Integrating Conditional Planning and Robust Execution in HRI

Candidate:
Valerio Sanelli

Thesis Advisor:
Luca Iocchi

Sapienza
Università di Roma

Contents

Chapter 1

Introduction

This chapter present the thesis, with motivations and goal of this work

1.1 Motivating Example: COACHES

This example is taken from [14]
The input to the problem is a qualitative information about the location of the robot and the person. The output is the executed behaviour which maximizes the global actual reward, which is to satisfy the person needs. The robot could execute some actions and provide a some services. These are:

- move to a location
- approach a person or a group of people
- perform advertisement
- bring a person to a location
- help people carry things

For the sake of clarity, we introduce a simple example in which we focus on the aspects of the interaction and we do not consider other uncertainties coming from the robotic system (issues in localization and navigation, face detection, etc). Consider the situation in the following image:

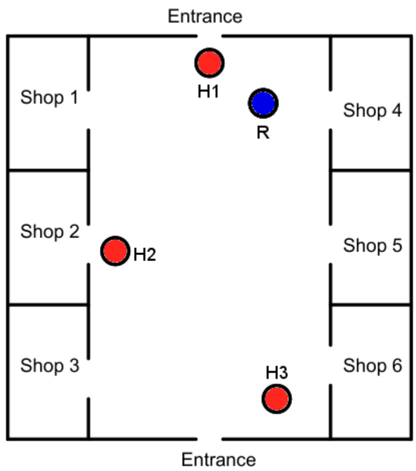


Figure 1.1: A snapshot of the mall, where R is the Robot and H_i a person

The qualitative information about the people locations indicates that there is a person in front of shop 2, another wandering nearby shop 6 and a third person in front of him. So the set of actions could be represented as

$$A = \{sense\_person, move, bringto, approach\_person, approach\_group, advertise, carry\_bags, wait, ask\_clothes, ask\_food, ask\_object, bye\}$$

where X could be a generic location or the position of a shop. The robot is able to represent knowledge about the environment and try to guess people wishes. For instance, based on the people locations, he tries to infer from its knowledge if a person is interested in one particular shop or services and could provide an advertisement. He maintains an internal database of all the goods for sale, divided by category, so he can better individuate people needs. If we recap the formula introduced in 4, we have that:

$$X = Y \cup Z$$

where X represents the complete state variables, Y are the environmental state variables while Z are the epistemic state variables.

$Z = \{food, clothes, object, WC, help, none\}$
$Y = \{time, person\_atX, group\_atX, RLocation, foodCategory_i, genericGood_j, clothingStore_k\}$

where $food, clothes, WC, help, none$ represent the person interest in a service, while $FoodCategory_i, GenericGood_j, ClothingStore_k$ represents the specific required goods.
Consider person $H2$, that is standing in front of shop 2.
Lets assume that

- shop 2 and shop 4 are clothes store. The first is a sportive clothes store, while the second is an high fashion store.
- shop 3 is a WC
- shop 1 is a market
- shop 5 and shop 6 are a chinese and an italian restaurant.

This situation is summarized in the following figure:

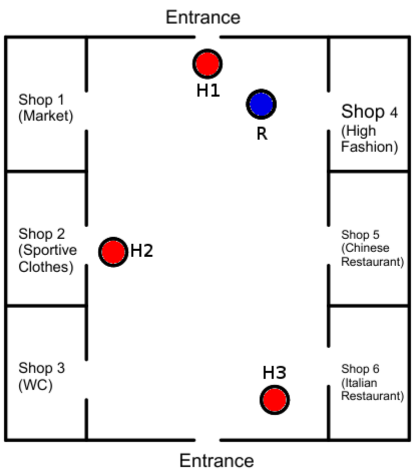


Figure 1.2

We can reasonably think that the person is looking for a specific article inside this shop, or another clothing shop, and that he's not certainly hungry.
So a possible plan for $H2$ would be:

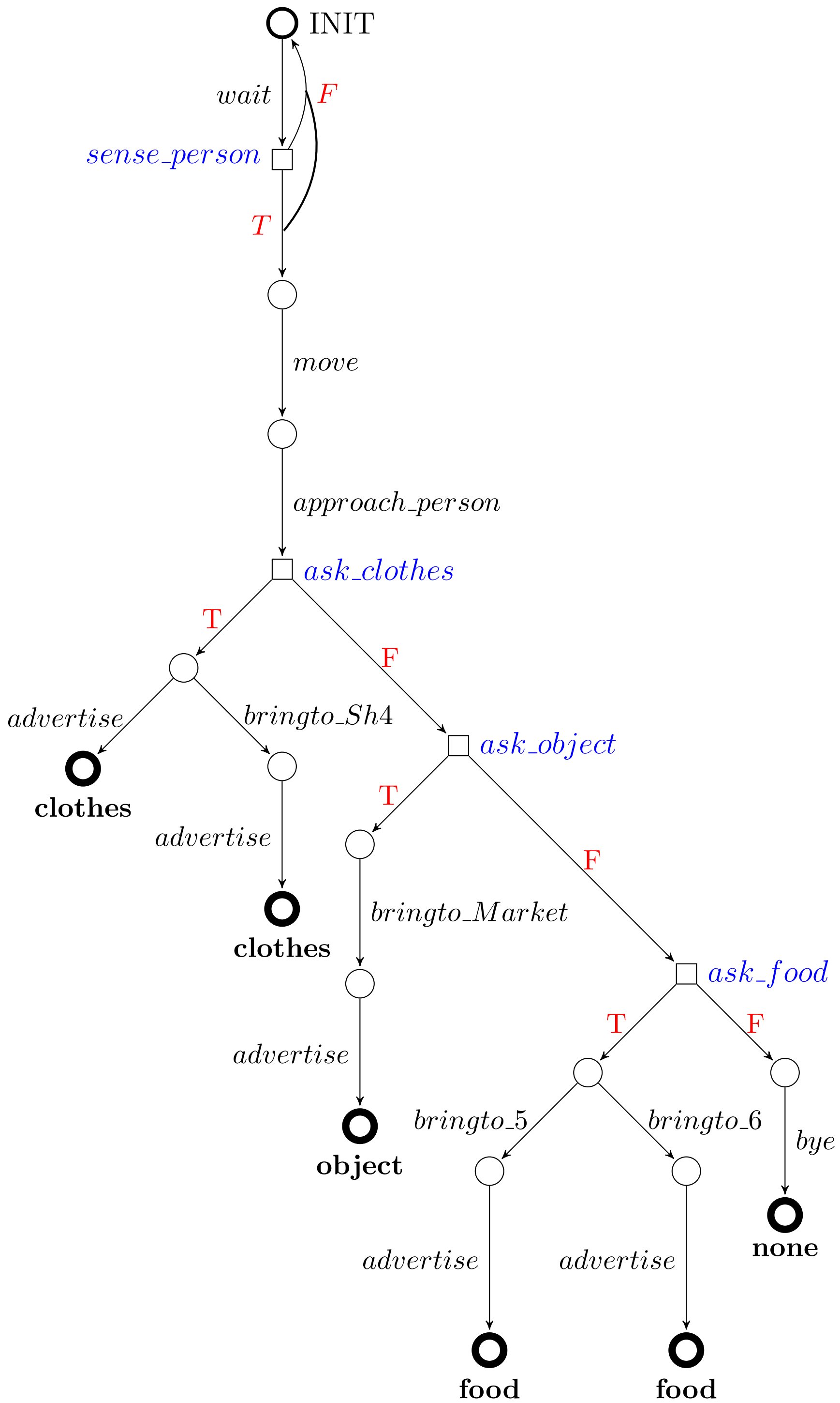


Figure 1.3: a plan for H2

Where the squares ($\square$) represents sensing actions.
A **sensing action** is an action that incorporates an observation (a boolean evaluation in this case) and it is used by the robot to characterise its (partial) knowledge of the environment. We assume that a state in which at least one of the Z properties holds is a goal state.

I will now introduce some formalisms for planning problem representation.
A more detailed discussion about this formalisms and others will be presented in 3.1.

1.2 Planning: some hints

Once a problem is characterized, we must define the techniques necessary to solve it.
Planning is "*the task of coming up with a sequence of actions that will achieve a goal*" [24].
When talking about complex problems, classical search-based planning approaches cease to be useful as any algorithm easily get lost in the possible huge state space. In this cases, a language for represent the problem represent a better solution. The convenience of using a formal language is that it guides the planning algorithms through the search of the solution by exploiting the problem logical structure.
On other hand, we must deal with some contraindications:

- **Uncertainty.** How to deal with uncertainty in action effects and in the state.
- **Goal existence.** Does a goal exist? Is it possible to compute it?
- **Expressivity vs Complexity.** Can we represent the problem? Is it possible to compute a solution?

