be replenished at particular *reload states*. The agent's specification is to satisfy its mission while retaining a positive amount of the resource throughout its operation.

A naive planning approach for resource-constrained agents is (i) to append the resource level as a part of the agent's state and (ii) to enrich the mission specification defined on the agent's states by "always $resource > 0$". This method multiplies the size of the agent's state space by the number of the possible resource levels which significantly adds to the computational burden of planning. (In particular, it leads to an *exponential* time complexity if the agent's resource capacity is specified in binary).

Our recent paper [5] introduces a formalism of *consumption Markov decision processes* which avoids encoding the resource levels into states. Instead, the consumption of the resource is defined as a special cost. It is then up to the agents to monitor their own resource levels and choose actions also with respect to their current remaining amount of the resource. This is exactly what *counter strategies* introduced in the same paper do. They use counters to monitor the resource level and *action selectors* to choose actions to play in a state depending on the resource level. As a result, an agent can exhibit two or more behaviors in one state. For instance, the agent may elect to go to a reload state when the resource level is too low to complete the mission without reloading, while it may proceed with the mission without reloading when it has a sufficient resource level. The decision does not have to be specified for each resource level, but rather for *intervals* of resource levels. Using this insight, [5] proves that counter strategies with polynomial-sized representation are sufficient for guaranteeing mission success with probability 1, and that the associated qualitative synthesis problem can be solved in *polynomial* time. Hence, consumption MDPs and counter strategies provably quicken the qualitative controller synthesis for resource-constrained agents.

The algorithms presented in [5] were implemented in a prototype implementation. After more than a year of additional development, in this paper we

introduce FiMDP (Fuel in Markov Decision Processes): a tool for design and analysis of consumption MDPs, and for synthesis of resource-aware control strategies for them. We designed FiMDP with two goals in mind: practicality and education.

The practical purpose of the tool is to provide high-quality results for interesting planning problems in reasonable time. Therefore, we have extended the previous work by (i) constructing novel heuristics that produce strategies that not only satisfy the mission almost surely, but also attempt to complete the mission in a short time, (ii) enabling support of missions specified by deterministic Büchi automata or by the corresponding fragment of linear temporal logic, (iii) enabling an interface with well-established modeling languages like PRISM [17] and JANI [8], and (iv) given a set of targets and homogeneous agents, providing a polynomial-time algorithm that computes a target allocation and assignment to minimize the resource capacity of each agent. We also show that the tool translates the theoretical gains of the new formalism to practice: FiMDP provides the ability to synthesize resource-aware strategies for significantly larger state spaces than the state-of-the-art model checker Storm [15].

To be educational, the tool makes the framework of consumption MDPs accessible by (i) having a simple user interface, (ii) possessing an ability to read models in existing modeling languages, (iii) constructing visual representations of the models and results, and (iv) supporting simulations of the synthesized strategies. These components help the users to understand the key concepts of the novel formalism without being exposed to the entire technical machinery behind the framework.

FiMDP comes bundled with FiMDPEnv: a set of environments inspired by real world that are built on top of consumption MDPs. FiMDPEnv interfaces with FiMDP for strategy synthesis, and simulates the strategies in a visually vivid manner. This feature allows the user to easily modify the high-level parameters of the mission at hand, such as the mission specification, as necessary.

FiMDP is available at https://github.com/FiMDP/FiMDP. FiMDPEnv is available at https://github.com/FiMDP/FiMDPEnv. We created a series of tutorials available at https://github.com/FiMDP/tutorials to get started with the tool. These tutorials, presented in interactive Jupyter notebooks, are designed to provide the user with an in-depth understanding of the key features offered by the tool.

The next section defines consumption MDPs and the necessary notation. Section 3 summarizes the features of FiMDP and explains the novel functionality not available before. More details about FiMDP follow in Sect. 4 and Sect. 5 presents FiMDPEnv. Finally, Sect. 6 empirically compares strategy synthesis for consumption MDPs in FiMDP and Storm, and also shows the effect of the novel heuristics on the quality of the synthesized control strategies.

2 Consumption Markov Decision Processes

Notation. The set of all non-negative integers is denoted by $\mathbb{N}$. If $\mathcal{X}$ is a set, the set of all infinite sequences of elements in $\mathcal{X}$ is denoted by $\mathcal{X}^\omega$. The set of

all subsets of $\mathcal{X}$ is denoted by $2^{\mathcal{X}}$. Notation $\mathbb{P}(X)$ denotes the probability of an event X in an appropriate event space.

Throughout this section, we follow the framework introduced in [5]. In order to model the agent's resource consumption and capacity, we first amend the definition of a standard Markov decision process [20] by defining a *consumption Markov decision process*.

Definition 1 (CMDP). *A* consumption Markov decision process *(CMDP) is a tuple* $\mathcal{M} = (S, A, P, C, R, cap)$ *where:*

- S *is a finite set of states,*
- A *is a finite set of actions,*
- $P : S \times A \times S \to [0, 1]$ *is a* transition probability function *which satisfies* $\sum_{s' \in S} P(s, a, s') = 1$ *for all* $s \in S$, $a \in A$,
- $C : S \times A \to \mathbb{N}$ *is a* consumption function,
- $R \subseteq S$ *is a set of* reload states, *and*
- $cap \in \mathbb{N}$ *is a* resource capacity.

