A Practical Framework for Robust Decision-Theoretic Planning and Execution for Service Robots

L. Iocchi[1], L. Jeanpierre[2], M. T. Lázaro[1], A.-I. Mouaddib[2]
[1] DIAG, Sapienza University of Rome, Italy
E-mail: {iocchi,mtlazaro}@dis.uniroma1.it
[2] GREYC, University of Caen Lower-Normandy, France
E-mail: {laurent.jeanpierre,abdel-illah.mouaddib}@unicaen.fr

Abstract

The deployment of robots in populated environments is recently gaining more interest because of increased maturity and capability of this technology. In this context, sophisticated planning techniques are required because there is a need of increasing the complexity of the tasks that the robot can accomplish. In particular, there is a large emphasis on service robots, i.e., robots that can satisfy several user needs.

In this paper, we present a practical framework based on a decision-theoretic formalism for generation and execution of robust plans for service robots. The proposed framework has been implemented and succesfully tested on service robots interacting with non-expert users in public environments, facing many sources of uncertainty and failures in task execution.

Figure 1: COACHES robot in shopping mall in Caen.

Introduction

There are many different types of robotic applications having different characteristics in terms of the kinds of robots used, the kinds of environments in which they operate, and the kinds of interactions they have with environments and people. Each type of applications has thus different characteristics that drive the difficulty of the task to be solved. In order to devise suitable and effective planning techniques for robotics applications, it is necessary to analyze the characteristics of such applications and to choose the planning technique that is the most adequate to each kind of tasks.

In this paper, we focus our attention on Service Robotics[1]. The difficulty in task execution for service robots mostly derives from the interaction with non-expert users. Indeed, there is a high variability of tasks to be executed and a high level of uncertainty, given that tasks must be executed upon requests of the users that are unknown in advance.

Planning under uncertainty, execution monitoring and interleaving planning and execution are thus crucial features for an autonomous robot acting in a real environment, specially when human interaction is involved. Although many existing systems have been developed and successfully applied in some application domains, their application to service robotics is still limited.

We consider in this paper the application environment of the COACHES project[2]: a service robot operating in a public environment (in particular, a shopping mall) for assisting customers in many tasks (Figure 1). The robot is equipped with several sensors, including two laser range finders, two RGBD cameras, microphone, touchscreen interface, audio speakers, speech recognition and synthesis for multi-modal interaction with customers. This equipment allows the robot to perform several tasks, such as assistance to provide information and instructions to customers or displaying the path towards a chosen point of interest, personalized and context-based advertisement depending on the profile and location of the customers, escorting a customer in the mall, securing an area when some specific landmarks are detected, such as wet floor, etc.

The problem considered in this paper is the generation and execution of robust plans for service robot tasks, that are characterized by several sources of uncertainty. In particular, in the COACHES application domain, this uncertainty is mainly due to unexpected behaviors of customers of the shopping mall. To overcome such uncertainties, classical and probabilistic planning and re-planning approaches (Ghallab, Nau, and Traverso 2004) could be considered, but

[1] Service robots are expected to "*perform services useful to the well-being of humans*" in Robotics 2020. Strategic Research Agenda for Robotics in Europe.

[2] https://coaches.greyc.fr/ - Funded within the CHIST-ERA research program by MIUR Italy and ANR France.

the frequent changes in the environments make them unsuitable when a re-planning procedure should be executed at each change. However, decision-theoretic planning techniques, such as Markov Decision Processes (MDP) (Puterman 2014), offer a robust mathematical tool to represent uncertainty. This standard way of approaching such problems allows us to drive more reactive policy behaviors dictating to the robot what to do for any state.

In this paper, we describe a framework that exploits and extends two formalisms: Progressive Reasoning Units and Petri Net Plans.

Progressive Reasoning Units (PRU) (Mouaddib, Zilberstein, and Danilchenko 1998) are a family of models designed for achieving anytime computation. As used in this paper, a PRU allows to define several task levels and each level includes several alternative modules with specific durations, qualities, and resource consumption. PRU planning and execution relate to decide which module must be executed at any time according to the PRU structure and to a reward function to optimize. In order to represent uncertainty in tasks involving human-robot interaction, in this paper we provide an extended version of PRU (that we call PRU+).

The execution layer of the proposed framework is based on Petri Net Plans (PNP) (Ziparo et al. 2011) formalism and on a set of Execution Rules (ER). PNP allows to specify a plan and to execute it according to Petri-Net semantics. PNP is an open-source project that has been successfully used in many different robotic applications[3].

As explained in the next sections, the robot programmer can describe the tasks to be executed and the execution rules in a declarative way, by using the PRU+ formal language and the ER specifications, respectively. The developed software described in this paper then guarantees planning and execution according to these specifications. In particular, we provide the algorithms for: 1) transforming the PRU+ description in an MDP, 2) solving the MDP and computing an optimal policy, 3) transforming the optimal policy into a PNP, 4) applying execution rules to generate a new PNP for robust execution, 5) executing the PNP on the robot.

In addition to the novel integration of these techniques in a fully working system for planning and execution on service robots, this paper contains the following additional novel contributions with respect to the state-of-the-art: 1) generation of an MDP from the PRU; 2) transformation of a policy into a PNP; 3) application of ER to generate a final PNP containing a robust plan for the robot.

