UNIVERSITY OF GENOVA

Decision Theoretic Planning for HRI Tasks: a Case Study for the iCub Robot

Praveenkumar Vasudevan, Francesco Leofante

Shashank Pathak, Armando Tacchella

1 Abstract

Decision Theoretic planning, where a probabilistic model of the environment is known, could represent a fast way to implement an optimal behaviour in a robot. However, applying such techniques to robotic tasks often poses many challenges e.g. state explosion, discrepancies between model and real domain. In this paper, we study a simple, yet challenging, cooperative task where the iCub robot has to reach for a ball which is moved by a human. We propose a formalisation based on MDPs and POMDPs and investigate whether Stochastic Planning techniques can be employed in such complex scenarios, where human presence is also considered. Results are validated using the iCub simulator.

2 Introduction

Robot programming is a daunting task, hence reprogramming the robot for a similar – but not identical task – could be undesirable. Learning is an elegant solution in such cases [14], [16]. Stochastic planning where probabilistic model of the environment is known, could be a fast way to implement an optimal behaviour in a robot while performing an explicit task. Examples of such scenario could be robot in the kitchen, handling a glass to an elderly or a programmable manipulator at workshop assisting human expert through tool handling.

The classical planning approach heavily relies on having a complete and certain description of the situation the agent is facing . Furthermore, actions need to be fully deterministic and the only changes allowed to occur in the environment are due to actions the agent decides to execute. Obviously, these constraints are too restrictive when it comes to modeling a domain depicting the real world: there might be other agents altering the state of the world (including nature), actions might have stochastic or probabilistic effects and the current state of the world might not be fully known.

This is where Markov Decision Processes (MDPs) come in the picture: by introducing stochasticity we can now deal with real world domains using probabilistic planning. This approach tries to generate policies, mappings from states to actions, giving the best action to take in each state. However, with MDPs we assume that the robot can perfectly sense the ambient environment. This assumption is reasonable in carefully engineered settings, such as robot manipulators on manufacturing assembly lines, but becomes more and more difficult to justify, as robots venture into new application domains in homes or in offices. In these uncontrolled, uncertain, and dynamic environments, motion planning with imperfect state information is critical for autonomous robots to operate reliably.

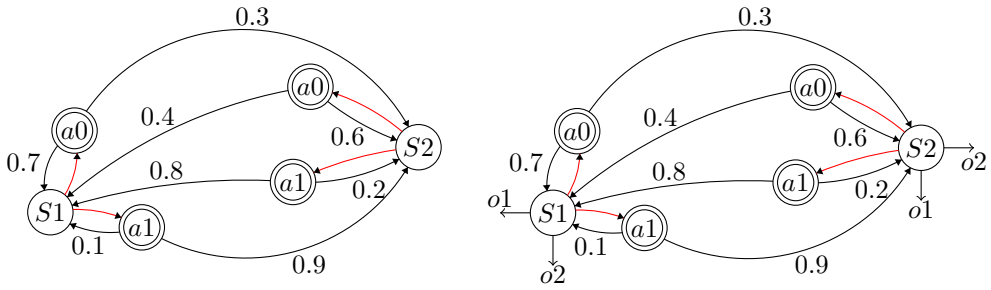
Partially Observable MDPs (POMDPs) are a principled general framework for such planning tasks. With imperfect state information, the robot cannot decide the best action based on a single current state, as it is unknown. Instead, the robot must consider all possible states consistent with the observations. POMDP planning systematically takes into account the uncertainty in robot control, sensing, and environment changes, in order to achieve robust performance. Further details will be given in section 3 of this document.

Finally, Model Checking techniques [13], [4] will be applied to the policy generated by planning. This will allow us to verify that the policy obtained is safe enough to be deployed on the robot.

3 Background

Markovian Decision Process and Markovian Decision Problem Markovian Decision Process(MDP) is a discrete-time stochastic control process. It is described by a 4-tuple $\langle S, A, T, R\rangle$ where T signifies transition (as a result of action $a \in A$) from state s to next one s' while R is the reward obtained through it. MDP is a special case of *stochastic*

games - a game with one player. The objective of Markovian Decision Problem (abuse of notation, and is also called MDP) is to attain a policy that maximizes some measure over rewards such as discounted infinite horizon reward.
Dynamic programming techniques could be used to solve MDP, for example using either policy iteration or value iteration or their variants [16]. However when either T or R or both are not known a-priori, we use reinforcement learning to solve underlying MDP. We do not deal with such cases here.
In some cases, environment might not be completely (or precisely) observable. Under such scenario, we reason over belief-states rather than states, which is another fancy way to say that instead of a given state we have probability distribution over states – higher is this probability, higher is our *belief* that system is in that state. Such systems are called partially-observable MDPs or POMDPs. Formally a discrete POMDP with an infinite horizon is specified as a tuple $\langle S, A, O, T, R \rangle$, where now O is a set of observations. Once again the agent takes an action a from s to s' but now, makes an observation on the end state s'. Due to the uncertainty in observation, the observation result $o \in O$ is described as a conditional probability over s, s'. To solve a POMDP, we reason in the belief space *B*, the space containing all beliefs, and compute a policy that prescribes the best action for every belief that the robot may encounter.