The CMDP $\mathcal{M}$ evolves in discrete time steps. In each time step $t \in \mathbb{N}$, the agent's state is denoted by s_t and its *resource level* by r_t. The agent chooses an action a_t with consumption not higher than r_t. Based on a_t, the agent transitions to state s_{t+1} that is chosen randomly using the transition probability function P: state s_{t+1} is a random variable such that $\mathbb{P}(s_{t+1} = s' | s_t, a_t) = P(s_t, a_t, s')$. The dynamics of the resource level are given by the following equation.

$$r_{t+1} = \begin{cases} cap & \text{if } s_{t+1} \in R \\ r_t - C(s_t, a_t) & \text{otherwise} \end{cases} \tag{1}$$

In other words, when an agent reaches a reload state, its resource is reloaded to the full capacity cap. Otherwise, the resource level is reduced by $C(s_t, a_t)$. This means that r_t is uniquely defined at every time t by the initial resource level r_0 and the history of states and actions.

We introduce the notion of a *path* as a finite or infinite state-action sequence

$$\Phi = s_0 a_0 s_1 a_1 s_2 a_2 s_3 \cdots \in (S \times A)^\omega \bigcup \left(\bigcup_{n \in \mathbb{N}} (S \cdot A)^n \right) \times S.$$

The length of a path Φ is ∞ if $\Phi \in (S \times A)^\omega$, and n if $\Phi \in (S \times A)^n \times S$. We call an infinite path a *run*, and a finite path a *history*.

A *strategy* for an agent operating on $\mathcal{M}$ is a function $\sigma : \mathcal{H}(\mathcal{M}) \to A$, where $\mathcal{H}(\mathcal{M})$ is the set of all histories on $\mathcal{M}$. In other words, an agent can decide its next action based on all previous states and actions, and thus, by (1), possibly also based on its resource level. *Positional strategies* (or *memoryless strategies*) known from regular MDPs base their decision solely on the current state. *Counter strategies* extend positional strategies by taking also the current resource level into consideration. A counter strategy consists of a counter that keeps track of

the resource level r_t, and an *action selector* that maps the current state and resource level into the action to play.

An action selector is a function $sel\colon S \times \{0, 1, \ldots, cap\} \to A$ which we effectively represent using intervals. In particular, for every state s we define values $0 = p_1^s < p_2^s < \ldots < p_{k_s}^s < cap + 1 = p_{k_s+1}^s$ and actions $a_1^s, \ldots, a_{k_s}^s \in A$. Then for given state s and resource level r the value $sel(s, r)$ is a_j^s if $p_j^s \leq r < p_{j+1}^s$.

A CMDP is *decreasing* if and only if each cycle either contains a reload state or has non-zero consumption. This means that the agent is forced to visit reload states in a decreasing CMDP. From now on, we only consider decreasing CMDPs.

Labeled Consumption Markov Decision Processes. In [5], we have only considered reachability or Büchi objectives defined explicitly using a set F of *accepting* or *target* states. To extend the range of possible mission specifications, we introduce a *labeling function* $L : S \to 2^{AP}$ which assigns to each state a set of *atomic propositions*. Atomic propositions are features relevant to the agent's objective. For instance, we can encode the set of accepting states F from above by setting $L(s) = \{F\}$ if $s \in F$ and define $L(s) = \emptyset$ otherwise.

Each path Φ naturally induces a sequence of labels $L(s_0)L(s_1)L(s_2)\ldots$. These sequences can be evaluated against specifications describing the agent's objective given as an automaton or as a formula of Linear temporal logic (LTL) [19]. We refer the reader to [2] for precise definitions. In this paper, we consider properties that can be expressed via *deterministic Büchi automata (DBA)* or via formulas from the *recurrence* fragment of the Manna-Pnueli hierarchy for LTL [18]. This fragment is strong enough to express many useful properties like "keep visiting states labeled by F while never seeing label *exit* before label *landed*", and each recurrence formula can be translated into an equivalent DBA.

3 Features of FiMDP

In this section, we first briefly review the theoretical foundations of FiMDP, as introduced in [5]. We then present FiMDP's main features, focusing on the novel features that distinguish it from the prototype artifact of [5].