Experiments with different robots executing different tasks (including a first set of tasks in the shopping mall in Caen for the COACHES project) have demonstrated the effectiveness of the proposed approach. The developed software is available from the COACHES web site for use in other relevant applications of service robotics.

The paper is organized as follows. After discussion on related work, we provide three motivating examples. Then, an overview of the proposed framework is described, followed by the description of the formalisms used to model the tasks and of the algorithms used to generate the plan. Implementation details include a discussion about the performed experiments and computational complexity. Finally, conclusions and future work are discussed in the last section.

[3]http://pnp.dis.uniroma1.it

Related work

Many approaches have been proposed in the literature for human-aware planning, for reasoning in service robots, and for interleaving planning and execution for complex robotic tasks. Most of these works however do not allow for an explicitly model of the uncertainty derived from user interactions. In this paper, we focus the literature analysis to those techniques that allows for explicit modeling of the uncertainty in human-robot interaction with decision-theoretic formalisms.

A standard way of modeling robot domains with uncertainty is to use a Markov Decision Processes (MDP), apply a solver to produce the policy (what to do and when) and then execute the policy, monitoring its success (Ghallab, Nau, and Traverso 2004). However, the execution of a policy generated from standard MDP techniques requires the full evaluation of the current state to determine the actions to be performed. When the state is composed of several independent variables, all of them must be evaluated in order to properly choose the next action. In order to overcome this difficulty, some recent approaches proposed the use of different levels of planning. For example, integration of classical planning with Partially Observable Markov Decision Processes (POMDP) is described in (Hanheide and others 2015). However, continuous switching planning can still be impractical in a dynamic world with frequent human-robot interactions.

In decision-theoretic planning, many techniques have been considered for compact representation of MDPs, states, actions and policies. For the state space, factored representations using a feature-based representation have been considered, leading to factored dynamic programming techniques using different representations, such as Dynamic Bayesian Networks, Decision Trees or Algebraic Decision Diagrams (Cheuk and Boutilier 1997). Bayesian Networks have also been considered for representing actions and transitions giving a graphical and structured model of representation for the transition functions. In factored (feature-based) representation, reward, value and policy tree representations have been proposed. Based on such representations, decision-theoretic and dynamic programming regression have been developed by removing redundant nodes and aggregating nodes using different properties on features, actions or value functions (Boutilier and Poole 1996).

Factored Markov Decision Processes (Boutilier, Dearden, and Goldszmidt 2000) are special MDPs where the state space is made of several state variables. The transition function describes the effect of any action upon each state variable. The main interest of this representation is that independent variables may be split and dealt with separately.

When compact representations of the value function and of the policy in Factored MDPs are used (e.g., (Hoey et al. 1999; Koller and Parr 2000)), the variables to be determined at any execution step are limited by the structure of the compact representation (e.g., decision trees). But it is necessary

to traverse all the decision tree or the decision list, in order to determine the action to be executed. This may imply the evaluation of all the variables in the worst case. One way of handling this problem without flattening the whole state space is to use Acyclic Decision Diagrams (ADD), as for example in SPUDD (Hoey et al. 1999). In this formalism, the function is represented as a decision tree, taking into account one variable at each level of the tree. But each variable may be present or not in any branch of the diagram. This allows for factoring whole branches as single leaves. For example, if the robot is low on energy, it has to recharge, whatever could be the value of the other variables.

The framework we propose in this paper is in the spirit of compact, structured and feature-based (state variables) representation by aggregating states using state variables and ignoring the other variables not relevant for the considered actions. Such a structured representation leads to some kind of hierarchy of levels: at each level there are different nodes representing various options of execution. An originality of this work is that the representation has not only led to an efficient planning method but also to a robust execution procedure, based on Petri Net semantics. This contributes in a tractable but expressive representation and leads to a scalable and robust approach.

More specifically, our approach proposes a rich framework with the following features: 1) it uses a compact, structured and incremental description of the problem, while standard decision models use a flat monolithic description and Factored MDPs offer compact representation but often not structured; 2) as (PO)MDP models, our approach is based on MDP and derives a policy to guide the behavior of the agent with no need of re-planning as opposed to classical planning techniques; 3) incomplete state description is supported by our approach using state variables as in Factored (PO)MDPs. Partial observability is not explicitly expressed in our approach (as opposed to POMDPs) but a kind of partial observability is adopted by allowing observation of only some variables.

Motivating examples

In this section, we describe three examples to motivate our work. The first example is a basic service robot application, that is used to illustrate the concepts of the proposed framework. The other two examples have been actually implemented with real robots interacting with non-expert users in public environments. These tasks are admittedly very challenging and have been realized in the framework of our project by combining several techniques from Robotics and Computer Vision.

Example 1: Basic Service Robot. A robot is able to execute two activities A and B upon request of the user, but the user does not know about these robot abilities. The robot is normally in a waiting state. Whenever a user is detected in front of the robot (for example, with a simple face detector procedure), the robot can decide whether to start an interaction (for example, through a spoken dialogue) and, if so, it describes the activities that it can do A and B. The user can now select one of them or answer that s/he is not interested (for example by selecting an option on a user interface or by a speech command). If a goal is selected, the robot executes a task for achieving it. Each goal can be achieved with two alternatives tasks $TaskA1$ and $TaskA2$ for A and $TaskB1$ and $TaskB2$ for B. After the execution of the task, the robot comes back to the wait state, waiting for a next user. During the execution, the following inconveniences may occur: the user does not complete the interaction (for example, s/he does not answer the robot question), any task may fail for reasons not modeled in the task description, and any task may be aborted according to some external command or condition.