Try to address all this questions is behind the scope of this thesis.
Instead, I will show how to deal with sensing uncertainty and how to generate robust plans, that are plans that execute safely with limited or no needs for

replanning.

In chapter 2 I will present the context of this work and the state of the art.
In chapter 3 I will review some of the formalisms in literature, showing their pros and cons in terms of representation expressivity and computation complexity.
In chapter 4 I make some consideration about the analysis done in the precedent chapter and I will introduce the used approach.
In chapter 5 the tools used for the developing of the thesis are presented.
In chapter 6 the system is presented, with the architecture and algorithms descriptions.
Finally, in chapter 7 and 8 some problem examples and considerations about the work are done.

Chapter 2

Related Works

This chapter reviews the state of the art and identifies the field of interest of this work.

2.1 Social Robotics

Social Norms: "*Norms are cultural products (including values, customs, and traditions) which represent individuals' basic knowledge of what others do and think that they should do.*"[1]

While the previously stated definition of social norms fits very well in an environment populated exclusively by humans, when robots are added to the context it can be very difficult to define what are social norms. In particular, what are the desirable characteristics that a robot must have in order to be defined "social"?
Through recent years we have seen an increasing interest with respect to robotics and artificial intelligence. While the first "intelligent" personal assistants have made their appearance thanks to smartphones diffusion, robots for the major part are still relegated in a factory and seen as fictional characters. Even in the industrial context, where we can see some examples of human-robot cooperation, the interaction is far from being "social" since the

[1] `https://en.wikipedia.org/wiki/Norm_(social)`

human is seen as an obstacle to avoid and the interaction is programmatic. However, the precedent definition give us some hints about the fundamental question:
"basic knowledge of what others do and think that they should do."
The importance of developing a robot that is human aware reside in its ability of model human state of mind, especially when we talk about robots that must operate in human environments.
We have started to see some examples of robots operating in public environments, and we always come to an end: people are for the most part frightened by robots as they do not know what to expect from it. Thus, when developing robots for operating beside humans, it is important that the robot has the ability of drive the interaction, i.e. be an active (not passive) actor. Beside from have an "attractive" design, a robot of this kind must promote different modalities for interaction and maintain at any time a snapshot of person state of mind. The proactivity helps the robot to reduce is uncertainty about the human and ensure, with appropriate behaviours, that the person is always comfortable when in presence of the robot.
In general we can state that social robotics norms that are norms of human being imitated/reproduced by the robot in order to encourage collaboration and that are socially acceptable, depending also on the cultural context.

2.1.1 Human Robot Interaction (HRI)

HRI: *"Understand and shape the interactions between one or more humans and one or more robots"* (Goodrich&Schultz 2007)

Human-robot Interaction study the interaction between human and robots. Its a multidisciplinary field, and attracts researcher from robotics, AI, cognitive psychology, and many more. Its main objective is to develop models for natural and social interaction between humans and robots. A central idea in HRI research is that humans expect from a robot to exhibit a human-like behaviour when interacting with them, so they can feel comfortable as when interacting with a person. Is is important to underline that a social robot must interact with non-expert users, that don't know what to expect when dealing with it.

Nowadays the most prominent context in which a social robot could be implied are:

- Entertainment
- Public Environment Assistance (museum, shopping mall, etc.)
- Shared Tasks (like assisted assembly)
- Healtcare, elder people assistance, etc.

2.2 Some Examples

2.2.1 Industrial Cases (No HRI)

Those are the cases in which robot and human are both aware of the plan and they cooperate deterministically in order to achieve the goal. There is no HRI as the human is seen as another agent in the environment.
Examples: *assisted assembly, logistic, space.*

2.2.2 Almost Social Cases (HRI Not Mandatory)

In this cases the interaction between the human and the robot is minimal. The human is a target and the interaction is a secondary objective. No representation of the human state is required.
Examples: *patrolling, search & rescure.*

2.2.3 Strictly Social Cases (HRI Mandatory)

Those cases represents the most significant for this work. In this case the interaction is the principal objective and the robot itself could start the interaction. A representation of the human state is maintained.

Examples: *Project COACHES (`https://coaches.greyc.fr/`), Amelia the Assistant (`http://www.ipsoft.com/amelia/`), assistance task.*

Chapter 3

Plan Representation

In this chapter I will discuss about some of the most diffused formalisms and languages for plan representation.

3.1 Formalisms

3.1.1 STRIPS (Classical Planning)

STRIPS [12] stand for STanford Research Institute Problem Solver.
It is a formal language (and a planner) used to describe deterministic, static, fully observable domains.
In STRIPS the initial state is assumed to be known with certainty and any action taken at any state has at most one successor (no multiple outcomes).
It is not possible to describe indirect effects of actions, as action preconditions/postconditions are expressed simply as conjunction of literals (positive for preconditions).
More formally, a STRIPS instance is a quadruple $< P, O, I, G >$, where:

- P is a set of conditions (propositional variables).
- O is a set of operators (actions). Each action is itself a quadruple with four set of conditions specifying which conditions must be true/false for the action to execute and which conditions are made true/false after the execution

- I is the initial state, specified by a set of conditions that are initially true (all others are false).
- G specify the goal conditions. This is a pair of two set specifying which conditions must hold true/false in a state in order to be considered a goal.

A plan computed by STRIPS is a linear sequence of actions from initial state to goal state with no branches (i.e. no conditional effects).
This makes STRIPS very fast: the search is done off-line and then the found plan is executed on-line with "eyes closed".
On other hand it cannot deal with uncertainty in both sensing/knowledge, like unreliable observations, incomplete/incorrect information or multiple possible worlds. Changes to its knowledge must always come from the robot itself and that make it inappropriate for interaction.
Not only, when the model became too complex the time to compute a plans grows exponentially, thus making impossible to benefit of its fastness.
An expanded version of STRIPS is Action Description Language (ADL) [20]which expand STRIPS with some syntactics features:

- state representation: allows also negative literals.
- action specification: add conditional effects and logical operators for express them.
- goal specification: quantified sentences and disjunctions.

3.1.2 PDDL

PDDL [18] is the standard "de facto" for planning domains description. It generalizes both STRIPS [12] and Action Description Language (ADL [20]). It is intended to describe the "physics" of a domain in terms of predicates, possible actions and their effects.
States are represented as a set of predicates with grounded variables and arbitrary function-free first-order logical sentences are allowed to describe goals. Action effects are the only source of change for the state of the world and

could be universally quantified and conditional. This makes PDDL asymmetric: action preconditions are considerably more expressive than action effects.
The standard version of PDDL divide the planning problem in two parts: domain description and problem description. Domain description contains all the element that are common to every problem of the problem domain while the problem description contains only the elements that are specific for that instance of problem. This two constitute the input for the planner. The output is not specified by PDDL but it is usually a totally/partially ordered plan (a sequence of actions that may be also executed in parallel). The domain description consist of:

- **domain-name**.
- **requirements**. declares to planner which features are used. some examples are strips, universal-preconditions, existential-preconditions, disjunctive-preconditions, adl. When a requirement is not specified, the planner skips over the definition of that domain and won't cope with its syntax.
- **object-type hierarchy**.
- **constant objects**.
- **predicates**.
- **actions**. are operators-schemas with parameters, that will be grounded/instantiated during execution. They had parameters (variables to be instantiated), preconditions and effects. Effects could be also conditional.

The problem description is composed by:

- **problem-name**.
- **domain-name**. to which domain the problem is related.
- **objects**. the definition of all possible objects (atoms).

- **initial conditions**. conjunction of true/false facts, initial state of the planning environment.
- **goal-states**. a logical expression over facts that must be true/false in a goal-state.

I refer to [18]for a more formal definition of the syntax. Through the years PDDL has received several updates (the last version is PDDL 3.1) and extensions. This enrich PDDL with useful properties that are not present in its standard version, like the possibility of represent uncertainty or probabilistic outcomes for actions. The most notable are:

- PPDDL [27]is a probabilistic version of PDDL.
  It extends PDDL 2.1 with probabilistic effects, reward fluents, goal rewards and goal-achieved fluents. This allows to realise Markov Decision Process (MDP) planning in a fully observable environment with uncertainty in state-transitions.
- RDDL [25]introduce the concept of partial observability in PDDL. It allows to describe (PO)MDP representing states,actions and observations with variables. Semantically, RDDL represent a Dynamic Bayes Net (DBN) with many intermediate layers and extended with a simple influence diagram utility node reresenting immediate reward.
- KPDDL [15]add to PDDL some constructs to represent the incomplete knowledge of the agent about the environment. It will be discussed in the upcoming sessions.