3.1 Theoretical Foundations

FiMDP is a tool for qualitative analysis and controller synthesis in CMDPs. As such, it builds on the polynomial-time synthesis algorithm for CMDPs presented in [5]. We are given a CMDP $\mathcal{M} = (S, A, P, C, R, cap)$ and a set of accepting states $F \subseteq S$. The task is to compute a strategy σ which is *safe* (i.e., ensures that the resource level never drops below zero) and almost surely satisfies the Büchi objective given by F. More precisely, for every initial state s, the algorithm computes the minimal initial resource level $\textit{min-lev}(s) \in \{0, \ldots, cap\} \cup \{\infty\}$ such that there exists a strategy σ ensuring the following properties:

a) for all $t \in \mathbb{N}$, $\mathbb{P}^\sigma(r_t < 0 \mid s_0 = s, r_0 = \textit{min-lev}(s)) = 0$; and
b) $\mathbb{P}^\sigma(s_t \in F \textit{ for infinitely many } t \mid s_0 = s, r_0 = \textit{min-lev}(s)) = 1$;

where $\mathbb{P}^\sigma$ is the probability measure induced by σ over the runs in $\mathcal{M}$. If $\textit{min-lev}(s) = \infty$, the objective is unachievable from s; we say that s is a *losing* state and the algorithm reports all such states. We call all the other states *winning*. The rationale behind the minimal resource levels is that due to the monotonicity inherent to the resource constraints, a strategy that satisfies a) and b) satisfies the same conditions for any initial resource level $r_0 \geq \textit{min-lev}(s)$. In [5], it was also proved that there exists a polynomial-sized counter strategy σ that satisfies both a) and b) in every winning state. The algorithm presented in [5] also computes such a strategy.

The computation of both the minimal resource levels and the corresponding strategy σ proceeds in two phases. In the first phase, the algorithm solves the *positive reachability* problem. The problem consists of computing the minimal resource levels $\textit{min-pos-reach}(s) \in \{0, \ldots, \textit{cap}\} \cup \{\infty\}$ and a strategy π which, given the initial resource level $\textit{min-pos-reach}(s)$, reaches an accepting state with a positive probability while preventing resource exhaustion:

a') for all $t \in \mathbb{N}$, $\mathbb{P}^\pi(r_t < 0 \mid s_0 = s, r_0 = \textit{min-pos-reach}(s)) = 0$; and
b') there exists $t \in \mathbb{N}$ s.t. $\mathbb{P}^\pi(s_t \in F \mid s_0 = s, r_0 = \textit{min-pos-reach}(s)) > 0$.

The corresponding algorithm iterates a suitable Bellman functional in a process analogous to value iteration. Note that the agent using π might eventually run in a situation where $r_t < \textit{min-pos-reach}(s_t)$.

The values *min-pos-reach* are the key for building the desired strategy σ out of π. Imagine a strategy that behaves like π and at the same time it keeps the resource level high enough so that r_t never drops below $\textit{min-pos-reach}(s_t)$. If that is possible, we have always a positive probability of reaching F (the property b')) and thus the agent will eventually reach F. And since even when reaching F at time t_F we have that $r_{t_F} \geq \textit{min-pos-reach}(s_{t_F})$, it will be reached again, and so on *ad infinitum*. Thus, the Büchi property b) is satisfied.

There is a small caveat in this approach: unlike π, the strategy σ needs to avoid states that are losing for the positive reachability problem at all costs. Thus it might differ from π and as such it might lose property b'). To solve this, we remove the losing states from the given CMDP, compute a new π and *min-pos-reach* in this reduced CMDP and repeat this until all states of the reduced CMDP are winning. In such case, π actually satisfies both b') and b) and we have our σ for the Büchi objective.

3.2 New Features

The algorithm that rests at the heart of FiMDP has polynomial complexity and its prototype scaled well in the somewhat lightweight experiments of [5]. FiMDP extends this algorithmic core with further heuristics and additional functionality that greatly enhances its practical capabilities. We present these new features in the remainder of this section.

Labeled CMDPs, Deterministic Büchi Automata and LTL. The algorithm in [5] synthesizes a resource-constrained strategy for almost-sure satisfaction of a Büchi objective, where the accepting states have to be explicitly

specified as a part of the MDP. FiMDP now fully supports CMDPs with states labeled by sets of atomic propositions. Given such a labeled CMDP $\mathcal{M}$ and a deterministic Büchi automaton $\mathcal{A}$ over the same atomic propositions, FiMDP can synthesize a resource-aware strategy that produces (for a suitable initial level of resource) runs whose labeling is almost surely accepted by $\mathcal{A}$. To achieve this, FiMDP follows the classical approach of constructing and analyzing the *product* $\mathcal{M} \otimes \mathcal{A}$ via the standard *synchronous parallel composition* technique [2].

Obviously, the counter strategy synthesized for $\mathcal{M} \otimes \mathcal{A}$ operates correctly only on this product, which might be impractical. For the convenience of users, FiMDP provides a capability to synthesize a counter strategy that stores $\mathcal{A}$ in its memory. This means, the strategy tracks in memory both the resource level (using a counter) and the progress in $\mathcal{A}$, and makes decisions based on the current state, the current resource level, and the current state of the automaton. As a result, the users can run these strategies on their original model. Another advantage of these strategies is that the potentially large object for $\mathcal{M} \otimes \mathcal{A}$ can be discarded from memory right after the analysis.

The product-based approach extends naturally to the recurrence fragment of linear temporal logic. Given a formula φ of this fragment and a labeled CMDP $\mathcal{M}$, FiMDP can compute the values $\mathit{min\text{-}lev}_\varphi$ and synthesize a strategy σ, which operates on $\mathcal{M}$ and which has the following properties:

a) for all $t \in \mathbb{N}$, $\mathbb{P}^\sigma(r_t < 0 \mid s_0 = s, r_0 = \mathit{min\text{-}lev}_\varphi(s)) = 0$; and
b) $\mathbb{P}^\sigma(\varphi \text{ is satisfied by the sample run} \mid s_0 = s, r_0 = \mathit{min\text{-}lev}_\varphi(s)) = 1$.