Example 2: Service Robot assisting customers of a shopping mall. A robot operates in the corridor of a shopping mall including several shops. Each shop-keeper can define a set of advertisements that the robot is asked to communicate to customers of the mall and each advertisement has an associated reward for the shopping mall. The robot can execute the following actions in the environment: move to any location (i.e., in front of any shop), approach a specific person or a group of people, perform advertisement actions, perform other assistive actions upon request, bring a person to any location in the mall.

The problem we want to solve is the following: given the current position of the robot, the information about shops and advertisements and an initial probability distribution of the presence of people in the shopping mall (for example, provided by an external system of video-cameras), plan and execute the behavior that maximizes the global actual reward, also recovering from possible failures due to unpredicted or erroneous interactions with people.

Example 3: Service Robot assisting visitors of an office. A mobile robot is used to help people in an office-like environment. The robot is able to stand in its home position, greet users when they get close to the robot, start an interaction with users asking if they need more info about some subject, bring people to an office, and come back to its initial position. The goal of the robot in this scenario is to detect people operating in the environment, interact with them and offer help if needed. Also in this case, many inconveniences may occur due to unpredicted behavior of the non-expert users.

Overview of the proposed framework

In classical on-line planning, execution monitoring is used to determine the correct execution of the planned actions. When some action fails, the execution monitor is able to determine a new initial state and activate re-planning from this new initial state. In general, this mechanism does not change the model of the domain (i.e., the domain description), therefore this approach is suitable as soon as the cause of the failure is somewhat modeled in the domain description, in such a way that a new re-planning will avoid this situation.

However, very often, it is not possible to anticipate and model all the possible causes of failures for the actions. In

these cases, the planner may not be able to find alternative solutions with respect to the ones that just failed. For example, in a domain in which a robot can open doors with its arm and in which the fact that doors may be locked is not modeled in the domain description, a plan can be generated containing an action to open a door. When this action fails, the planner would re-plan to open this door again, since the fact that it may be locked (thus it must be unlocked, before opening) is not modeled in the preconditions of this action. In this case, it would need to upgrade the model adding a representation of the property of a door to be locked or not. A similar situation arises if we want to avoid having the robot to move in the environment when the battery level is low. Again, without an explicit representation of this condition in the planning domain, it would not be possible to generate behaviors that depend on it. On the other hand, adding additional state variables to the plan domain increases the complexity of the planning procedures and, although there exist many efficient planners, it may prevent the use of this technique on-line for complex applications. Moreover, it increases the difficulty of writing the planning domain by the planning expert.

In this paper, we propose an alternative solution to this problem that consists in separating the set of state variables that are needed for planning and execution monitoring in two groups: 1) variables that are used both at planning and at execution time; 2) variables that are used only at execution time. State variables that will be used only at execution time do not affect the complexity of the planning procedure. Moreover, the MDP states are defined only on the set of state variables used at planning time and we do not require them to be observable.

The overall process that is performed in a fully automated way is the following:

- **Input:** a PRU (defined on state variables X and observable properties O) and a set of ER (defined on state variables Y)
- **Output**: execution of a PNP p generated from the optimal policy π according to the PRU and augmented with ER (p requires the observation of the variables O, Y and possibly a subset of variables in X if they are expliclty indicated in the PRU).
- **Procedure:**
  1. From the PRU, an MDP μ is generated
  2. From μ, the optimal policy π is computed through a Value-iteration algorithm
  3. The optimal policy π is transformed into a PNP p_0
  4. ER are applied to the PNP p_0 to generate a final PNP p containing the implementation of the rules.
  5. The final PNP p is executed on the robot.

As already mentioned, the novelty of the above process with respect to the state-of-the-art is twofold: from one side, this process properly integrates different formalisms and components in an effective complete planning and execution system; from the other side, the following components are original: 1) definition of an extended formalism for PRUs (called PRU+) and generation of an MDP from the PRUs; 2) transformation of a policy into a PNP; 3) definition of ER and their application to generate a final PNP containing a robust plan for the robot.

An important feature of the proposed framework is that state variables Y that are used in the ER do not need to be necessarily the same as the variables X used in the PRU+ description. In general, $X \neq Y$ and thus we can use additional variables in the ER for increasing the complexity of the final plan without affecting the complexity of the domain description and hence of the generation of the policy. Moreover, as explained later, while variables in O and Y must be always observable, a variable in X must be observable only if it is explictly used in some execution condition of the PRU+.

In the next sections, we first describe the formalisms used to represent the information about the tasks to be executed (i.e., PRU and ER) and then the algorithms used in the above described process (i.e., generation and solving of the MDP and generation of the PNP).

Task description formalism

The formalism used for describing the robot tasks is composed of two languages: 1) Extended Progressive Reasoning Units (PRU+) for describing the task and the planning domain, 2) Execution Rules (ER) for representing execution conditions and recovery procedures.

Extended Progressive Reasoning Units (PRU+)

PRU+ is a formalism to describe resource-bounded tasks using an acyclic-graph representing modules to execute for accomplishing the task, the execution context variables (needed resource and variable state instances), the probabilistic execution outputs (outcomes) and the reward for executing modules. From this structure, we derive a Relational Dynamic Influence Diagram Language (RDDL) like description of states, actions, observations and stochastic transitions. Similarly to RDDL, PRU+ is a simple description of factored (PO)MDPs.