(a) A simple MDP with two actions

(b) A simple POMDP with two actions and two observations

Solving Bellman's equation
Essential components of our model are: states, actions, rewards and policy. An agent perceives the current state s_t and using some policy, which is a mapping $\pi(s) \rightarrow a$ from state to action, takes an action a_t. As a consequence of this action, environment might change and a reward $r_{t+1} \in \mathbb{R}$ is observed by the agent.
We define, value $V^\pi(s) \in \mathbb{R}$, as expected value of this average-reward R_t where $s_t = s$. In other words,$V^\pi(s) = E^\pi(R_t|s_t = s)$. In context of MDP, Bellman equation refers to the recursive definition

$$V^\pi(s) = r(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^\pi(s')$$

where s' is next-state and Bellman optimality equation is

$$V^*(s) = r(s) + \gamma \arg\max_a \sum_{s'} P(s'|s, a) V^*(s') \tag{1}$$

Given this, approximating optimum *value*, $V^*(s)$ is achieved through an iterative process

$$V(s_t) \rightarrow V(s_t) + \alpha(R_t - V(s_t))$$

where $\alpha \in \mathbb{R}$ is a step-size parameter. This is possible because Bellman optimality equation 1 when viewed as operator L is a contraction mapping i.e $\exists \lambda \in [0,1)$ s.t $\forall u, v \in V^*$

$$\| Lu - Lv \| \leq \lambda \| u - v \|$$

where $\| v \| = \max_i |v(i)|$ and hence has a fixed-point (using Banach fixed point theorem).

4 Problem Formulation

An object $\mathcal{O}$ i.e. a ball resides in d_w-dimensional state-space $\mathcal{S}_w$ and moves with velocity V along a probabilistic parametric trajectory $T_o^\theta(\Delta_i)$ where θ is the parameter vector, while trajectory maps control-step $\Delta_i \rightarrow \mathbb{R}^{d_w}$. For example, trajectory could be a planar ellipse, while $\theta \rightarrow (\mu, \sigma)$ could be Gaussian noise added at each position in trajectory. Human elbow which is thought of as an actuator for moving the object is parametrized by angle ϕ: we assume that, while moving the ball, human elbow pose is always determined by one value of ϕ.
A robot resides in d_r-dimensional state-space $\mathcal{S}_r$ and is also constrained within a subset of this space $\mathcal{S}_{ws}$. An action $a \in \mathcal{A}\ s.t\ |\mathcal{A}| \geq d_r$, by the robot is determined in relation with change of its position in this state-space, along with some auxiliary decisions. A deterministic action a_d is one where given a state s, performance of a_d always results in same next state s'. Stochastic action is where the next state is chosen (with some known, fixed probability) from a non-empty set of states. For eg. $\mathcal{S}_r$ could be 6 dimensional state-space (3 for position and 3 orientation) and $\mathcal{A}$ could be 7 dimensional state-space, where 6 dimensions are velocities in each of dimensions of $\mathcal{S}_r$ while last dimension encodes values such as grasp, release, start, stop or reset for the arm.
A controller for the robot is a mapping from action space $\mathcal{A}$ to change in position of robot i.e $\Delta_{\mathcal{S}_{ws}}$, let us denote it by $\mathcal{C}_{ctr}$. In stochastic setting, there would be more than one such delta. Motion of object could also be seen as a controller, albeit an independent one and possibly constrained to some trajectory (for e.g in 2D case, x velocity is specified and y is derived from trajectory of the object).
A combined state-space is a product of state-space $\mathcal{S}_{ws}$ with $\mathcal{S}_w$, and is simply denoted by $\mathcal{S}$. A transition function is a mapping from (s, a) to next-state probability i.e $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R} \in [0, 1]$.
A catch is defined as boolean condition which is true when $\mathcal{O}$ coincides with corresponding part of $\mathcal{S}_{ws}$ and grasp is activated. (Grasp itself might have probability!)
Distance between robot and object is defined as some norm of difference of common dimensions of $\mathcal{S}_{ws}$ and $\mathcal{S}_w$. For e.g it could be L-2 norm of vector formed by difference of 2-D $\mathcal{S}_{ws}$ and taking corresponding two dimensions from $\mathcal{S}_w$. Clearly $d : \mathbb{R}^{d_w} \rightarrow \mathbb{R}$. The same holds for distance between robot arm and human arm.
Reward is a scalar defined for each state, a reward may have many components such as reward for each state, for unsafe termination and for goal denoted by $\mathbf{R} = \langle R_s, R_t, R_g \rangle$ respectively. Each of these components could have some weight i.e $\mathbf{w} = \langle w_s, w_t, w_g \rangle$. Overall scalar reward then is dot product of $\mathcal{R} = \mathbf{R}.\mathbf{w}$. More specifically, we assign $R_s = -d$ where d is the current normalized distance between robot and object; $R_u = -c_u$ where c_u is a negative constant, assigned whenever $d_{rh} \leq r_h$ where d_{rh} is the smallest distance between robot arm and human arm, and r_h is arm radius ; $R_g = c_g$, where c_g is a positive number.
Policy is defined as a map $\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R} \in [0, 1]$ i.e probability of choosing action a in state s.
A discretization function is a mapping from a state-space to an integer i.e $\mathcal{D} : s \in \mathcal{S} \rightarrow \mathbb{Z}$.
Problem definition:
Given $T_o^\theta(\Delta_i)$, $\mathcal{S}_w$,$\mathcal{S}_{ws}$ and $\mathcal{A}$, find a policy π that minimizes cost.
Assumptions:

Underlying assumptions in our model are few but very important. Firstly, the trajectory of the ball is not affected – in any way – by actions of the robot. This implies that we can indeed think this as restrictive case of MDP rather than a stochastic game. Secondly, we assume that a suitable discretization functions exists for both $\mathcal{S}_w$ and $\mathcal{S}_{ws}$, where *suitable* means that it is coarse enough to make planning feasible but fine enough so that observation in underlying continuous domain are faithfully observed in discretized space too.

Overall algorithm:

A high-level picture of overall algorithm is given in 1.

Algorithm 1: OVERALL PLANNING

Input: System parameters $T_o^\theta(\Delta_i)$, $\mathcal{S}_w$,$\mathcal{S}_{ws}$ V_x and $\mathcal{A}$
Output: Model of system, optimal policy π

$i \leftarrow 0$
$\mathbf{R} \leftarrow \mathbf{0}$ reward table is initially empty
$\mathbf{T} \leftarrow \mathbf{0}$ transition table is initially empty
while *no-terminate and enough samples* **do**
    Generate next position for object using $T_o^\theta(\Delta_i)$ and V
    Generate next position for robot using some controller
    Calculate combined state-space for previous and this position $s_{\Delta_{i-1}}$ and s_{Δ_i}
    Calculate reward $\mathcal{R}$
    Update reward table $\mathbf{R}$ and transition table $\mathbf{T}$
    $i \leftarrow i + 1$
model $\leftarrow \langle \mathbf{R}, \mathbf{T} \rangle$
$\pi \leftarrow$ plan for model
return π

5 Implementation

5.1 iCub Simulator

The iCub simulator has been designed to reproduce, as accurately as possible, the physics and the dynamics of the robot and its environment [17]. The same software infrastructure and inter-process communication used for the simulator could be used to control the physical robot as well, which makes the development and testing process a breeze. iCub uses YARP (Yet Another Robot Platform) as its software architecture. YARP is an opensource software tool for applications that are real-time, computation-intensive, and involve interfacing with diverse and changing hardware. The simulator and the actual robot have the same interface either when viewed via the device API or across network and are interchangeable from a user perspective. The simulator, like the real robot, can be controlled directly via sockets and a simple text-mode protocol; use of the YARP library is not a requirement. This can provide a starting point for integrating the simulator with existing controllers in esoteric languages or complicated environments.

The iCub simulator has been created using the data from the physical robot in order to have an exact replica of it. As for the physical iCub, the total height is around 105cm, weighs approximately 20.3kg and has a total of 53 degrees of freedom (DoF). These include 12 controlled DoFs for the legs, 3 controlled DoFs for the torso, 32 for the arms and six for the head. The robot body model consists of multiple rigid bodies attached through a number of different joints. All the sensors were implemented in the simulation on the actual body, such as touch sensors and force/torque sensors.