This capability is justified by the fact that each formula of the recurrence fragment can be translated to an equivalent DBA [18]. FiMDP uses the Spot library [10] for this translation and then employs the product approach described above. The user requires just one line of code with FiMDP to run this pipeline and get a strategy without ever being exposed to the product construction.

Strategies and Simulations Framework. The core algorithm for strategy synthesis does not compute the strategies (objects with memory) directly. It produces action selectors, the integral parts of counter strategies. The strategy simply asks the selector, what should be the next action given its current state of memory. As we saw in the labeled framework, altering the type of the strategy's memory (and of the selector) might result in more useful strategy representations. FiMDP provides an interface for objects representing strategies that completely hides the implementation details of the strategy from the users. A built-in strategy *simulator* uses the interface to create sample paths created by given strategies. Moreover, following this interface, users can define their own types of strategies and test them using the infrastructure of FiMDP.

Integration with Storm. FiMDP is now integrated with the state-of-the-art probabilistic model checker Storm [15] through its Python interface Stormpy. The integration works in two ways. First, we can translate a CMDP from FiMDP into Storm's data structures as an equivalent MDP that has the energy constraints encoded in states and actions. Second, we define CMDPs as a special

case of MDPs with cost (for consumption) and labels (for reload states) and FiMDP can read all such MDPs from Storm into its own data structure as CMDPs. Storm's ability to read PRISM and JANI models extends thereby allowing us to read CMDPs expressed in the two languages into FiMDP. We implemented a direct support for reading PRISM files, including a convenient interface for parametric models, e.g., models with undefined constants.

Heuristics for Expected Reachability Time. The basic synthesis algorithm of [5] ensures that an accepting state is eventually reached with probability 1 (and then again and again, *ad infinitum*). However, it is purely qualitative and ignores the precise transition probabilities during strategy synthesis. Further, it does not take into account the expected number of steps to reach an accepting state, hereinafter referred to as the expected reachability time (ERT), a parameter of significant practical importance: e.g. from a patrolling unmanned vehicle we expect that it visits all the checkpoints in a reasonable amount of time. In order to tackle this issue, FiMDP employs two heuristics proposed in [6]: the *goal-leaning* heuristic and its extension, the *threshold* heuristic with a *threshold* $0 \leq \delta \leq 1$ parameter. Both heuristics modify only the way in which the strategy π and the values *min-pos-reach* are computed.

Optimal Allocation in Multi-agent Scenarios. In many scenarios, we can use multiple agents to reach a set of accepting states instead of a single agent. Utilizing multiple agents instead of a single agent may significantly reduce the expected number of steps to reach all targets and the required energy capacity of each agent. Given a consumption MDP, a set of target states, and a set of homogeneous agents with fixed initial states, we compute a target allocation and an assignment of targets to agents. The objective is to minimize the *resource capacity* of the agents while ensuring that each target state is infinitely often visited by an agent with probability 1. We utilize a reduction to a new combinatorial optimization problem called *minimal-cost SCC matching* defined on graphs with edges denoting the minimal capacity needed to reach one target from another [9]. We first compute a decomposition of this graph into its strongly connected components (SCCs) using a binary search over the values for the resource capacity. We then assign each SCC to an agent to minimize the resource capacity of each agent. Our recent work [9] showed that this problem belongs to P, and our algorithm can solve this problem in polynomial time.

4 FiMDP: What Is Under the Hood and How to Drive It

FiMDP is written in Python 3 as a library with an interface suitable for the interactive environment of Jupyter notebooks. The basic functionality is accessible without any dependencies, and three third-party libraries are used for more involved features: Spot [10] with its Python bindings is needed for using the Büchi-automata-based and LTL specifications, Storm [15] with Stormpy is needed to read models described in PRISM or JANI languages, and finally GraphViz [13] is used to render visualizations in Jupyter notebooks.

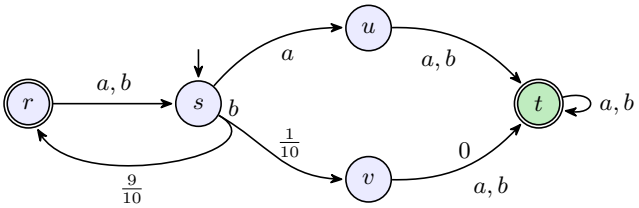


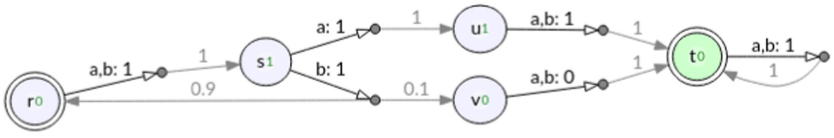
Fig. 1. Example CMDP. The doubly circled states r and t are reload states, s is the initial state, the green-shaded state t is an accepting state. We have two actions, a and b. The only probabilistic effect arises when playing b in s, where $P(s, b, r) = \frac{9}{10}$ and $P(s, b, v) = \frac{1}{10}$. We specify the consumption as $C(v, a) = C(v, b) = 0$ while $C(q, c) = 1$ for all states $q \neq v$ and all actions $c \in \{a, b\}$ (not shown in the figure). (Color figure online)