Formally, a PRU+ is composed of a sequence of *processing levels* $L = (l_1, l_2, \ldots, l_{|L|})$, a set of *state variables* $X = \{X_1, \ldots, X_{|X|}\}$, and a set of *observable boolean properties* of the environment $O = \{o_1, \ldots, o_{|O|}\}$. Each state variable X_i can be assigned a value within a set of finite values, i.e., $X_i \in H_i = \{\bot, h_i^1, \ldots, h_i^{|H_i|}\}$, with $|H_i|$ finite, and $\bot$ denoting a special null value. State variables do not need to be observable at execution time. On the other hand, each observable property o_i must be evaluated at execution time by an external function that is always able to return a truth value for it. Notice that O can contain properties to observe some of the state variables and the special case in which any state in X can be evaluated from observations in O correspond to classical MDP with full observability.

Each processing level l_i is composed of a set of *modules* $M_i = \{m_i^1, \ldots, m_i^{|M_i|}\}$ and it is associated to a set of active state variables $V_i \subseteq X$. We denote with $M = \bigcup_{i=1,\ldots,|L|} M_i$ the set of all the modules in all levels.

Each module m_i^j is defined by a non-empty set of *options* $\{\alpha_i^j, \beta_i^j, \ldots\}$, representing possible outcomes of its execu-

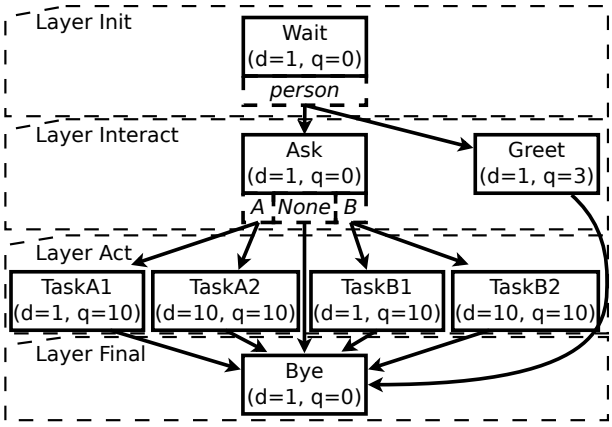

Figure 2: PRU+ for **Example1**.

tion. The symbols used to denote an option (e.g., α_i^j) are assumed to be unique identifiers in all the PRU+.

Each option α_i^j contains the following information:

- *execution condition* $\alpha_i^j.\phi$: a logical formula over atoms in O and equality checks of values in state variables; this formula denotes an observable condition used at execution time to recognize this outcome; we assume that all the conditions for the options of a given module are mutually exclusive;
- *probability* $\alpha_i^j.p$: probability of occurrence of this outcome; the sum of all the probability values for all the options in a module is 1;
- *quality* $\alpha_i^j.q$: estimated quality for achieving this outcome (q can be expressed either as a constant value or as a function of state variables);
- *duration* $\alpha_i^j.d$: estimated time for achieving this outcome (d can be expressed either as a constant value or as a function of state variables);
- *successor modules* $\alpha_i^j.SM$: a set of successor modules that are enabled after this outcome; for each $m_k^* \in \alpha_i^j.SM$, we have $k \geq j$, so successor modules should be either at the same level or at a next level with respect to the current one;
- *state variable updates* $\alpha_i^j.SVU$: a set of state variable assignments that must be considered after this outcome; for each $(X_k \leftarrow h'_k) \in \alpha_i^j.SVU$, $X_i \in V_i$ (only state variables active for the current level are allowed) and $h'_i \in H_i$.

It is important to observe that the definition of some parameters (in particular, the quality) is not trivial, while it affects the generation of the optimal policy. In this paper, we assume that the designer is able to determine these parameters according to his/her experience or from external sources. Learning techniques for determining their values may also be considered, but they are outside the scope of this paper.

Example 1 is modeled with the following PRU+ (also shown in a graphical form in Figure 2): $L = (l_1 = Init, l_2 = Interact, l_3 = Act, l_4 = Final)$, X is empty, $O = \{person, A, B, none\}$. The set of modules M are indicated also in Table 1. Specific values for duration (d) and quality (q) are indicated only in Figure 2, while probability of the outcomes are in Table 1.

module (name)	option	exec.cond	prob.	succ.
$m_1^1\ (Wait)$	α_1^1	$person$	1.0	m_2^1, m_2^2
$m_2^1\ (Greet)$	α_2^1		1.0	m_4^1
$m_2^2\ (Ask)$	α_2^2	A	0.4	m_3^1,m_3^2
$m_2^2\ (Ask)$	β_2^2	B	0.4	m_3^3,m_3^4
$m_2^2\ (Ask)$	γ_2^2	$none$	0.2	m_4^1
$m_3^1\ (TaskA1)$	α_3^1		1.0	m_4^1
$m_3^2\ (TaskA2)$	α_3^2		1.0	m_4^1
$m_3^3\ (TaskB1)$	α_3^3		1.0	m_4^1
$m_3^4\ (TaskB2)$	α_3^4		1.0	m_4^1
$m_4^1\ (Bye)$	α_4^1		1.0	m_4^1

Table 1: PRU+ modules for **Example1**.