As many factors impact on the torque values during manipulations, the simulator might not guarantee to be perfectly correct. However the simulated robot torque parameters and their verification in static or motion are a good basis and can be proven to be reliable

5.2 DDL and Planner

The domain description language specified the problem domain in an symbolic/explicit way in a format supported by Planners. There are many Domain description language available for probabilistic domain definition. In particular, for this project we decided to model the domain using the so called *.pomdp* format, developed by Anthony Cassandra at Brown University [3]. Before choosing POMDP format, an extensive research on the available DDLs has been carried out. We invested a fair amount of time investigating Relational Dynamic Influence Diagram Language (RDDL) developed by Scott Sanner at Australian National University [15]. RDDL has been chosen as official language for ICAPS 2011 and 2014 competions [2]. This language is probably the most expressive among the ones available (richer syntax, extended functionalities) but unfortunately RDDL is not fully supported by any of the planners available on the market. For this reason, we decided to adopt Cassandra's format. With POMDP format, we can specify all the parameters characterizing the model in an explicit way, assigning a probability to each tuple $\langle a, s, s' \rangle$. However, one of the major drawbacks of this kind of explicit encoding is that, when the number of states grows, modifying the model by directly handling the code becomes impossible.

```
##DOMAIN DEFINITION##
discount: %f
values: [reward]
states: [ %d, <list of states> ]
actions: [ %d, <list of actions> ]
observations: [ %d, <list of observations> ]

start: <start-state>

T: <action> : <start-state> : <end-state> %f

O: <action>
%f %f ... %f

R: <action> : <start-state> : <end-state> : <observation> %f
```

Figure 2: Example of *Cassandra Pomdp* file structure

Once the problem domain is defined, we can feed the *.pomdp* file to the planner to compute a policy. The choice of the planner was relatively easy: after choosing one DDL, the number of compatible planners is pretty limited. For this project, we decided to use APPL (Approximate POMDP Planning Toolkit) developed at National University of Singapore (ranked 1st in ICAPS 2014 competition) [1] . The planner integrates two algorithms: one based on offline policy computation and one based on online search. It chooses one of them, depending on the state space size of the given task. For small instances, the offline algorithm, SARSOP, is executed. SARSOP [5] is an efficient point-based algorithm that backs up from belief points which are close to be optimally reachable from the given initial belief. For large instances, the online algorithm is triggered. It uses particle filters to represent beliefs and UCT, a Monte-Carlo simulation method, to evaluate the value of potential actions. In addition to supporting Cassandra POMDP format, the

APPL planner supports POMDPX which is XML format for specifying the MDP/POMDP domains. It is possible to specify the full/partial observability in the POMDPX file.

5.3 Software Architecture

The Software architecture for the entire pipeline is designed and implemented in a modular, configurable and extendable manner. The implementation is done is $C++$ incorporating Object oriented programming techniques. Many extension points are provided to add support for different Domain Description language generators, DDL translators in the future. The schematic diagram of the software architecture along with the dataflow is specified in figure 3. The system is completely configurable using *Javascript Object Notation (JSON)* files. The system accepts the JSON configuration file as the input which contains all the necessary information needed by all the modules that composes the software.

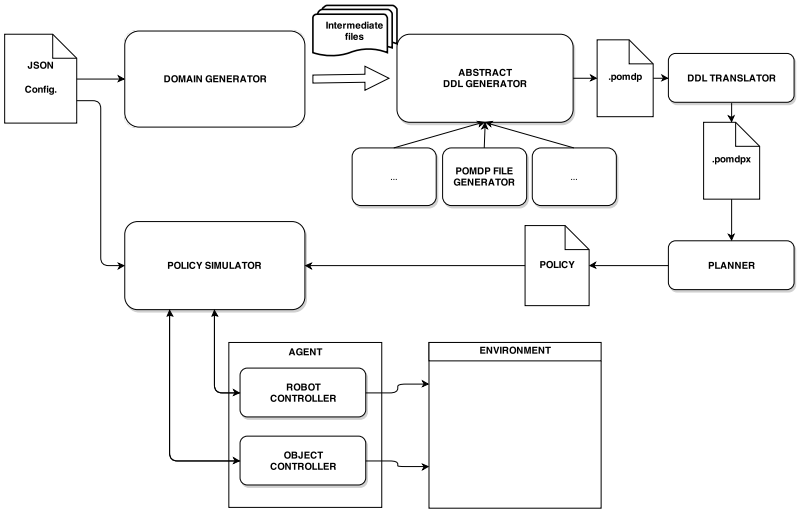


Figure 3: Software Architecure

The **Domain Generator** consumes from the configuration file, the information about the state space bounds, discretization, action discretization, trajectory type for which it has to generate the model, reward functions etc., The domain generator constructs the augmented state space described in section 4 and constructs the transition probabilities and state reward mappings. These information are serialized into intermediate files which will be needed by the other components in the system. The **DDL Generator** reads the intermediate files generated by the Domain generator module and serializes the domain model in prescribed DDL format say for instance, the Cassandra POMDP format. The generated file is translated into POMDPX format and the full observability is specified for the MDP case. Note that translations from one DDL to another are made possible thanks to ad hoc translators. For POMDPs, it is possible to append Observation table to POMDP file specifying, for each state, a probability distribution over the states sensed by the robot, given the actual current state. In order to set these probabilities, say s' we simply pick a fixed number of neighbours of s' and assign probabilities by sampling a Gaussian distribution, with configurable μ and σ. The POMDP/POMDPX file is then fed to the APPL planner which generate the MDP/POMDP policy π. The