Threshold Example

[1]:
```
import fimdp
fimdp.setup()

mdp = fimdp.ConsMDP()
mdp.new_states(5, names=["r", "s", "u", "v", "t"])
mdp.set_reload([0, 4])
# action: src_state, dst_distribution, label, consumption
mdp.add_action(0, {1:1}, "a,b", 1); mdp.add_action(1, {2:1}, "a", 1)
mdp.add_action(1, {3:.1, 0:.9},"b",1); mdp.add_action(2, {4:1}, "a,b", 1)
mdp.add_action(3, {4:1}, "a,b", 0); mdp.add_action(4, {4:1}, "a,b", 1);
```

[2]:
```
targets = [mdp.state_with_name("t")]; capacity = 3
solver = fimdp.energy_solvers.GoalLeaningES(mdp, capacity, targets, threshold=0.2)
solver.get_min_levels(fimdp.objectives.BUCHI)
solver.show()
```



[3]:
```
s = mdp.state_with_name("s")
print(solver.get_selector(fimdp.objectives.BUCHI)[s])
```

```
{
  1 – 1: b,
  2+: a
}
```

Fig. 2. FIMDP in a Jupyter notebook. The notebook demonstrates the analysis of the threshold example. Reload states are doubly-circled and the accepting state t is green. (Color figure online)

Figure 2 represents a simple use case in Jupyter notebook. In cell [1] we build the CMDP from Fig. 1 and in cell [2] we compute and visualize the vector *min-lev* for the Büchi objective on the state t; for each state p, the value *min-lev*(p) is shown as the little green number next to the state's name. Finally,

in cell [3] we show that the threshold heuristic indeed chooses action a for all resource levels ≥ 2, and the action b for the resource level equal to 1.

The package is built in a modular fashion; we describe the most important modules from the perspective of users. Figure 2 already uses three of them. The data structures for CMDPs and counter strategies are implemented in `fimdp.core`. To synthesize a strategy, the tool also uses one of the solvers implemented in `fimdp.energy_solvers` for one of the objectives defined in `fimdp.objectives`.

Labeled CMDPs, their product with DBAs, and strategies that keep track of an automaton in their memory are implemented in the module `fimdp.labeled`. Apart from the main symbolic algorithm sketched in Sect. 3, FiMDP allows to encode the energy constraints of a given CMDP into states and actions of a regular MDP that is equivalent to the CMDP via the `fimdp.explicit` module. Finally, such an MDP can be translated to the data structures of Stormpy with a function of the module responsible for the Stormpy integration: `fimdp.io`.

Solvers. Solvers are objects that do the main work. We need to supply a CMDP $\mathcal{M}$, the desired capacity[1], and the set of target states in order to create the solver. Additionally, solvers can accept specific parameters. For example, the `GoalLeaningES` solver accepts the parameter `threshold` in cell [2] of Fig. 2. FiMDP currently offers three solver classes (the `ES` in the names refers to *energy solver*). `BasicES` implements the algorithms as presented in [5]. `GoalLeaningES` implements both heuristics presented in Sect. 3; in fact, the goal-leaning heuristic is now implemented as the special case of the threshold heuristic with `threshold=0`. Finally, `LeastFixpointES` can solve the *safety* problem, e.g. never deplete the energy, more efficiently than `BasicES` on certain classes of CMDPs.

After we create a solver object, we can call the following three functions: `compute`, `get_min_levels`, and `get_selector`. All these functions take an objective (from `fimdp.objectives`) as a parameter and do what their names suggest.

Strategies and Simulators. After we call `solver.get_selector`, we can feed the selector to the constructor of `CounterStrategy`. We also need to initialize the initial memory of the strategy (`init_energy` in case of counter strategies) and set the initial state of the history. Then, `strategy.next_action()` returns the action picked by the strategy for the history it saw so far. The strategy then needs to know how the outcome of this action is resolved, e.g. what is the next state of the history. It accepts the information via the function `strategy.update_state`. Finally, in order to run the strategy again from the same initial conditions, we can call `strategy.reset()`.

Given an initialized strategy and a number of steps n, the class `Simulator` from the `fimdp.core` module queries the strategy n times for an action to play and resolves the outcomes of these actions based on the transition probability

[1] Unlike the definition of CMDP in this paper, the implementation keeps the capacity outside of the CMDP object. That enables us to compare strategies built for different capacities without modification of the CMDP object.

function. The generated history is accessible by `simulator.state_history` and `simulator.action_history`.

Storm, PRISM, and JANI Models. FiMDP can both read and create CMDPs expressed in Storm's data structures. In the first direction, function `fimdp.io.encode_to_stormpy` takes a CMDP and capacity, builds a product CMDP in which the energy constraints are also encoded in states and actions, and converts this product to the `SparseMdp` object of Storm. Alternatively, we can create an MDP with the same state-space as the input CMDP, with the consumption expressed as an action-based reward called `consumption`, and with the reload states labeled by `reload`.