Execution Rules (ER)

Execution rules define execution conditions and recovery procedures associated with possible failures of the actions.

Execution rules have the following form:

$$\textbf{if } (\phi) \textbf{ during } a \textbf{ do } \{\sigma; \rho\}$$

where ϕ is a boolean expression over a set of observable state variables Y, a is an action, σ is a (possibly empty) program (i.e., an action, a sequence of actions, a PNP, a remote procedure, etc.), and ρ is a statement used to determine how to continue the execution of the plan, as explained below.

The semantics of such a rule is the following. During the execution of the action a the condition ϕ is continuously monitored. If $\neg\phi$ remains true for the entire duration of the action, then the action terminates normally and the execution rule has no effect. Otherwise, as soon as ϕ becomes true, the following operations are performed in sequence: 1) the action a is interrupted, 2) the program σ is executed, 3) the plan is recovered according to the statement ρ.

The statement ρ can be one of the following values $\{restart\_action, skip\_action, restart\_plan, fail\_plan\}$ that are interpreted as follows.

- *restart_action* : the action a is restarted;
- *skip_action* : the action a is skipped and the next one is executed;
- *restart_plan* : the entire current plan is restarted;
- *fail_plan*: the current plan is terminated with a failure state;

Three remarks about these execution rules are important.

First, the condition ϕ is a boolean formula over a set of variables, which in general are not included in the planning domain description. This feature allows for increasing the overall complexity of the plan execution mechanism, without affecting the complexity of the planning component. In

other words, adding execution variables in ERs would not affect the efficiency of the planning procedure. Variables that are used only in the ERs represent conditions that in general cannot be controlled by the robot and that we assume to be usually "adequate" for the execution of the action.

Second, σ is a recovery procedure that can be implemented in many ways. In this paper, we assume it is an action or a PNP. Again, it is not necessary that the actions used in σ are described in the planning domain descriptions. The goal of this procedure is to ensure that the robot goes back to a suitable state for continuing the execution of the plan. If this is not possible, a failure notice is reported and a new planning task can be activated. Therefore, a rule with empty σ and $\rho = fail\_plan$ corresponds to classical monitoring to detect action failures and activate replanning.

Third, execution rules are associated to actions, but may depend on the actual planning problem (goal, initial situation, etc.) and thus on the generated plan. It is thus possible to have different sets of execution rules for different planning problems.

Example 1 is augmented with the following ER for the action ask:

$$\textbf{if } (\neg person) \textbf{ during } ask \textbf{ do } \{restart\_plan\}$$

and with a set of ER for any of the tasks $task\delta i$ ($\delta \in \{A, B\}$, $i \in \{1, 2\}$). provided that an observable execution condition $valid\delta$ is available to detect the correct execution of the corresponding task:

$$\textbf{if } (\neg valid\delta) \textbf{ during } task\delta i \textbf{ do } \{restart\_action\}$$

Finally, we assume that it is always possible to abort any task upon a specific user command or external condition labelled $abort$, in these cases the robot has to stop any activity and return to its home position (i.e., executing the $home$ action). So the following rules are added for any action $task\delta i$:

$$\textbf{if } (abort) \textbf{ during } task\delta i \textbf{ do } \{home; fail\_plan\}$$

Algorithms

Generating the MDP from the PRU

To compute the optimal behaviour of the robot, we first need to transform the various PRUs describing each active task into a large MDP. Merging PRUs simply consists in concatenating while merging the initial state. This will allow the planner to select the best task to achieve.

Given a PRU+ as described above, it is possible to transform it into an MDP to compute an optimal policy. An MDP is defined as in (Bellman 1957) as (S, A, T, R), with S a finite set of states, A a finite set of actions, $T(s, a, s')$ a transition function denoting the probability for going from state s to state s' using action a, and $R(s, a, s')$ a reward function expressing the expected gain for using action a in state s and arriving in state s'.

State space representation: Each state $s \in S$ is defined by $[\alpha_i^j, X_1, \ldots, X_{|X|}]$, where $X_k = \perp$ for all the nonactive state variables for this level, i.e., $X_k \notin V_i$. An initial state $s_0 = [\alpha_0^0, \perp, \ldots, \perp]$ is also defined in the set of states S, with α_0^0 being a special symbol used only in the representation of the initial state. This state describes the modules that can be used in the beginning. Finally, states that are marked as final by a module's outcome are grouped in a set of final states $G \subset S$. When the execution reaches one of these, the plan is considered terminated and the robot will wait for a next goal. Notice that in the proposed framework, the states do not have to be fully observable at any time, since at execution time only the observable properties and the state variables used in the execution conditions are required to determine the state of the execution of the policy. However, each transition must be fully observable since the execution module needs to know which option has been activated at runtime.