Policy Simulator takes the policy and domain model information and executes the policy interacting with the agent controller and the object controller. The **Agent Controller** is the low-level controller of the agent which in our case is iCub robot but could be modified to support any agents/robots with minimum effort. The agent controller receives the target information and commands the cartesian controller to move the agent to the desired position. When the policy simulator sends the grasp action command the agent controller commands the position controller to grasp the object. The **Object Controller** on the other hand takes care of controlling the object to move it in a desired trajectory at a given speed. It also controls the position of elbow for each of the object locations or vice versa. In the case of noisy conditions it takes into account the noise distribution and executes the trajectory accordingly. The policy simulator chooses the best action from the policy by augmenting the states of the agent and the object at any particular time instant. The YARP communication infrastructure is used for the transactions between Policy simulator and Agent, Object controllers. However this could easily be ported to any other Robotic platforms like ROS to experiment this pipeline with different Robots available. The **Policy Simulator** also generates the necessary encodings needed to check the safety of the policy using model checking techniques described in section 6.

6 Model Checking

We are now interested in using Model Checking techniques to verify safety properties (i.e. probability of reaching a bad state).Since we wish to consider stochastic policies, safety cannot be assessed in terms of Boolean logic, but probabilistic reasoning is to be used instead. This can be achieved through Probabilistic Model Checking techniques [7] wherein system and properties have probability values associated to them. For instance the usage of probabilistic model checking in safe learning is described in [11]. Now since we want to answer to questions like "what is the probability that the robot will show undesirable behaviour?", we need quantitative analysis. Properties of this kind can be expressed and analyzed using logics such as Probabilistic Computation Tree Logic (PCTL) [12]. These methods often rely on deriving a probabilistic model of the underlying process, hence the formal guarantees they provide are only as good as the estimated model. As we already said, in a real setting, these estimations are affected by uncertainties due, for example, to measurement errors or approximation of the real system by mathematical models. Starting from the domain definition, we generate a policy π as described in previous sections. We know that this policy specifies the best action to be taken for each of the combined states in $\mathcal{S}$, so for example we know that starting from a generic state s_1, $\pi(s_1)$ will give an action that eventually will take the robot to state s_2. The agent may perceive noise in the trajectory of the object, that is, we have uncertainty on the object-related component of s_2 . In particular here we assume that, as a result of this noise, the action we have performed will actually result in a fixed number of different possible outcomes, each of which will have a probability associated. These probabilities are sampled from Gaussian distributions $\rightarrow (\mu, \sigma)$ defined over the object discretized domain, and assigned to the neighbouring states of s_2 keeping into account the kind of discretization we performed on the object trajectory.

By taking in account of noise in the computed policy, we will obtain a Discrete-Time Markov Chain (DTMC), where the system undergoes transitions from one state to another with given probabilities. Now that we have the probabilistic model describing the behaviour of the system, we can perform safety property verification using PCTL, which allows for probabilistic quantification of desired properties. As a result of this process, we will be able to know to what extent our policy is safe (i.e. what is the probability of ending in an undesirable state in which the robot collides with the human arm). If the policy results to be safe, we can deploy it on the robot. Safety cannot be guaranteed in two cases, either the ball is moving along the desired trajectory with too much noise (σ too big), or the ball is moving along a shifted version of original trajectory (acceptable σ but $\mu \neq 0$). The second case is analogous to another human subject moving the object and re-planning is only possible in this case, while for the first case we can just notify that the safety

cannot be assured. A typical scheme of such kind is described in figure 4
In order to perform verification we can use PRISM model checker [6]. Generally the complexity of PCTL model checking is polynomial in model size (number of states), moreover models for realistic case studies are typically huge (State space explosion problem). PRISM offers fully automated processes, tools for visualisation of quantitative results [10]. The formal description and specifying safety properties in robotics context is specified in [8].