In the other directions, we can read such MDPs with `consumption` reward and `reload` label from Storm to FiMDP via the `storm_sparsemdp_to_consmdp` function. We can read similarly encoded models expressed in the PRISM language by `fimdp.io.prism_to_cmdp`. The function accepts a filename, and possibly a dictionary `constants` in which we can set values to constants left undefined in PRISM parametric models.

Data Structures. To follow the main design goals—simple user interface and easy exploration of the CMDP formalism—FiMDP works with explicitly encoded models. The class that represents CMDPs is called `fimdp.core.ConsMDP`. The states of a CMDP are represented implicitly by integers, and the actions are stored as a list of `ActionData` objects. An additional list maps each state s to the first action of s. Finally, all actions that start in s are linked by a nested list (linked by an attribute of `ActionData`) that is used for effective iterations over them. As seen in Fig. 2, FiMDP offers a convenient interface that hides this representation from the users.

The selected data structure enables interactive building of the CMDPs, simple modifications and processing of actions. It however incurs the price of higher memory requirements in comparison to sparse-matrix based representations of MDPs. As CMDPs can be substantially smaller than their equivalent MDP counterparts, the higher memory consumption is not a considerable limitation. Using the nested linked list enables the tool to quickly iterate the outgoing actions of one state without the need for sorting or other limitations of sparse matrices.

5 FiMDPEnv: Environments for FiMDP

This section presents FiMDPEnv, an open-source Python package containing environments that model real-world consumption Markov decision processes. In particular, we detail two environments that model (i) the stochastic dynamics of one or more unmanned underwater vehicles (UUVs) operating with limited onboard energy storage capacity, and, (ii) the stochastic energy consumption of an autonomous electric vehicle (AEV) operating in the busy streets of Manhattan, New York. All environments in FiMDPEnv are based on real-world data and are designed to show the utility and scalability of our tool. In addition to these two environments discussed in detail below, adding more relevant environments is a part of our release roadmap.

5.1 UUV Environment

The UUV environment models the high-level dynamics of unmanned underwater vehicles (UUVs) operating in environments with stochastic ocean currents. We discretize the area of interest into two-dimensional grid-world environment where the cells form the state space and the UUV is expected to take high-level control decisions in each cell. The action space is comprised of two different classes of actions; the *weak* actions consume less energy but have stochastic outcomes whereas the *strong* actions have deterministic outcomes with the downside of significantly higher energy consumption. In the context of the UUV, the strong actions can model an additional actuator that can be used to correct the UUV course even in the presence of stronger currents. The environment offers up to 16 actions in total with weak and strong variants for each of the 8 directions: East, North-East, North, North-West, West, South-West, South, and South-East. While the ocean currents are stochastic, any data on mean flow velocity and UUV heading velocity available to the user can be readily incorporated into the environment [1]. Often, the UUVs with limited onboard energy storage capability, are expected to safely reload at predetermined locations while pursuing their objective of exploring given targets of interest.

The environment can be accessed by creating instances of one of the two classes in the `UUVEnv` module, `SingleAgentEnv` and `SynchronousMultiAgentEnv` where the former models a single agent operating in the grid-world whereas the latter models a user-specified number of agents acting synchronously in the environment. We now discuss the required inputs and the functionality provided for the single-agent environment with the understanding that the discussion extends to the multi-agent environment where the vector inputs provide information related to multiple agents. The user needs to specify the desired grid size, the reload states, the target states, and the energy capacity of the agents as required inputs. In addition, the users can also optionally specify the initial state, the size of the action space, and the UUV velocity. To run simulations on an instance of the environment, the user needs to generate a counter strategy using the `create_counterstrategy(solver, objective)` method by specifying the solver and objective to use. The state of an instance can also be updated using the `step()` method in one-step increments while the `reset()` can be used to reset the internal state of the environment. We refer the reader to the documentation for further details.

5.2 AEV Environment

The AEV environment, introduced in [5], models the routing problem of an autonomous vehicle operating on a street network. For our study, we consider the area in the middle of Manhattan, from 42nd to 116th Street. The user can specify their own region of interest by providing appropriate data. Intersections in the street network and directions of feasible movement form the state and action spaces of the MDP. We use intersections in the proximity of real-world fast charging stations [21] in the area of interest as the set of reload states.

After the AEV picks a direction, it reaches the next intersection in that direction deterministically with a stochastic energy consumption. As described in [5], we estimate the energy consumption distribution using the distribution of velocity on different road segments and discretize it into three possible values (c_1, c_2, c_3) reached with corresponding probabilities (p_1, p_2, p_3). We then model the transition from one intersection to another using additional dummy states creating a CMDP with 7378 states and 8473 actions.

The environment can be accessed by creating an instance of the `AEVEnv` class in the `AEVEnv` module of FiMDPEnv. The required inputs are similar to the UUV environment with the exception of reload states which are already provided in the environment. The user needs to specify the starting and destination states of the AEV along with its energy capacity. While we consider a simple routing problem in this environment, a similar structure can also be used to model a variety of resource allocation and navigation problems in stochastic environments.