3.1.3 $ALCK_{NF}$

It is an autoepistemic logic, an extension of ALC with the add of the epistemic operator **K** and default assumption operator **A**. It is presented in [9]. The logic ontology assumes the existence of *concepts* and *roles*.
A concept represent a class of *individuals* while the roles model the relations between classes.

The interpretation structure of Default logics is a Kripke structure, with labelled nodes and edges. Each node represents an individual and is labelled with the concept corresponding to the individual while each edge is a role. This structures could be interpreted as a dynamic system where the individuals are the states of the system that holds the properties valid in that state (like fluents in situation calculus). Edges represent transactions between system states and are labelled with the action that causes the change in state. So each node/individual represent an *epistemic state* of the robot so that there is an edge between two nodes if the associated action modifies the source epistemic state in the destination epistemic state.
There are two types of actions: ordinary actions and sensing actions.
Ordinary actions are deterministic actions with effects that depend on the context they are applied, while the sensing actions do not modify the environment but the knowledge of the agent since they evaluate boolean properties of the environment.
Thanks to the latter it is possible to explicitly model the effects that this type of actions has on the knowledge and to characterise plans that the agent could actually execute.
It is also possible to specify actions that can run in parallel. Parallel actions in a state can be executed iff each could be executed individually in that state and the effects of the actions are mutually consistent.
For what concerns the Frame problem, the language use default axioms to represent the persistence of properties and frame axioms that express the causal persistence of rules.
Epistemic operators guarantee that the generated plan is actually executable, unlike first-order logic, and sensing actions produces ramifications that could help to reach the goal. This type of plan, with sensing plus ordinary actions, is called conditional plan.

3.1.4 KPDDL

KPDDL [15]is an extension of PDDL based on the logic $ALCK_{NF}$.
Its aim is to augment the planning domain with the (incomplete) knowledge

of the agent about the environment.
KPDDL introduces several specifications:

- *sense terms*. effects of actions that helps to disambiguate the true state of the environment.
- *non-inertial terms*. specifies non-inertial properties.
- *strongly-persist terms*. specifies strongly persistence properties.
- *formula-init term*. is the specification for an incomplete initial state.

The fundamental contribute of KPDDL from the semantic viewpoint is the interpretation of a dynamical system through its epistemic state. An epistemic state represents what the agent knows about the world, that could be different from what is true in the world, and the planning is realised by modelling at this meta-level.
A planning problem can be specified as:

$$KB \cup \phi(init) \models \pi_{\psi}(init)$$

where KB is the set of axioms representing the planning domain, $\phi(init)$ is the assertion representing the initial state and $\pi_{\psi}(init)$ is an $ALCK_{NF}$ assertion that is true in the initial state iff executing the plan agent knows ψ. The associated planner produces conditional plans in the form of conditional trees (or direct acyclic graphs), in which the root is the initial states, the leaves are goal states and the branches are different results of sensing. Conditional plans are distinguished in *weak/strong*, whether or not are present leaves that are not goal states and from which a goal state could not be reached. Plans could be also *cyclic*, in which case the termination is not guaranteed. Formally, a plan is defined as quadruple $P =< G(V, E), I, F_G, F_F >$ where:

- G is a direct graph, with V nodes and E edges.
- $I \in V$ is the initial state.
- $F_G \subseteq V$ is the set of final states in which the goal is achieved.
- $F_F \subseteq V$ is the set of final states in which the goal is not achieved.

3.1.5 Answer Set Programming (ASP)

Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems. [16][1]
It's declarative in the sense that it describe the problem, not how to solve it. In particular, a problem instance I is translated into a non-monotonic logic program P and passed to an ASP solver which will search the stable models of P. From these, the solution of I is extracted.
ASP programs are composed of Prolog-like rules, but it uses a different computational system. In fact, ASP solvers are based on DPLL algorithm, that in general guarantees the termination (unlike SLDNF resolution in Prolog).

A rule has the form:

```
<head> :- <body>
```

where $< head >$ is a disjunction of positive/negative literals, while $< tail >$ is a conjunction of positive/negative literals.
The symbol :- is dropped if $< body >$ is empty; this are called *facts*. Rules with an empty $< head >$ are called *constraints* and are used to reduce the number of possible solutions (stable models). These are useful to express action preconditions like in other formalism (for instance 3.1.2).
ASP includes two form of negation: strong ("classical") negation and negations as failure. This allow to represent defaults, useful in order to express the closed world assumption and characterize a solution to the frame problem (inertia of the world: *"everything is presumed to remain in the state in which it is"*).

I will now introduce some examples from [16]to illustrate ASP properties:

```
(1) p :- q.
(2) cross :- not train.
(3) cross :- -train.
```

(1) is an example of a classic Prolog like rule. It can be interpreted as $q \rightarrow p$, that is *"if **q** hold, than **p** is in the stable model"*.

[1] https://en.wikipedia.org/wiki/Answer_set_programming

(2) represent an example of negation as failure. This rule represent the knowledge: *"it is safe cross if there are no information about an approaching train"* (not a good idea indeed).
(3) is preferable in this case, since uses the strong negation: *"it is safe cross if it is known that the train is not approaching"*.
If we combine both negations it is possible to express the closed world assumption (*"a predicate does not hold unless there is evidence that he does"*):

```
-q(X,Y) :- not q(X,Y), p(X), q(Y).
```

interpreted as *"**q** does not hold for a pair of element of **p** if there are no evidence that it does"*. This allow to include also negative facts about q, so to specify the closed world assumption for some predicates and leave the others with the open world assumption.
Finally, it is possible to express time explicitly with constants. This feature is used to define a metric for plans and for express the *"frame default"*:

```
p(T+1) :- p(T), not -p(T+1), time(T).
```

*"**p** will remain true at time **T+1** if there is no evidence that becomes false"*.
ASP programs are often organized in three sections: generate, test and define. The *"generate"* section describes the set of possible solutions while the *"test"* section is used to discard bad solutions. The *"define"* section defines auxiliary predicates used in the constraints. The order of the rules in an ASP program doesn't matter.

ASP has been widely used in the service robotics context, both in simulated or real scenarios. Some examples are [10] and [7].
However, ASP is not well suited for modelling/reasoning about uncertainty in a domain. Incomplete information cannot be handled correctly due to the fact that stable model semantics is restricted to boolean values. Not only, there is no notion of probability and thus all stable models are equally probable. Some extension to ASP with probabilistic information has been proposed, like in [26]where Markov Logic Networks (MLN) are used.

3.1.6 Situation Calculus

Situation Calculus [17] [21] is a logic formalism used to represent and reason about dynamical systems. It allows to express qualitative uncertainty about the initial situation of the world and the effects of actions through disjunctive knowledge.

The state is represented as a set of first-order logic formulae. The basic elements are actions, fluents and situations.

A *Situation* is a first order term denoting a state and is characterised by the sequence of actions applied to the initial state in order to reach it.

Fluents are functions or predicates that have as last argument the situation of application and represents properties of a particular situation. Actions may also be parametrized with variables. A fluent differs from a (normal) predicate or function symbol as its value may change from situation to situation.

The binary function symbol $do(\alpha, s) \rightarrow s'$ represent the result of apply action α on a situation s, where s' is the resulting situation.

The binary predicate symbol $Poss(.,.)$ represents the possibility of executing an action. For instance, $Poss(\alpha, s)$ is true iff is possible to apply α in situation s.

A Planning Problem in the Situation Calculus is then defined as

$$\mathcal{D} \models \exists s.Goal(s)$$

where $\mathcal{D}$ is a theory of actions, which consists of:

- **Axioms for the Initial Situation**. A formula that represent properties verified in the initial situation S_0.

- **Unique Name Axioms for the Actions**.

  $$do(\alpha_1, s_1) = do(\alpha_2, s_2) \Rightarrow \alpha_1 = \alpha_2 \wedge s_1 = s_2$$

  Unique name for actions and situations.

- **Precondition Axioms**. They codify necessary properties in order to execute actions.

- **Successor State Axioms**. For each fluent establish if is true after the execution of an action. Those axioms solve the frame problem as they enforce the persistence of those properties that are not involved or changed by the current actions, describing the causal laws of the domain with an axiom per fluent.

A plan is a sequence of actions that brings the system from the initial situation to a situation that satisfies the condition *Goal(s)*, whenever the formula representing the goal is satisfiable.