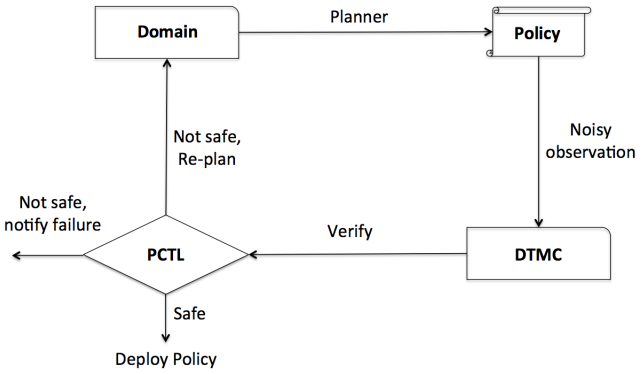
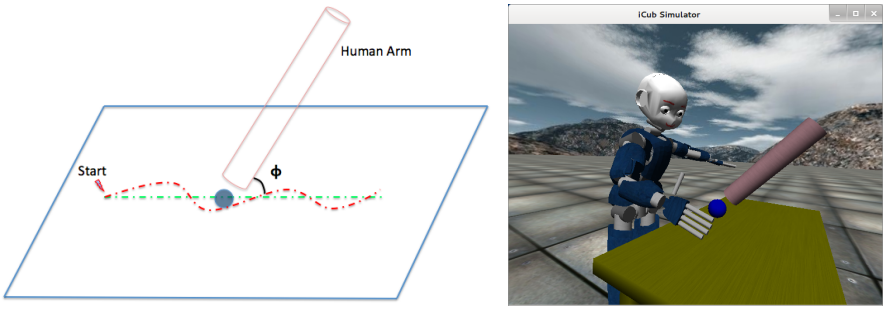


Figure 4: Model Checking scheme for Safety assurance and Policy Repair

7 Experimental Setup

Consider a object-space $\mathcal{S}_w$ as 3-D, with reals in range $[0, 1]$ while robot-space as $\mathcal{S}_{ws}$ as 4-D with 3 reals in range $[0, 1]$ while one boolean (used for grasping). Consider the trajectory $T_o^{\theta}(\Delta_i)$ as noisy straight line ie of form $\frac{x+a}{b} = t$, $y = c$, $z = d$ where $x = \hat{x} + \epsilon_1, y = \hat{y} + \epsilon_2, z = \hat{z} + \epsilon_3$ such that $\epsilon_1, \epsilon_2, \epsilon_3$ are independent Gaussian noise of mean 0 and $\sigma = .03$. Note that both trajectory choice and noise parameters have been made configurable.Consider human arm as a cylinder of length $l = 0.30$ and radius 0.02. Let $\phi \in \left[\frac{\pi}{3}, \frac{2\pi}{3}\right]$ and let initial start position of the arm be $\phi = \frac{2\pi}{3}$. Let ball radius be 0.03 and initial start position of the ball i.e. position at Δ_0 be $(\hat{x}, \hat{y}, \hat{z}) = (-0.1, 0.53, 0.35)$. Let action of robot be 4-D with reals in range $[-0.02, 0.02]$ while one boolean for grasp. Let uniform discretization of spaces $\mathcal{S}$ be $\mathcal{D}_s$ with step $\delta = 2cm$ and $\mathcal{A}$ be $\mathcal{D}_a$ with step $\delta = 4cm$, we do not consider the orientation discretization. We assume that the orientation is fixed in such a way that the robot can grasp the ball. For what concerns the object, a fixed number of samples from the trajectory are picked up and combined with the robot state. Let $\mathcal{C}_{ctr}$ be deterministic hence robot could reach any position in the neighbourhood (depending on action range $[-0.02, 0.02]$ here) of its current position with certainty. Let velocity of the object be $V_x \in \mathbb{R} | s_{\mathcal{S}_x}(\Delta_{i+1}) \leftarrow s_{\mathcal{S}_x}(\Delta_i) + V_x * \Delta_i$ whereas y and z positions are constant. Here we also assume that if object collides with the robot, it is perfectly inelastic collision and robot could grasp it in next control step Δ_{i+1}.
Rewards may be defined to have 3 terms, common negative cost in each state, huge negative reward for unsafe termination (collision with human arm) and huge positive reward for goal state. Obviously, we have rewards components that are scalar here. We can choose $\mathbf{R} = \langle R_s, R_u, R_g \rangle$, where R_s is defined as in section 4, R_u and R_g could be for instance -50 and 500 respectively.
Normalization is important and also the kind of norm we use must make sense (since underlying physics is euclidean it would make sense to stick with either L1 or L2 norm). We uniformly explore all the states in the state space and take all the actions possible from the current state and build the probabilistic state transition model. The complete experimental setup is shown in figure 5. All the constant values considered workspace limits,object position, arm

position are obtained by playing around with the iCub simulator and choosing the appropriate values.