6 Evaluation

In this section, we first compare the time needed by FiMDP for strategy synthesis for a given CMDP, to the time needed by Storm to solve the equivalent problem with the energy constraints encoded in the state space of a regular MDP. Then we demonstrate the effect of the goal-leaning and the threshold heuristics on the expected reachability time.

6.1 Analyzing CMDPs in FiMDP and Storm

Storm is an open-source, state-of-the-art probabilistic model checker designed to be efficient in terms of time and memory. This section reveals whether the theory behind FiMDP can beat the efficient implementation of Storm.

We use the UUV environment described in Sect. 5 to generate CMDPs for grid-worlds of varying sizes and capacities. We measure the time FiMDP needs to analyze such CMDPs and to synthesize the corresponding strategy for the given Büchi objective. We also transform the CMDP into the equivalent MDP with the energy constraints encoded in states and actions. We then measure the time that Stormpy needs to finish `stormpy.model_checking(mdp, prop)` for this MDP and the qualitative Büchi property expressed in PCTL [2] as:

```
prop = 'Pmax>=1 [G Pmax>=1 [F "target" & Pmax>=1 [F "reload"]]]'
```

The computation time for different test scenarios, averaged over multiple runs, is presented in Fig. 3. The plots present the variation of average computation time with both capacity and size of the grid-world, both of which together define the overall size of the model. We can observe that FiMDP outperforms Storm in terms of computation time in all test cases with the exception of small problems where Storm, owing to its efficient C++ implementation, is faster. The advantage of FiMDP lies in the fact that the state-space of CMDPs (and also the time needed for their analysis) does not grow with rising capacity.

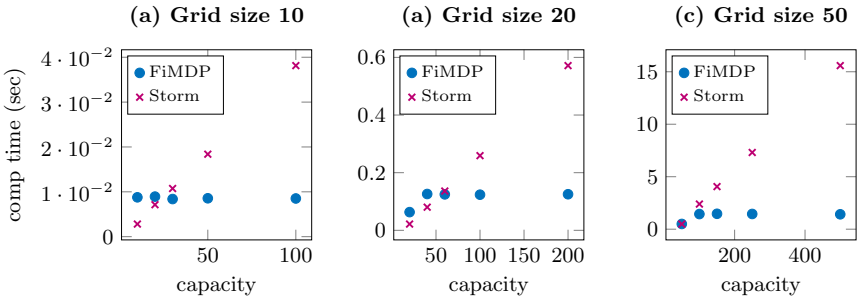


Fig. 3. Mean computation times for solving the CMDP model of the UUV environment with varying capacities proportional to the size of the grid-world. Each subplot in the figure corresponds to a different size of the grid-world.

6.2 Goal-Leaning Solvers

This section investigates the novel heuristics from a practical, optimal decision-making perspective. We utilize the UUV environment discussed in Sect. 5. The test scenario contains a single reload state and a single target, where the objective of the agent is to travel from its initial state to the target state and keep enough energy to be able to come back to the reload state. We consider agents with three different strategies generated by the solvers of FiMDP and measure the expected reachability time (ERT) introduced in Sect. 3 using 10,000 independent runs with a simulation horizon of 10,000. The agent following a counter strategy generated by the `BasicES` solver (no heuristic) never reached the target within the simulation horizon, since the probability of reaching the target between two visits to the reload state was too small. The agent with a strategy generated by the `GoalLeaningES` solver with no threshold (goal-leaning heuristic) needed about 124 time steps on average to reach the target. Finally, the agent following a strategy generated by `GoalLeaningES` with threshold 0.1 needed 62 time steps on average. Table 1 summarizes the ERT values for the discussed three solvers.

The goal-leaning solver ensures that the agent heads towards the target with high probability; only at the one place in the middle between the goal and the reload state, this agent picks a "wrong" action. The additional threshold eliminates this drawback and thus ensures that the agent always proceeds in a near-optimal fashion leading to a significant improvement in the ERT.

Table 1. Comparison of different energy solvers

Solver	Expected reachability time
BasicES	–
GoalLeaningES with threshold 0	124
GoalLeaningES with threshold 0.1	62

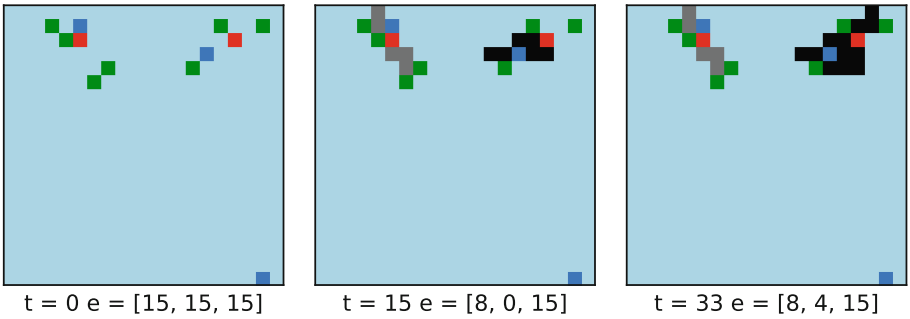


Fig. 4. Demonstration of the multi-agent allocation algorithm with 3 agents (blue cells), 7 targets (green cells) and 2 reload states (red cells). The images provide snapshots of the current and historical locations of the agents at different time instances. We denote the trajectory of two different agents with black and gray cells. Note that the agent at the bottom right of the grid is not allocated to any target and therefore takes no action to transition into different cells in the grid. (Color figure online)