3.1.7 Markov Decision Process (MDP)

A Markov Decision Process is a stochastic machine state representation of a dynamic system, in which every transition has assigned a probability. This process is 'Markovian' since Markov property hold, that is the future state depends only on the current action and state and not from the past states.
The actions to take in every state are specified by a utility function, that guide the agent through the goal and represent the utility of being in a state or take an action. This function specifies a preference over the paths of the graph.
Actions are the only source of change and are assumed to be instantaneous. In MDP there is no explicit modelling of time. Nor the world dynamics or the reward function depend on absolute time.
A policy $\pi : S \to A$ is a total function that specifies for each state the action to take. The planning problem is then an optimisation problem in which the objective is to find the policy that maximises the expected utility associated with the states and the action. The goal is represented as a reward and is a deterministic function of the current state.
This it is a quantitative representation, unlike the previously introduced formalisms, in which the uncertainty is represented through probabilistic means (see also 3.1.8).

More formally, an MDP is a 5-tuple $\sum =< S, A, P(.,.), R(.,.), \gamma >$ where:

- S is a finite set of states

- A is a finite set of actions
- $P_a(s'|s)$ is the probability that action $a \in A$ in state $s \in S$ at time t will lead to state $s' \in S$ at time $t+1$. Notice that $\sum_{s' \in S} P(s, a, s') = 1$.
- $R_a(s'|s)$ is the immediate reward received after the transition from s to s'.
- $\gamma \in [0, 1]$ is the discount factor, which weights the importance of the future versus the present rewards.

The probability of a sequence of states $< s_0, s_1, ..., s_n >$ is called history and is defined given a policy as:

$$P(h|\pi) = \prod_{i \geqslant 0} P_{\pi(s_i)}(s_{i+1}|s_i)$$

The utility function is specified by associating for each $< state, action >$ a cost/reward depending on the value that transition must have for the agent. If we define a cost function as $C : S \times A \rightarrow \mathrm{I\!R}$ and a reward function as $R : S \rightarrow \mathrm{I\!R}$, is it possible to define the utility of execute an action a in the state s as:

$$V(s|a) = R(s) - C(s, a)$$

Consequently the utility of a policy π in a state s is defined as:

$$V(s|\pi) = R(s) - C(s, \pi(s))$$

and thus for the history:

$$V(h|\pi) = \sum_{i \geqslant 0} R(s_i) - C(s_i, \pi(s_i))$$

This sum usually doesn't converge and we need a discount factor γ in order to reduce the effect of costs/rewards distant from the actual state.
Introducing γ, we can rewrite the utility of the history as:

$$V(h|\pi) = \sum_{i \geqslant 0} \gamma_i(R(s_i) - C(s_i, \pi(s_i)))$$

with $0 < \gamma < 1$. With a value of γ near 1 we promote immediate rewards over futures, while with $\gamma \cong 0$ the vice versa. From the history utility is possible to calculate the expected value of a policy utility by considering all the history that the policy induce. Given H as the et of all possible history induced by π, this expected value is:

$$E(\pi) = \sum_{h \in H} P(h|\pi) V(h|\pi)$$

a policy π^* is said to be optimal for a stochastic system Σ if:

$$E(\pi^*) \geqslant E(\pi) \quad \forall \pi$$

where π is an admissible policy for Σ. A solution to an MDP is then a policy that, once found, is applied in any state in a deterministic fashion.

3.1.8 Partially Observable MDP (POMDP)

In many cases, the assumption of complete observability for the state is too strict so Partially Observable MDP was introduced.
In this case, a set of observations is defined that characterise the observable part of the stochastic system Σ.
A POMDP is a stochastic system $\Sigma =< S, A, P(.,.) >$ as defined in 3.1.7, with the addition a finite set of observations O with probability $P_a(o|s)$ for all $a \in A, s \in S, o \in O$. $P_a(o|s)$ represent the probability of observe o in the state s after the execution of a. This is defined for every state/action and the condition $\sum\limits_{o \in O} P(o|s) = 1$ hold.
More than one observations could correspond to one state (i.e. $P_a(o|s) = P_a(o|s'))$ and thus is not possible to obtain the actual state from the observations. Analogously the same state could correspond to different observations depending on the action executed, that is $P'_a(o|s) = P_a(o|s)$. Thus observations depend both on the state and on the action. For instance, a sensing action does not modify the state of the system but could lead to a different observation.

Probability distributions over the states are called *belief states*. Let be b one belief state and B the set of all belief states. Then $b(s)$ denotes the probability for the system to be in state s.
In order for $b(s)$ to be a probability distributions both $0 \leqslant b(s) \leqslant 1$ and $\sum\limits_{s \in S} b(s) = 1 \ \forall b \in B$ must hold. Given an action a and a belief state b, the next belief state deriving from apply a in b is:

$$b_a(s) = \sum_{s' \in S} P_a(s|s')b(s')$$

Similarly we could compute the probability of observe o given the action a:

$$b_a(o) = \sum_{s \in S} P_a(o|s)b(s)$$

At last, we can compute the probability of being in s after executing a and observing o:

$$b_{a,o}(s) = \tfrac{P_a(o|s)b_s(s)}{b_a(o)}$$

Planning in POMDPs is formulated as an optimization problem over the belief space to determine the optimal policy $\pi^* : B \to A$ with respect to an utility function defined as in 3.1.7. In fact we can see at a POMDP as a continuous MDP where the continuous space is the belief space.
The expected value for the utility of a belief state b is defined as:

$$E(b) = \min_{a \in A} C(b,a) + \gamma \sum_{o \in O} b_a(o)E(b_{a,o})$$

where:

$$C(b,a) = \sum_{s \in S} C(s,a)b(s)$$

3.2 Analysis of Formalisms

The introduced formalisms and models show the essential bond between expressiveness and computational complexity. The planning problem in classic logic is decidable but the complexity varies from constant to EXPTIME depending on which characteristics are added to the language.

For instance, STRIPS is the less expressive of the presented formalisms, still deciding whether a plan exists is PSPACE-complete [4].Many restrictions can be enforced in order to make this problem decidable in polynomial time, like restricting the type of formulas or the number of pre-postconditions.

$ALCK_{NF}$ is an example of decidable formalism with a good expressiveness, since it allows to express the agent epistemic state explicitly. However the planning in its most general case falls in PSPACE complexity, thus intractable for an online approach. In this and similar cases, partial planning and restricted version of the formalism are used in order to solve the complexity problem.

In KPDDL is it possible to express incomplete information through epistemic states and define sensing actions that help to disambiguate the states. In practice, however, it was seen that even a small variation in the number of states makes the problem intractable, proving that the class of complexity is at least PSPACE (instance checking/reasoning in $ALCK_{NF}$ is PSPACE-complete [9].

Talking about POMDPs models, they are often computationally intractable to solve exactly and thus it is necessary to introduce methods to approximate solutions. Some examples are [23]and [3],where particle filters and PCA are respectively used to represent the belief state compactly or exploit its properties.

Planning with partial observability is the most difficult planning task as it falls in EXPSPACE-complete class of complexity [22]. The difficulty resides in the plan verification since the state space, in this case, is doubly exponentially bigger in the size of the problem instance with respect to classical planning case. The discussion of good heuristics and search algorithms that try to exploit the structure of such problems is not the purpose of this thesis. We will focus instead on the representation of the state space, relying on state of the art planners present in literature.

Chapter 4

Conditional Planning and Execution

In this chapter I will explain the reasons behind this work.

As we have seen in the previous chapter, different planning formalisms try to capture different aspects of real-world problems. By the way they model and abstract the world, they also produce different solution concepts.
Is not always obvious in advance which formalism is the best for a specific problem. Less expressive formalism are often preferred due to the fact that are less expensive in terms of computational power.
For instance if we consider decidable logic formalisms, $ALCK_{NF}$ is one of the more expressive and yet the planning in general it's included in PSPACE, so too complex to run on a real robot in realtime.
Classical Planning models only deterministic actions, and the planner is often embedded in a controlled execution loop that replan on unexpected outcomes.
In fully-observable and non-deterministic planners, actions are assumed to be non-deterministic but there is no uncertainty about the perception. They produce strategies that are guaranteed to reach the goal.
If we expand the domain and consider also probabilistic planners, they introduce also outcome probabilities. Their aim is to maximise a cumulative reward. They are also the most expensive and often intractable at represen-

tation and/or execution level.
Rule-bases are not manageable when they become larger. This rules are created by experts that express their knowledge through the rules and it is difficult to find all the inconsistencies and the side-effects when new rules are inserted.