Actions: Actions in MDP correspond to the modules in the PRU+. More formally $A = \{m_i^j | 1 \leq i \leq |L|, 1 \leq j \leq |M_i|\}$

Transition Function: The transition function T of the MDP is defined according to the probabilities and the set of successor modules defined in the PRU. Thus we can define the transition function for the initial state s_0 as follows: $T(s_0, m_1^j, succ(s_0, \alpha_1^j)) = \alpha_1^j.p$, where $succ(s_0, \alpha_1^j)$ denotes a function returning the successor state of the corresponding outcome α_1^j of m_1^j. More formally, for a state s' with a possible outcome α_l^j, $succ(s', \alpha_l^j) = [\alpha_l^j, e_l^{j,(1)}, \ldots, e_l^{j,(|X|)}]$, with

$$e_l^{j,(k)} = \begin{cases} \perp \text{ if } X_k \notin V_l, \text{ (i.e., non active state variables)} \\ h'_k \text{ if } (X_k \leftarrow h'_k) \in \alpha_l^j.SVU, \text{ (just updated)} \\ h_k \text{ otherwise (i.e., the value } h_k \text{ is unchanged)} \end{cases}$$

This process can be repeated at any level to fully define the transition function.

Reward function: Estimated quality and estimated time duration of an option of a module can be represented either as a constant value or as a function of the state variables. In the latter case, we define the following standard function prototype: $f : S \times A \times S \times P \times \Re \rightarrow \Re$ for both the quality and the duration function. More precisely, the 5 arguments of the function prototype have the following meaning: 1) current state s, 2) current action a, 3) successor state s', 4) function type, 5) constant parameter. Notice that these functions are called at planning time to estimate the quality and the duration of an action executed from a state to its successor. These functions (being obviously domain dependent) are implemented by the developer of the action and should return an estimate of quality and duration in the conditions set as arguments. The function type and the constant parameter are arguments used to define general functions that can be used in a parametric way by different modules. For example, the time needed for moving the robot from location A to location B can be computed by path-planning and can include a constant value depending on the context (e.g., say goodbye before leaving).

state s	action a	successor states SS
s_0	$Wait$	$\{(s_1, person)\}$
s_1	Ask	$\{(s_2, B), (s_3, A), (s_4, none)\}$
s_2	$TaskB1$	$\{(s_7, true)\}$
s_3	$TaskA1$	$\{(s_6, true)\}$
s_4	Bye	$\{(s_5, true)\}$
s_5	Bye	$\{(s_5, true)\}$
s_6	Bye	$\{(s_5, true)\}$
s_7	Bye	$\{(s_5, true)\}$

Table 2: Optimal policy for **Example1**.

Dealing with durative actions: Action duration can be implemented as a linear sequence of states with a deterministic transition. Thus, the action will take as many time steps as there are states in the sequence, with no possibility of choice nor failure in between. Please note that in case of failure or interruption, the execution rules added to the policy later will account for it, with no effect on the computation complexity.

Overall algorithm: The generation of the MDP can be thus summarized as follows:

1. Select PRUs for solving current active goals.
2. Expand state variables, evaluate qualities and durations.
3. Build the MDP equivalent to each PRU.
4. Expand durative actions.
5. Merge the MDPs.
6. Solve the large MDP.

Computing the optimal policy

Computing the optimal policy simply consists of solving the MDP model. Classical algorithms like *Value Iteration* (Bellman 1957) or *Policy Iteration* (Howard 1960) are able to compute an optimal policy efficiently. More precisely, the output of the MDP planner is $\langle s_0, G, \{\langle s_i, a_i, SS_i\rangle\}\rangle$, where s_0 is the initial state, G is a set of final states, and in each tuple $\langle s_i, a_i, SS_i\rangle$, s_i is a state, a_i is the action to be executed in this state, SS_i is a set of pairs (s_i^k, ϕ_i^k), with s_i^k being a successor state and ϕ_i^k is the execution condition declared in the PRU+ and associated to the corresponding outcome. Since execution conditions are explicitly added in the output policy, the PNP executor can determine the successor state s_i^k of an action by only observing the condition ϕ_i^k.

As a final note, the policy can be computed with a finite horizon, when there is a maximal number of allowed actions, or with an infinite horizon, when the system is expected to work forever. In the finite case, the policy may change with time. This implies that the generated PNP will be replicated as many times as there are decision steps, each replication being based on a specific time setting. In the infinite case, the policy is said stationary. It does not depend on time and the PNP can be generated only once.

The optimal policy for **Example1** is shown in Table 2, with s_0 being the initial state and $G = \{s_5\}$ the set of goal states. This optimal policy is obtained when the quality values of the actions are set in such a way that the robot prefers to start interactions with users and that $Task\delta 1$ has a higher reward than $Task\delta 2$ ($\delta \in \{A, B\}$), as in Figure 2.

Policy to PNP transformation

The optimal policy computed by the MDP planner is transformed into a PNP by an algorithm that applies the PNP operators. The output of the MDP planner contains the initial and final states, the optimal policy, and the possible successor states for each execution of an action in the policy. In other words, this output corresponds to the optimal policy and to a portion of the MDP model: initial and final states and a subset of the transition function of the MDP containing information only for the pairs (state, action) that are considered in the returned policy and the corresponding execution conditions.