6.3 Multi-agent Allocation

In this section, we demonstrate the multi-agent allocation algorithm in the UUV environment, as in the previous subsection. This test scenario consists of allocating seven targets to three agents to satisfy a Büchi objective and then return to their initial location. As previously mentioned in Sect. 5, the allocation algorithm from [9] computes a target allocation and assignment to an agent to minimize the resource capacity of each agent, while satisfying the Büchi objective. Figure 4 shows a situation where the required capacity increases if all agents are required to allocate some of the target locations. In Fig. 4 we illustrate the initial locations of the agents and targets (left), the time-step (indicated with t) where the current energy level (the vector e) of one of the agents is minimal (middle), and the final time-step where all targets are visited by some agent.

The capacity required with the assignment (and the corresponding strategy) computed by the optimal allocation in this multi-agent scenario is 15. On the other hand, if we require to allocate some of the target locations to each agent, the minimal required capacity is 54, which is about four times larger compared to the optimal allocation. Such an allocation may induce a shorter time to visit all locations by using all agents. However, the difference in the required capacity highlights the tradeoffs between computing an allocation and a strategy that induces a trajectory with minimal time and energy capacity. We also estimate the expected time to visit all target locations using the allocation by simulating the strategies in the underlying consumption MDP. We run 10000 simulations with this strategy. On average, the strategy synthesized by using `GoalLeaningES` solver with threshold 0.3 needed 32.3 steps on average.

7 Conclusion

We introduced FiMDP, a tool for strategy synthesis in consumption MDPs with deterministic Büchi LTL objectives. The tool provides a robust framework for modeling, synthesis, simulation, and analysis of discrete resource-constrained stochastic systems. Our experiments show that FiMDP can efficiently handle models of real-world scenarios.

References

1. Al-Sabban, W.H., Gonzalez, L.F., Smith, R.N.: Extending persistent monitoring by combining ocean models and Markov decision processes. In: 2012 Oceans, pp. 1–10 (2012)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Bertsekas, D.P.: Dynamic Programming and Optimal Control, 3rd edn, Vol. II. Athena Scientific (2007). ISBN 1886529302
4. Bharadwaj, S., Le Roux, S., Pérez, G.A., Topcu, U.: Reduction techniques for model checking and learning in MDPs. In: 26th International Joint Conferences on Artificial Intelligence, pp. 4273–4279 (2017)
5. Blahoudek, F., Brázdil, T., Novotný, P., Ornik, M., Thangeda, P., Topcu, U.: Qualitative controller synthesis for consumption Markov decision processes. In: 32nd International Conference on Computer-Aided Verification, vol. II, pp. 421–447 (2020)
6. Blahoudek, F., Novotný, P., Ornik, M., Thangeda, P., Topcu, U.: Efficient strategy synthesis for MDPs with resource constraints (2021)
7. Brechtel, S., Gindele, T., Dillmann, R.: Probabilistic decision-making under uncertainty for autonomous driving using continuous POMDPs. In: 17th International IEEE Conference on Intelligent Transportation Systems, pp. 392–399 (2014)
8. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 151–168. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_9
9. Cubuktepe, M., Blahoudek, F., Topcu, U.: Polynomial-time algorithms for multi-agent minimal-capacity planning (2021)
10. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
11. Fainekos, G.E., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for mobile robots. In: IEEE International Conference on Robotics and Automation, pp. 2020–2025 (2005)
12. Feinberg, E.A., Shwartz, A.: Handbook of Markov Decision Processes: Methods and Applications. Springer, Cham (2012)
13. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Softw. Pract. Exp. **30**(11), 1203–1233 (2000)
14. Hartmanns, A., Junges, S., Katoen, J.-P., Quatmann, T.: Multi-cost bounded reachability in MDP. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 320–339. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_19

15. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker storm. Int. J. Softw. Tools Technol. Transfer 1–22 (2021)
16. Kober, J., Bagnell, J.A., Peters, J.: Reinforcement learning in robotics: a survey. Int. J. Robot. Res. **32**(11), 1238–1274 (2013)
17. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
18. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: 6th Annual ACM Symposium on Principles of Distributed Computing, pp. 377–410 (1990)
19. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57 (1977)
20. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Hoboken (2014)
21. United States Department of Energy. Alternative fuels data center (2019). https://afdc.energy.gov/stations
22. Wolff, E.M., Topcu, U., Murray, R.M.: Robust control of uncertain Markov decision processes with temporal logic specifications. In: 51th IEEE Conference on Decision and Control, pp. 3372–3379 (2012)