This brief analysis shows that there is a compromise between expressivity and computational complexity. Epistemic logics could helps us to represent real world uncertainties.
In this case the states of the planning problem are called belief states, since they represent the uncertainty of the agent about the world. The problem with this representation is that the belief space is computationally exponentially bigger than classical state space. A good representation is thus needed. A solution is to maintain only a small group of states that represents a compact description of the epistemic knowledge of the robot.
So if we denote as Z the states that represents the epistemic knowledge and with Y the generic states, we have that:

$$|Y| + |2^Z| \ll |2^X| \text{ and } X = Y \cup Z$$

The next step is to identify a group of formalisms from the ones described in 3.1 that allow us to represent explicitly the epistemic state.
In our case, a language must supply the syntactic constructs to define sensing actions with which we can represent conditional plans. We can identify a bunch of properties that helps make this choice, as we can see from the following table:

Language/ Planner	Constr.	NonDeter.	Conc.	Sensing	CP Gen.
PDDL/FF	Yes	No	No	No	No
PDDL/ CLG	Yes	Yes	Yes	Yes	Yes
$ALCK_{NF}$	Yes	Yes	Yes	Yes	Yes
KPDDL/ KPlanner	Yes	Yes	No	Yes	Yes
ASP/Clingo	Yes	Yes	Yes	Yes	Yes
Sit. Calculus/ CONGOLOG	Yes	Yes	Yes	No	No
RDDL/Spudd	Yes	Yes	Yes	Yes	Yes

Figure 4.1: a comparison betweem different formalisms/planner.

Where the properties showed are:

- **Constr.**: the capability of define domain constraints.
- **NonDeter.**: the capability of define non-deterministic actions.
- **Conc.**: the capability of define concurrent actions.
- **Sensing**: the capability of define sensing actions.
- **CP Gen.**: the capability (of the planner) to generate conditional plans.

As we can see, PDDL (ROSPlan version), KPDDL, ASP and RDDL seems to fit our prerequisites. For this work we decide to consider the two version of PDDL (standard and ROSPlan) plus KPDDL. We will motivate this choice in the upcoming chapters.
Once we have identified the languages that matches our prerequisites, we select them to represent and solve a test domains. The last step is to translate the computed plan into a conditional plan as we will discuss later.
Summarizing, our aim is to define an architecture capable of translate different formalisms into a conditional plan that will represent a PNP, in order to satisfy some tasks requirements, while finding the best trade-off between complexity and expressivity.

Chapter 5

Foundation Tools

In the following I will illustrate the tools used for the development of this thesis.

5.1 Petri Net Plan

We need a language that can represent non-instantaneous and conditional actions.
Petri Net Plans (PNP) [29] [28]is a formalism for represent complex plans using an high-level description. It is inspired by languages for reasoning about actions, like the ones presented in 3.1, but it is more expressive and offer all the operators needed for representing conditional plans, concurrent actions and multi-agent plans. They also offer non-deterministic choice operator for representing non-deterministic choice action, useful in learning algorithms.
The execution of a PNP is very efficient, and allow the design of real-time and active behaviours. Some examples of PNPs applications are [2] [19] [11].

5.1.1 Formal Definition

A PetriNet Plan is a PetriNet augmented with a set of goal markings G. More formally, it is a tuple $< P, T, F, W, M_0, G >$ where:

- P = $\{p_0, ..., p_n\}$ is a finite set of places.

- T $= \{t_0, ..., t_n\}$ is a finite set of transitions.
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of edges.
- $W : F \Rightarrow 1, 2, 3, ...$ is a weighting function.
- $M_0 : P \Rightarrow 0, 1, 2, ...$ is the initial marking.
- $P \cup T \neq 0$ and $P \cap T = 0$

Petri Nets represent the structure of a system as a directed, weighted and bipartite graph with two type of nodes: places and transitions. With this tools we can model complex systems in terms of their internal state and its changes.

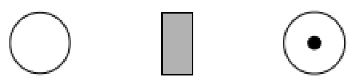


Figure 5.1: (1) a place. (2) a transition. (3) a place with one token.

Places represent the execution phases of an action, while transitions represent events. In particular each action is represented by three states: initial, execution and final state. Between them there are action starting transition, action terminating transition and there might be some action interrupts and/or control transitions.
The evolution of the Petri Net is described by the firing rule: a transition is enabled when the number of tokens for its initial places is equal at least to the number of tokens of the edge weight. When the transition fires, a number of tokens equal to the weight of the input edges is removed from the initial places and a number of tokens equal to the output edges is added to output nodes. Petri Net Plans are defined in terms of actions and operators. There are three types of actions:

1. **no-action**. is a PNP with one place and no transition.
2. **ordinary-action**. is a PNP with three places and two transitions, as described before.

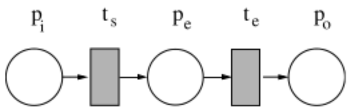


Figure 5.2: an ordinary action.

3. **sensing-action**. this are used when the action involve a property to evaluate at run time, and places/transitions are defined accordingly.

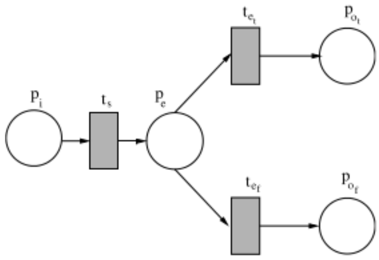


Figure 5.3: a sensing action.

Actions could be combined in order to achieve complex behaviours. These are called operators.

- **sequence**. allows the execution of two actions in sequence.

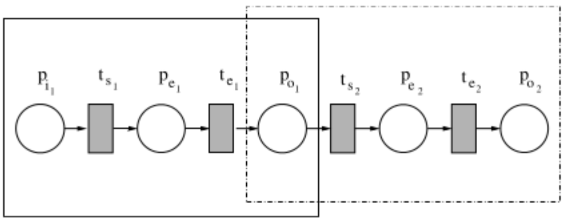


Figure 5.4: sequence operator.

- **conditional**. combining sensing action with ordinary actions is possible to obtain conditional structure.

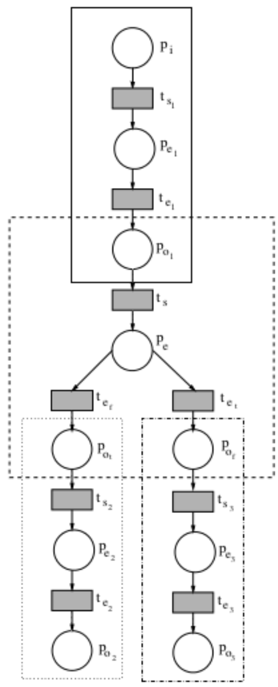


Figure 5.5: conditional operator.

- **interrupt**. allows to interrupt the execution of an action, if some condition is met, and to continue the execution on a new branch.
- **loop**. using an interrupt, is possible to define action that iterates until a condition is true.

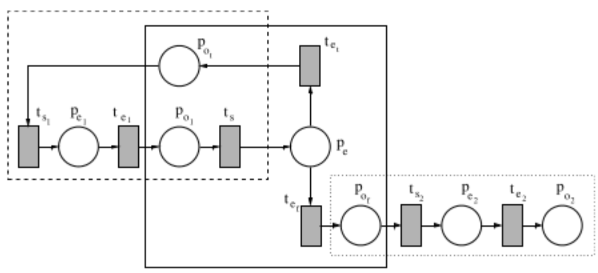


Figure 5.6: loop operator.

- **fork/join**. allows concurrency. fork generates multiple threads starting from one, while join allow the synchronization of multiple threads into one.