(a) Schematic Setup (b) Simulator Setup

Figure 5: Experimental Setup

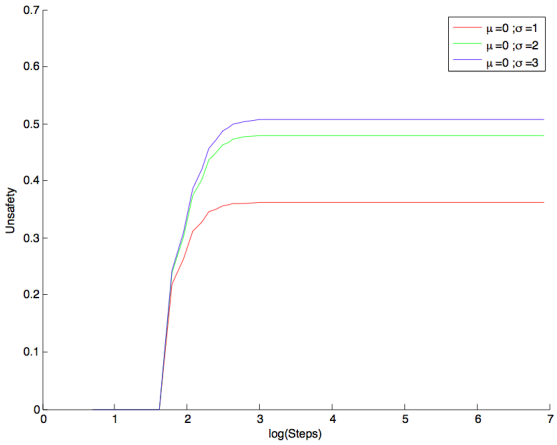
8 Results

For the scenario specified in the previous section, we ran several experiments for different test configurations. Thanks to the configurable software architecture we developed we could have different test configurations and switch between them as and when needed. Please note that for all experiments we ran, the number of actions has been kept constant i.e. 16 actions were allowed. Moreover, since we were interested in studying only the effects noise, also the number of states has been kept constant and equal to 10584. Different levels of noise have been tested, for all experiments, $\mu = 0$, while different standard deviations have been used i.e. $\sigma_1 = 1, \sigma_2 = 2$ and $\sigma_3 = 3$. These values have been chosen according to the physical setup i.e. we want to simulate errors of the order of few centimeters.
For MDP, computing a policy takes on average 7 minutes, reasonable amount of time. For POMDP instead, even with toy problems with a reduced number of states (e.g. 404), including observations made planning time grow dramatically: generally more than 24 hours are needed. Overall we can say that while MDPs could represent a useful tool to approach this kind of problems, POMDPs become soon impractical due to state explosion.
From the collected data, it is possible to infer that, as far as human arm is not considered in the scenario, the policy extracted from the MDP model proves to be pretty robust. When instead we consider the arm, we observed that up to some level of noise the policy can still converge. When noise becomes too high (i.e. σ_3) instead, the robot did not manage to reach the target. This is reasonable: when noise becomes comparable to the ball radius, the environment becomes too noisy to be handled.

MDP	Domain Gen. Time	π Gen. Time	# of steps to convergence
Collision free:			
Moving ball, no noise:	0.19s	6.92s	7
Moving ball, σ_1:	1.71s	538.48s	9
Moving ball, σ_2:	1.88 s	537.65s	11
Moving ball, σ_3:	1.65 s	537.73s	19
Collision:			
Moving ball, σ_1:	1.76s	540.96s	8
Moving ball, σ_2:	1.76s	545.52s	11
Moving ball, σ_3:	1.76 s	538.39s	fail
POMDP	Domain Gen. Time	π Gen. Time	# of steps to convergence
Static ball: 404 states, 404 observation	3.19s	84639s	-

Figure 6: Final results

What can we say about safety of the policy? We wanted to compute the probability of incurring in an undesirable state when following our policy π. Different tests have been carried out using the model checking tools PRISM and COMICS, we started from a trivial case in which $\mu = 0, \sigma = 0$ and then we moved to $\sigma = 1$, $\sigma = 2$, $\sigma = 3$. Here we present some of the results obtained. First of all, we could observe that, increasing sigma, the probability of hitting a bad state grows. This reflects in unsafety increasing with sigma. This result is actually close to expectations: when the variation of noise increases, our policy will be more prone to fail since the position of the ball will be more unpredictable, and therefore the probability of hitting the human arm might increase and as a consequence safety will decrease. The effects of the noise on the probability of reaching unsafe states is specified in figure 7a.



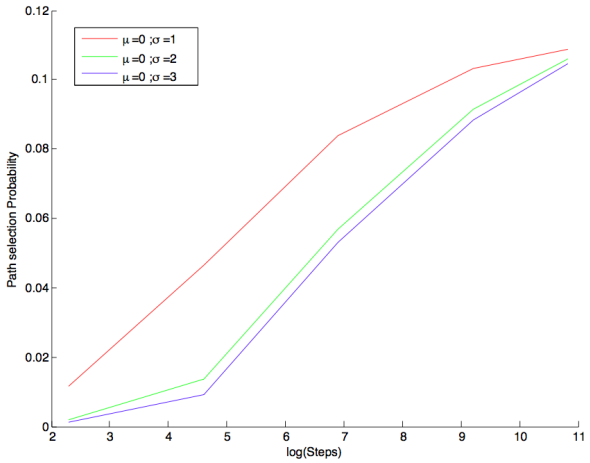
(a) Effect of noise on safety

Mean	σ	States	Transitions	Model generation time
0	1	140	501	2294.061
0	2	140	501	2351.027
0	3	140	501	2918.313