Algorithm 1: Policy to PNP transformation

Input: $\pi = \langle s_0, G, \{\langle s_i, a_i, SS_i\rangle\}\rangle$: policy to execute
Data: Q: a queue of instances of actions
V: set of visited states
Output: p: PNP implementing π

```
push(Q, s_0);
p = empty_PNP;
V = ∅ ;
while Q ≠ ∅ do
    s = pop(Q);
    select ⟨s, a, SS⟩ ∈ π ;
    p = PNP_add(p, ⟨s, a, SS⟩) ;
    foreach s' ∈ SS do
        if s' ∉ V then
            V = V ∪ {s'} ;
            push(Q, s') ;
return p ;
```

The PNP is built by navigating the policy from the initial state to the goal states, as depicted in Algorithm 1. A standard approach to build a graph using a queue of states and a set of visited states is used. The queue Q is initialized with the initial state s_0 which is also added as the initial place in the PNP. While the queue is not empty, the PNP_add function applies the sequence operator to add the action a to the current state s, followed by a set of transitions that will check the conditions ϕ_i^k bringing to any of the states s_i^k, as defined in SS. If a state in SS has not been visited yet, then this state is added to the queue and it will be processed later. The process thus continues until the queue is empty. When encountering final states in G the corresponding places are labeled as 'goal', indicating to the PNP executor to terminate the execution of the plan and return "success".

It is important to observe that the descriptions of the successor states in the output returned by the MDP planner are generated according to the specification of the PRU, as described in the previous section. Thus, they contain the conditions ϕ_i^k over observable properties of the environment that

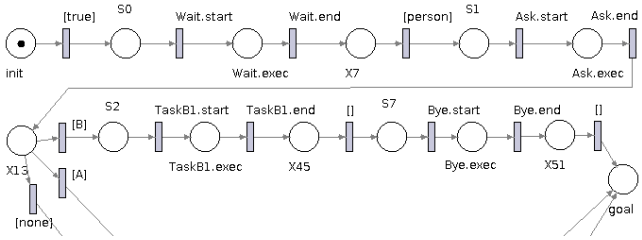

Figure 3: Portion of PNP for **Example1** before ERs.

can be actually checked at run time to determine the current outcome of actions and thus the correct successor state from which the execution should continue. Consequently, it is not necessary to explicitly observe the MDP states $s_0, \ldots, s_7$ during the execution of the PNP.

Figure 3 shows a portion of the PNP generated for the optimal policy of **Example1**, before the application of the Execution Rules. This PNP is formed by 25 places, 27 transitions and 54 edges.

Generating the final PNP including ER

Execution rules are implemented by automatically adding interrupts in the PNP generated from the optimal policy. For each rule $\mathbf{if}\ \ (\phi)\ \ \mathbf{during}\ \ a\ \ \mathbf{do}\ \ \{\sigma;\rho\}$, the following steps are performed.

For every occurrence of the action a in the current PNP, an interrupt (i.e., a transition) labeled with the condition ϕ is added. The sequence of actions contained in σ are added to the interrupt transitions according to the PNP action formalism. Finally, one of the following operations is executed depending on the value of ρ:

- *restart_action* : the sequence of actions σ is connected to the initial place of action a (i.e., the action a is restarted);
- *skip_action* : the sequence of actions σ is connected to the final place of action a (i.e., the action a is skipped);
- *restart_plan* : the sequence of actions σ is connected to the initial place of the plan (i.e., the entire current plan is restarted);
- *fail_plan*: the sequence of actions σ is connected to a final place labeled with 'fail' (i.e., the current plan is terminated with a failure state)

Figure 4 shows a portion of the PNP for **Example1** augmented with the application of the Execution Rules. This PNP is formed by 34 places, 39 transitions and 78 edges.

In this very simple example, the size of the PNP is still limited and in principle this plan can be manually written by an expert user with small effort. However, in the other examples in which the formalism has been tested, the size of the PNP is significantly bigger and manual writing and management becomes impractical even for expert users. The final PNP generated by this process that is then executed by the PNP executor, as described in (Ziparo et al. 2011).

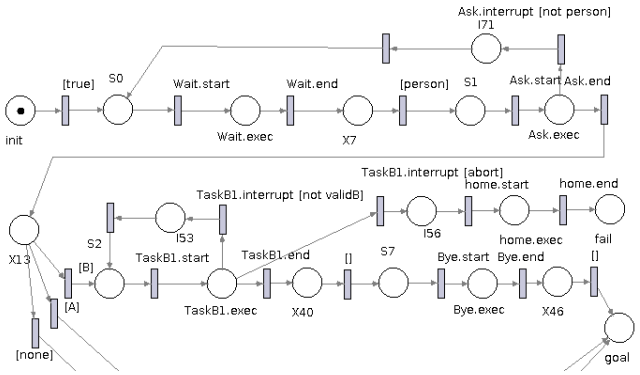

Figure 4: Portion of PNP for **Example1** after ERs.

Implementation

The proposed framework has been fully implemented and tested both on simulated robots and on real robots. The developed software is fully available as open source in the already mentioned PNP and COACHES web sites. PRU+ can be defined as an XML file, while ERs are defined through a text file with a simple sintax reflecting the rules. While in the previous sections we provided the full process applied to **Example1**, in this section we provide additional details of the implementation and execution in real environments with real robots. More details, videos, and full specifications of the PRU, ER, and PNP used in these tests are provided in the COACHES web site.

Shopping mall experiments

Several runs of 2 different tasks modeled with the formalism described in this paper have been performed at the "Rives de l'Orne" shopping mall in Caen on February 8th, 2016. **Example2** provides a description of one of these tasks.