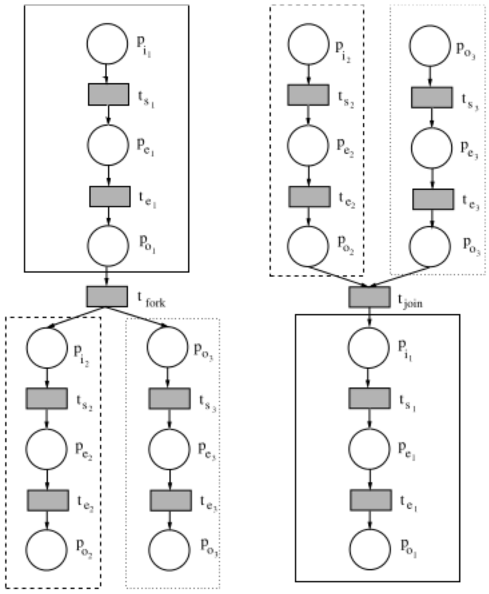
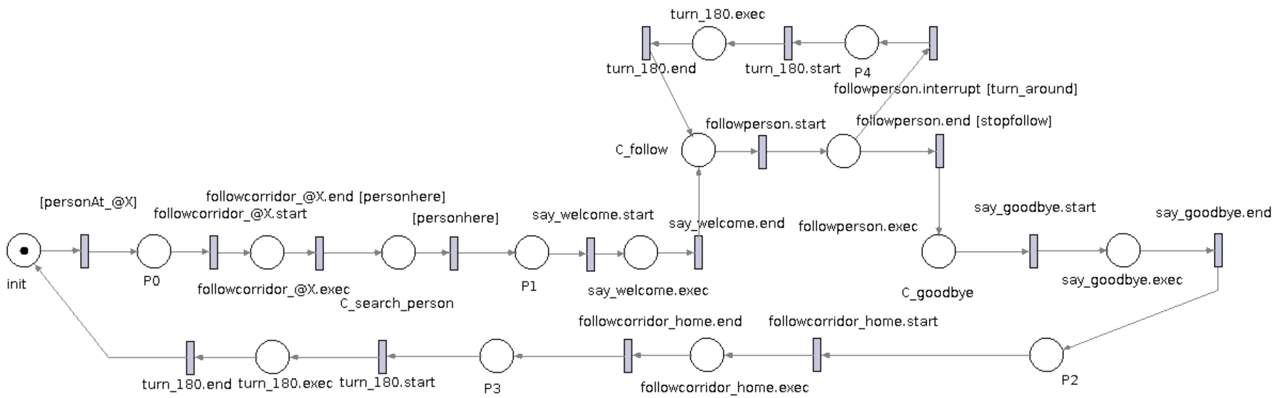


Figure 5.7: (left) fork operator. (right) join operator.

In the following page there is an example of a complete PNP, with an interrupt defined during the action *followperson*:



5.2 PNPgen

PNPgen[1] [5]is a library included the Petri Net Plans distribution responsible for the automatic generation of PNPs from a formal definition provided.
This is usually used in conjunction with a planner, which output is interpreted and executed as a PNP.
The main advantages of this approach are two:

1. The produced plan can be executed and monitored through the PNP execution engine and easily integrated into ROS thanks to the PNP-ROS bridge.
2. Once produced,the PNP can be extended of modified to deal with unexpected outcomes that may arise during the execution of an action and not considered in the domain, either manually or automatically as we will discuss later.

Furthermore, when used in combination with a formal language we can describe and generate plans that are otherwise too complex to be constructed manually (using a visual editor, i.e. Jarp). Some examples of PNPgen input are enlisted in the following. Conditional plans generation is discussed in 6.3 and 7.

5.2.1 Linear Plans

PNPgen is able to generate linear plans from a file that describes a sequence of actions to execute, separated by semicolons. An example of a valid file is:

```
goto_printer; say_hello; goto_home
```

saved in a ".plan" file. When this file is given as input to PNPgen, the following PNP is generated:

[1] `https://sites.google.com/a/dis.uniroma1.it/petri-net-plans/pnp-generation`

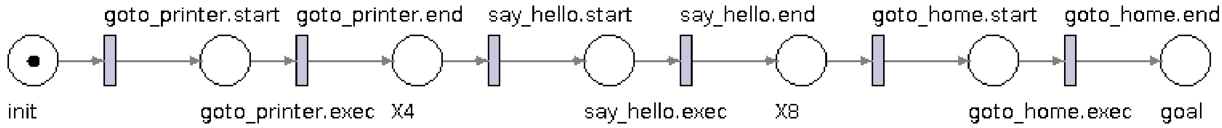


Figure 5.8: a simple linear plan.

5.2.2 Policy

Another valid format for PNP plans generation is represented by policies. Policies are transition graphs with non-deterministic and sensing actions, that enhance the robustness of the described plan. For instance, consider the following valid file describing a policy:

```
Init:  S0
Final:  S3
S0 :  goto_printer ->  [personhere] S1, [not personhere] S2
S1 :  say_hello ->  [] S2
S2 :  goto_home ->  [] S3
```

This generates the following PNP:

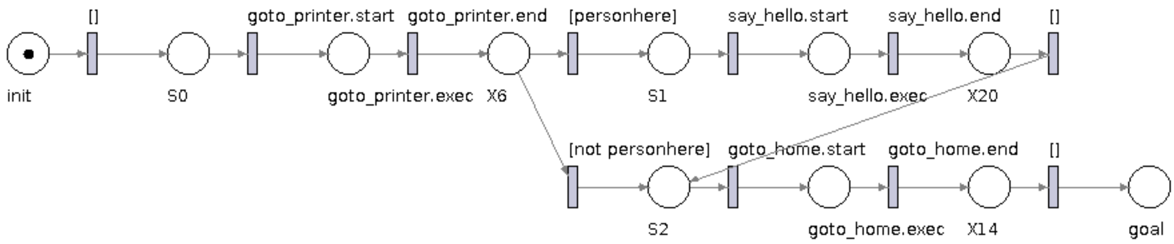


Figure 5.9: a PNP generated from a a policy.

Where we can see that the property "*personhere*" is evaluated at execution time in state 1, enhancing the plan robustness.

5.2.3 Execution Rules

In both the introduced cases, is it possible to specify an execution rules file as a support to the PNP generation mechanism.

This is a file that enlists a series of rules describing possible exogenous events that may occur during the execution of an action. These rules introduce interrupt in the specified action.

For instance, consider the following execution rule:

```
*if* (and personhere (not closetotarget)) *during* goto *do*
        say_MoveAway; waitfor_freespace; restart_action
```

this led to the introduction of a branch in the following PNP:

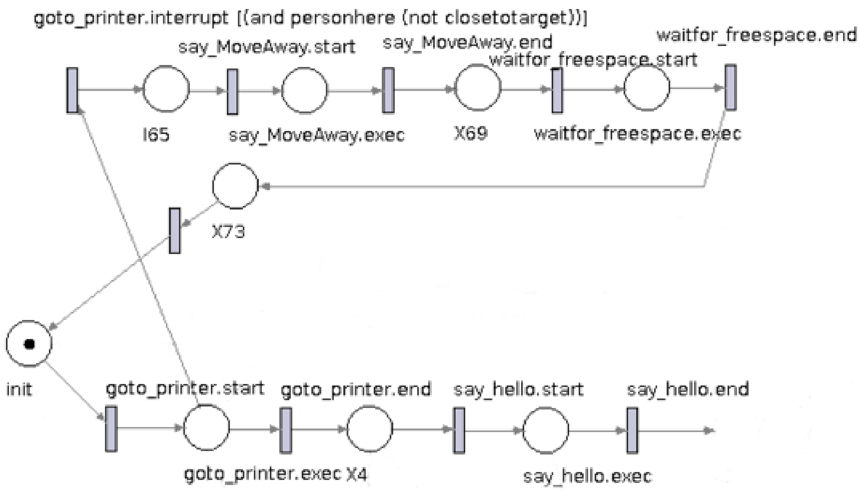


Figure 5.10: an execution rule example.

The syntax of an execution rule is:

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

Where ϕ is a boolean expression over some properties, a is an action, σ is a (possible empty) program (i.e. a sequence of actions, another PNP, etc.), and ρ is a statement that specifies how to continue the execution of the plan (like *restart action, skip action, restart plan, fail plan*). Execution rules are described in [14],where also a third method to generate PNPs from progressive reasoning unit (PRU) is discussed.

5.3 Robot Operating System (ROS)

"Robot Operating System (ROS) is a collection of software frameworks for robot software development, providing operating system-like functionality on a heterogeneous computer cluster. ROS provides standard operating system services such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, post and multiplex sensor, control, state, planning, actuator and other messages." [2]

ROS was originally developed at Stanford Artificial Intelligence Laboratory in support of the Stanford AI Robot STAIR (STanford AI Robot) project. From 2008 to 2013 the development take place at Willow Garage, a robotics research institute. Finally, from 2013, ROS become part of the Open Source Robotics Foundation.
During all this time, ROS has growth in importance and diffusion.
In fact, before ROS advent, each laboratory or research group had to write down from scratch specific code for his platform and for every task, even the more mundane. This not only is a time expensive task but restrains the diffusion of new algorithms and ideas. With the layer of abstraction offered by ROS, it is possible to write nodes for common tasks and specific hardware that are reusable and so more easy to diffuse. For instance, ROS offers packages for perception, object identification, segmentation and recognition, face recognition, planning, grasping, localization, mapping and much more. The main limitation of ROS resides in its latency since is built on top of Linux and is not a real-time OS. This led to the born of ROS 2.0, which support real-time operations, and RT-ROS, which is natively real-time.