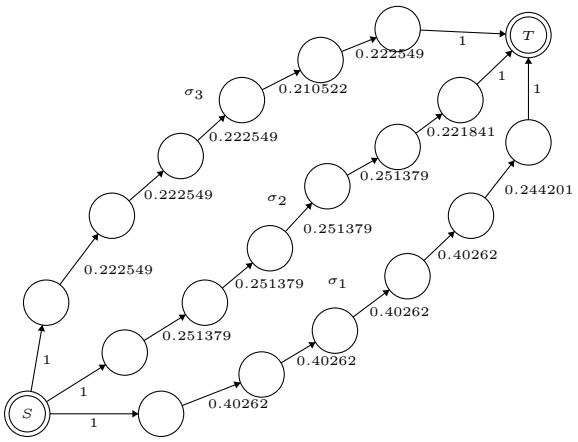
(b) Model building time

After performing verification, we may want to obtain some counter example showing what are the paths with highest probability of ending in a bad state. A path is termed *unsafe* when it carries a risk higher than a threshold $\epsilon \in [0, 1]$ decided *a-priori*, and *safety* is dual of this. The threshold ϵ is arbitrary, and it depends on the system requirements, but typically $\epsilon \ll 1$. Intuitively, in order to stay away from unsafe policies an agent must have a global picture of *risky*

states as much as it needs a global picture of the *goal*. COMICS is used to this purpose (see [9]).These paths can be then used to improve the policy generated by planning and eventually re-plan if necessary.



(a) Reachability probability



(b) Critical subsystem showing unsafe paths

9 Conclusion

Summing up, we have shown a method based on probabilistic planning to implement grasping task on humanoid robots in human-in-the-loop scenarios. Moreover, we have used probabilistic model checking techniques to automate verification and repair of policies learned by planning. Our experimental results do support the feasibility of our method in a realistically-sized application on the iCub. In particular, we have shown that it is possible to implement planning and verification without compromising the performance in a substantial way.
Further developments along this line of research will include more complex setups where the iCub is to guarantee safety at all times and learning control programs for real robots, in order to assess scalability issues of verification and repair. Another interesting direction, would involve more complex domains, where partial observability could play an important role in influencing the final outcome of a policy.

References

[1] Approximate POMDP Planning Toolkit , 2014. http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/index.php?n=Main.PomdpXDocumentation.

[2] International Conference on Automated Planning and Scheduling, 2014. http://www.icaps-conference.org.

[3] A. Cassandra. .pomdp format companion web site, 1999. http://cs.brown.edu/research/ai/pomdp/.

[4] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):263, 1986.

[5] Hanna Kurniawati, David Hsu, and Wee Sun Lee. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *In Proc. Robotics: Science and Systems*, 2008.

[6] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 113–140, 2002.

[7] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. *Formal methods for performance evaluation*, pages 220–270, 2007.

[8] G. Metta, S. Pathak, and A. Tacchella. Is verification a requisite for safe adaptive robots? *IEEE International Conference on Systems, Man, and Cybernetics*, 2014.

[9] Matthias Volk Ralf Wimmer Joost-Pieter Katoen Nils Jansen, Erika Abraham and Bernd Becker. The comics tool - computing minimal counterexamples for dtmcs. *In Proc. of the 10th Int. Symp. on Automated Technology for Verification and Analysis (ATVA12)*, 7561:349–353, 2012.

[10] D. Parker. PRISM - Probabilistic Model Checker, 2010. http://www.prismmodelchecker.org.

[11] S. Pathak, L. Pulina, G. Metta, and A. Tacchella. Ensuring safety of policies learned by reinforcement: Reaching objects in the presence of obstacles with the icub. *IROS 2013*, pages 170–175, 2013.

[12] Alberto Puggelli, Wenchao Li, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. Polynomial-time verification of pctl properties of mdps with convex uncertainties. In *Computer Aided Verification*, pages 527–542. Springer, 2013.

[13] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[14] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd edition*. Prentice Hall, 2003.

[15] S. Sanner. Relational dynamic influence diagram language (rddl): Language description. *IPPC 2011*, 2011.

[16] R.S. Sutton and A.G. Barto. *Reinforcement Learning – An Introduction*. MIT Press, 1998.

[17] Vadim Tikhanoff, Angelo Cangelosi, Paulct Fitzpatrick, Giorgio Metta, Lorenzo Natale, and Francesco Nori. An open-source simulator for cognitive robotics research: the prototype of the icub humanoid robot simulator. In *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*, pages 57–61. ACM, 2008.