During the experiments we have experienced some inconveniences that we were able to fix on the fly by just adding the proper execution rules. First, the robot was trying to just avoid people in the environment when moving to a location, without exploiting the fact that encountered people can be interested in interacting with the robot. An ER was thus added to greet a person when the robot encounters him/her. In some cases, the current task is interrupted and the encountered person is allowed to start an interaction with the robot. Another situation arised when the robot reached a shop location and executed a default behavior of turning back to the center of the mall. This behavior is not adequate however when a person is close to the shop. Also in this case an ER allowed to fix this inconvenience. **Video_ Rive_Feb16** in the COACHES web site show the execution of these tasks.

Office experiments

In the experiments made at DIAG Sapienza University, the robot performed several assistance tasks for visitors, as described in **Example3**.

In these experiments, ERs have been useful to improve the robot behavior in approaching people. For example, if

the robot has to reach a location and the user moves towards the robot before it has completed the action, blocking its path to the target position, the action will not end and the interaction with the person will not be started. We thus added an ER to skip the current action whenever the robot is close enough to its target position and the person is in a social distance to the robot. Additional examples are shown in **Video_DIAG_Nov15** available in the COACHES web site.

Computational complexity

In this section, we discuss a computational analysis of the proposed algorithms.

The complexity of state generation mainly depends on the number of state variables to instanciate and on the number of layers and outcomes in the PRU. For instance, a PRU containing $L = 3$ levels, with $A = 2$ alternative modules having each $B = 5$ outcomes, would generate $S_c = L \times A \times B = 30$ different MDP states for each instance of the variables. If there are 2 variables with 3 possible values on each layer ($C = 3^2 = 9$ possible combinations), then the final MDP has $S = S_c \times C = 270$ states. Since each layer has its own set of variables, this exponential growth is limited by the number of necessary variables for each level. Building the reward and the transition functions depends on the number of states where each action can be used and on the number of outcomes each action can lead to. Adding observations to the MDP policy to build an executable plan is linear in the number of states, since only one action is used in each state. This complexity may be further reduced if the initial state of the system is known, since only a subset of the states is typically used from a given situation.

In summary, the complexity for computing the optimal plan for a given PRU is $O(S \cdot A \cdot B \cdot H + S \cdot A \cdot B + L \cdot A \cdot B \cdot C) = O(L \cdot A^2 \cdot B^2 \cdot C \cdot H)$. With L the number of layers, A the average number of modules per layer, B the branching factor (the average number of outcomes per module), C the average number of state-variable combinations per layer, and H the horizon.

The cost of generating the PNP is proportional to the number of MDP states. As described in the previous section, the policy to PNP transformation algorithm builds a PNP action for any state that is described in the policy and adds a number of transitions to other MDP states. Thus the size of the generated PNP is linear in the number of MDP states.

The application of the ERs is linear in the number of occurrences of actions and in the number of rules. In the worst case (arising when all the rules apply to all the actions), the complexity is proportional to the product of the number of occurrences of actions in the PNP and the total length of all the rules. However, this situation is very uncommon. If, for each action, only a few rules apply and the length of the recovery procedures in the rules is limited (that is a more realistic case), then this complexity can be considered proportional to size of the original PNP without ER, thus again proportional to the number of MDP states in the policy.

Conclusion

In this paper we presented a practical framework for generation and execution of robust plans for service robots. Our framework integrates compact representation of complex tasks using Progressive Reasoning Units (PRUs); a robust mathematical tool for decision-theoretic planning techniques based on Markov Decision process (MDP) and execution model to support failures by transforming the policy into a Petri-Net Plan (PNP). This general framework has been implemented and succesfully tested with real robots in shopping mall and office environments. The main future directions of this work are: 1) to test its scalability to more complex tasks involving complex user interactions, 2) to apply machine learning techniques for long-term adaptation and evolution of the PRU, 3) to augment the robot knowledge with common-sense knowledge.

References

Bellman, R. 1957. A markovian decision process. *Indiana Univ. Math. J.* 6:679–684.

Boutilier, C., and Poole, D. 1996. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, 1168–1175.

Boutilier, C.; Dearden, R.; and Goldszmidt, M. 2000. Stochastic dynamic programming with factored representations. *Artificial Intelligence* 1-2(121):49–107.

Cheuk, A. Y. W., and Boutilier, C. 1997. Structured arc reversal and simulation of dynamic probabilistic networks. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97)*, 72–79.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated planning - theory and practice*. Elsevier.

Hanheide, M., et al. 2015. Robot task planning and explanation in open and uncertain worlds. *Artificial Intelligence*.

Hoey, J.; St-aubin, R.; Hu, A.; and Boutilier, C. 1999. Spudd: Stochastic planning using decision diagrams. In *Proc. of 15th Conference on Uncertainty in Artificial Intelligence*, 279–288. Morgan Kaufmann.

Howard, R. 1960. *Dynamic Programming and Markov Processes*. Published jointly by the Technology Press of the Massachusetts Institute of Technology and.

Koller, D., and Parr, R. 2000. Policy iteration for factored mdps. In *Proc. of 16th Conference on Uncertainty in Artificial Intelligence (UAI)*, 326–334.

Mouaddib, A.-I.; Zilberstein, S.; and Danilchenko, V. 1998. New directions in modeling and control of progressive processing. In *ECAI*, volume 98.

Puterman, M. L. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Ziparo, V.; Iocchi, L.; Lima, P.; Nardi, D.; and Palamara, P. 2011. Petri Net Plans - A framework for collaboration and coordination in multi-robot systems. *Autonomous Agents and Multi-Agent Systems* 23(3):344–383.