[2] `https://en.wikipedia.org/wiki/Robot_Operating_System`

5.4 PNPros

PNPros serves as bridge between ROS and PNP.
It allows the execution of PNP under ROS using the **actionlib** module.
Like for standard actionlib nodes, it implements the client/server protocol through the definition of an Action Server, called **PNPActionServer**, and an Action Client, **PNPActionClient**.
They communicate through the use of **PNPAction** messages.
Furthermore, PNPros defines a service, **PNPConditionEval**, that is invoked to evaluate conditions at runtime.
A sketch of the PNPros architecture is depicted below:

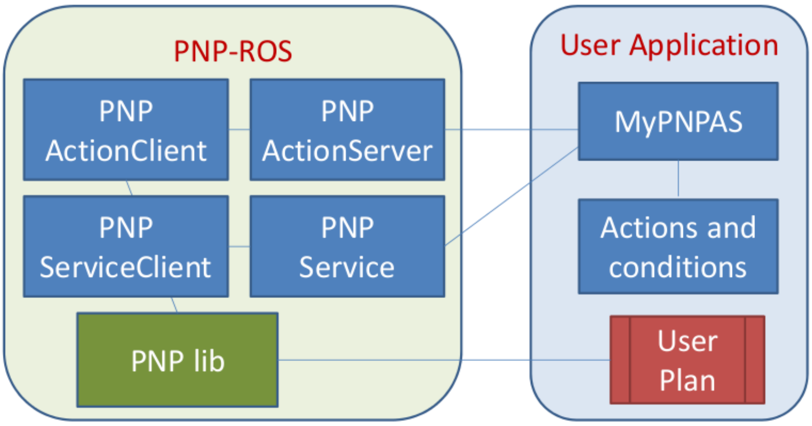


Figure 5.11: the PNPros architecture.

As we can see, the user must provide its own implementation of the server, actions and the conditions to be evaluated. PNPros actively listens a topic for plans to execute; as soon a plan is sent, it starts the execution and monitoring of the PNP. When needed, it will invoke PNPActionServer to evaluate a condition or execute an action. The actions provide the actual low-level implementation (typically in C++).
Finally, PNP plans can be written and monitored using Jarp, a graphical tool interface written in Java.

5.5 ROSPlan

ROSPlan [6][3] is a framework that provides a general method for task planning in ROS. It encapsulates both planning and dispatch and, thanks to its modular design, it is easy to modify to test new approaches to plan representation, plan execution, etc.
The main modules are the **Knowledge Base**, that stores PDDL models, and the **Planning System**, that is the interface to the planner. The Planning System itself is composed of three modules, which are replaceable: problem generation, planning, and plan dispatch.

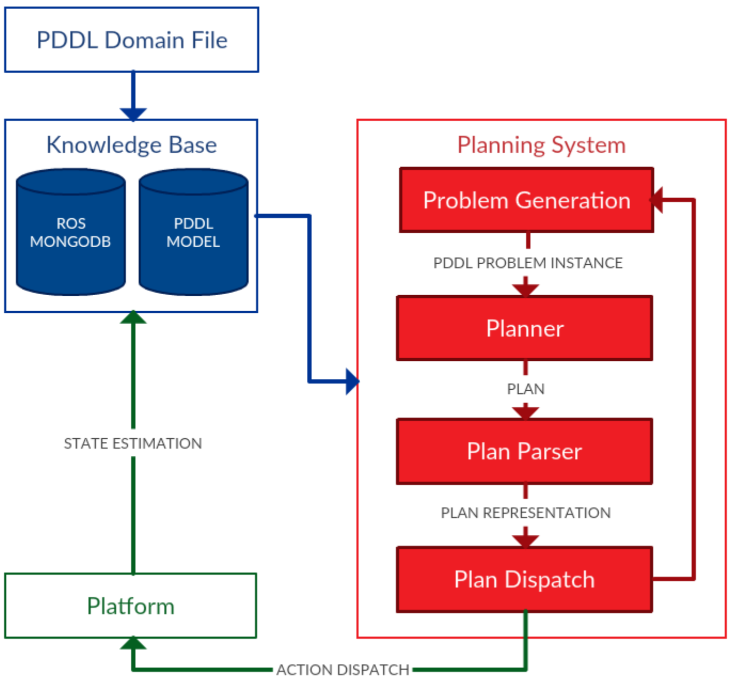


Figure 5.12: ROSPlan architecture (image taken from `http://kcl-planning.github.io/ROSPlan/documentation/`)

The Knowledge Base stores both the problem domain and problem instance, along with any information to support the planning and stored in a ROS MongoDB. The KB can be updated with ROS messages and queried on-line.

[3]`http://kcl-planning.github.io/ROSPlan/`

The Planning System:

1. Fetch information from the KB and generates the PDDL problem file.
2. Call the planner and process the output
3. Dispatches the plan through ROS messages.

It can also handle replanning due to action failure or plan invalidation. The plan responsible for the planner output parsing and dispatch is the **Plan Parser**. Two default plan parser are included in ROSPlan: one for sequential plans and another for Esterel program.
The first provides a list of action dispatch messages. The second stores the plan as an Esterel language program[4], that is represented as a DOT graph.

The standard language and planner in ROSPlan are PDDL 2.1 and POPF [8],that accepts only domains with temporal constraints and no uncertainty. Anyway, is it possible to replace the planner, the problem generation and the plan dispatch by simply providing your implementation of this modules.
This was also our case since to integrate the work of this thesis with the ROSPlan architecture some modification, in collaboration with the authors, was necessary. We will talk more about later.

[4] `https://en.wikipedia.org/wiki/Esterel`

Chapter 6

System Overview

In this chapter I will present the system infrastructure.

The main objective of this work is the generation of robust condition plans that deals with uncertainty in the problem representation.
This approach consist of interpret the output of a planner and generate the correspondent PNP.
This PNP is robust with respect to action failure, as it incorporate sensing actions that helps the robot to discern cases from cases. The architecture presents two layers, each of which will be discussed in the following:

- **the Translator**. This module interprets the output of the planner and translate it into a Conditional Plan.
- **the Generator**. Generates the actual PNP from a Conditional Plan representation.

Each high level action is represented as C++ atomic function, and the PNP is executed using the PNP execution engine.

6.1 Conditional Plan Representation

A conditional plan is a plan that includes also observations. This lead to plan with branches, where a branch represent an action outcome that depends on a property (the observation) that holds true (or not).
These observations are used to handle uncertainty that arises from the lack of knowledge. The uncertainty is handled at execution time, rather than a planning time, where undesirable events are more likely to appear.
The conditioning on actions can remove threats by separate the contexts, through the use of conditional or sensing operators. These operators help the robot to disambiguate the context and must be always executable. At a low level, this means that the robot is equipped with some perception routines that helps him characterise the scenario.
Some could be:

- Face Detection: to detect if there is a person standing in front of him.
- Speech Recognition: to understand human needs.
- Semantic Mapping: to reason about spatial context.

If one can predict the type of failure that may arise during the execution of an action, and model them in the domain, the produced plan is more likely to be continuously executed, with no or limited needs for replanning.
The explicit representation used for conditional plan is the following:

```
plan{
    <state_id>[label=<state_label>,action=<action>]
                ...
    "<state_0>" -> "<state_1>"
    "<state_1>" [A] -> "<state_2>" ; "<state_1>" [not A] -> "<state_N>"
                    ...
}
```

Where are first listed the state with their id and description, and then all the transitions. Transitions could be either deterministic or conditional, where the conditions, enclosed between the square brackets, are boolean predicates evaluated at runtime.

As this certainly increase the robustness of the computed plans, at the same time the search space increase tremendously. Consider For instance an state space of observable properties of dimension N. If we consider only boolean evaluations, the complete space increases as $O(2^N)$. So we must be very careful when dealing with this kind of representation. The discussion of intelligent search algorithms that can exploit, for instance, the structure of the problem are behind the discussion of this thesis. Instead, I will talk about the translation of this kind of plans into an executable PNP.

6.2 Translator Module

The translator module maintain an internal representation of the conditional plan described in 6.1 and is responsible for calling PNPgen, that will generate the actual PNP. The subset of languages/planners chosen in this thesis are PDDL, KPDDL and ROSPlan PDDL.
In the following section I will discuss, for each of the chosen formalism, how the translation works.

6.2.1 PDDL Translator

PDDL, as stated in 3.1.2, is the standard de facto for AI planning.
PDDL vers. 2.1 doesn't have natively syntax constructs that allow conditioning on action effects and this lead to the born of different extension, like KPDDL. Thus, PDDL planners usually generate total ordered plans.
Also in this case, the chosen planner (Fast-Forward) provides us an ordered list of actions. It estimate an heuristic by relaxing the problem and apply to it a graphplan algorithm. This kind of solution are interpreted as linear plans. The algorithm that realize the translation is listed below:

Algorithm 1: ConditionalPlan translation from FF output Algorithm

Input : π: the PDDL description of the plan
Data: v_s: a list of strings
Output: π_{cp}: A Condition Plan

open π;
while π *not eof* **do**
  if $line \neq empty\ and\ valid$ **then**
    $preProcess$(line);
    $v_s \leftarrow$ line;
  end
  $getLine$(line);
end
initialize π_{cp};
for *i less than* $sizeof(v_s)$ **do**
  $s = makeState$(i-th element of v_s);
  i-th element of $\pi_{cp} = s$;
  $addOutcome$(s, i-th+1 element of π_{cp});
end
return π_{cp};
Function $makeState(state\_description)$
  state_name = $findName(state\_description)$;
  state_action = $findAction(state\_description)$;
  foreach *parameter in state_description* **do**
    add(state,$parameter$);
  end
  return state;

The algorithm reads action by action from the FF output. The states are generated incrementally from each of this line and linked one after the other.

Even if the generated plan is not conditional, we decide to incorporate PDDL for multiple reasons.
First of all, we want to show the actual needs for conditional plans in the study cases. Second, PDDL is the most diffused planning language. Since one of the primary intents of this work is to provide a general tool for PNP generation, it is clear that we cannot ignore it.
Many formalisms derive from PDDL, so this class could represent a starting

point to implement other translators for different PDDL extensions.
Furthermore, the generation of this kind of plans is very fast (linear in time in the number of actions) and composing different linear plans is possible to obtain more complex behaviours.

6.2.2 KPDDL Translator

Linear plans are good to model simple cases, but the real world is full of non-determinism and uncertainty. KPDDL introduces a special construct to the classical PDDL structure that could model sensing effects, and allows formalising sensing actions. This helps the agent to disambiguate between two epistemic states: the one in which the sensed property holds and the case in which not.

Algorithm 2: ConditionalPlan translation from KPlanner output Algorithm

Input : π: the KPDDL description of the plan
Data: v_s: a list of strings
Output: π_{cp}: A Condition Plan

open π;
while π *not eof and line not empty* **do**
    $v_s \leftarrow$ line;
end
$\pi_{cp} \leftarrow makeState($ first of $v_s)$;
$\pi_{cp} \leftarrow makeState($ last of $v_s)$;
foreach *element e of* $v_s \neq$ *(first,last)*
do
    s $= makeState(e)$;
    $\pi_{cp} \leftarrow makeState(e)$;
end
return π_{cp};

Function $makeState(state\_description)$
    state_name $= findName(state\_description)$;
    state_action $= findAction(state\_description)$;
    if *condition in state_description* **then**
        while $observation \leftarrow findObservation(state\_description)$ **do**
            $addConditionalOutcome(state, observation)$;
        end
    else
        $addDeterministicOutcome(state)$;
    end
    return state;

6.2.3 ROSPlan Translator

ROSPlan offers different representation for plans, as discussed in 5.5.
Nonetheless it was necessary to use a different planner respect to the one provided, namely CLG [1][1]. CLG stand for Closed-Loop Greedy planner.
This planner rely on a slightly modified version of PDDL which introduce a syntax structure to action effects, **:observe**, that it is used to define sensing actions, in a very similar way to KPDDL.
The description of the problem P, which represent a non-deterministic search problem in belief space, is translated into a non-deterministic problem X(P) in state space by the CLG suite.
CLG Planner then accepts P and solves it using X(P) for keeping track of the beliefs and a strengthening relaxation X+(P) to choose which action to apply next. The relaxation is a classical planning problem that provides an informed heuristic estimator which embeds the sensing.
The solution is semantically interpreted as a digraph (a directed graph), which is represented as:

```
digraph plan_#N{
    state1[ label= "sense_action" ];
    state2[ label= "action_param1_param2" ];
                 ...
    "state1" -> "state2" [ label="obs" ];
    "state1" -> "stateN" [ label="(not(obs))" ];
                 ...
}
```

Each state is presented with the corresponding actions, and actions with multiple outcomes have multiple entries along with the property to observe (enclosed between square brackets). This plan is published on a ROS topic and processed by our node, as we will discuss later.
The algorithm is similar to the ones of KPDDL and PDDL:

[1] `http://www.ai.upf.edu/software/clg-contingent-planner`

Algorithm 3: ConditionalPlan translation from CLG output

Input : π: the digraph description
Data: v_p: a list of pair
Output: π_{cp}: A Condition Plan

open π;
while *line* $==$ *state_description* **do**
    id = $readId(line)$;
    action = $readAction(line)$;
    $v_p \leftarrow \{id, action\}$;
end
initialize π_{cp};
foreach p *in* v_p **do**
    state_name = first of p;
    state_action = second of p;
    i-th element of π_{cp} = state;
end
while $line \neq eof$ **do**
    source = $findSource(line)$;
    destination = $findDestination(line)$;
    if *line contains observation* **then**
        observation = $findObservation(line)$;
    else
        observation = "";
    end
    if *observation* $\neq$ "" **then**
        addConditionalOutcome(observation,$findPosition(\pi_{cp}, destination)$);
    else
        addDeterministicOutcome($findPosition(\pi_{cp}, destination)$);
    end
    $getLine(line)$;
end
$\pi_{cp} \leftarrow addGoal()$;
return π_{cp};

Notice that the planner doesn't specify a goal state, so we add a goal after each leaf for safety of execution. The solution and the correspondent translation are computed almost instantaneously, thanks to CLG heuristic, making it possible to run the complete algorithm on-line on a ROS node.

6.3 Generator Module

6.3.1 PNPgen: A Conditional Planning Extension

Summarizing, the languages formalism induce the following solutions:

- **PDDL (FF planner)**: produce a linear plan.
- **PDDL (CLG Planner)**: produce a tree.
- **KPDDL (KPlanner)**: produce a direct graph.

In order to achieve our results, it was necessary to extend PNPgen module to consider also conditional plans.
This means, in particular, to realize a graph search on the conditional plan. In fact the generation of the actual PNP, once we have created the conditional plan, is common for every formalisms.

In the next section I will discuss the algorithm that is responsible for PNPs generation.

6.3.2 the Generation Algorithm

The algorithm responsible for the generation of the PNP performs a graph expansion using the conditional plan representation kept by the correspondent translator class.
The simplest case is the one of the standard PDDL domain, in which the algorithm recursively expands the serial sequence of states. In fact, for each, there is only one successor and no observation.

Algorithm 4: Generate Serial Plans Algorithm

Input : π: the conditional plan,
s_i: π initial state
Output: π_c: PNP generated from the plan

```
if π has next then
    p_1 ← generatePlace(s_i);
    addPlace(π_c, p_1);
    recursive call on (π.next, p_1);
else
    return;
end
```

More interesting is the case of the plans with observations, as showed in algorithm 5.

In order to be sure that every node is visited only once, and thus to avoid dangerous loops in the generation procedure, we maintain a stack of the visited nodes. Each time, we take the top of this stack, check if it is a final state or has outgoing edges, and expand it. If it hasn't outcomes, we add a final state and continue with the search. Otherwise, for each edge, we check if the corresponding destination is already in the visited map. If this is the case, we connect it with the current state and close the loop. Otherwise we expand it and create a new branch.

In the next chapter I will show some examples and results to demonstrate the approach.

In particular, I will characterize the input/output of each problem and provide a sketch for a possible optimal solution.

Algorithm 5: PNPgen from Conditional Plan

Input : π: a Conditional Plan,
s_f: π final state
M_{SA}: a Map between state-action and outcomes

Data: M_V: a Map of the visited places
SS: a Stack containing the places to explore
v_o: a list of possible action outcomes

Output: π_c: PNP generated from a Conditional Plan

$curr\_state \leftarrow \pi.first\_state$;
$\pi_c \leftarrow addCondition(init)$;
$M_V \leftarrow insert(curr\_state)$;
$SS \leftarrow push(curr\_state)$;
while *($SS \neq 0$)* **do**
    curr_state = top of SS;
    $SS.pop()$;
    action = $M_{SA}[curr\_state]$;
    if *(action == "" and curr_state $\neq s_f$)* **then**
        continue;
    end
    $v_o \leftarrow M_{SA}[curr\_state].outcomes$;
    if $v_o == 0$ **then**
        $addFinalState(\pi_c)$;
        continue;
    end
    $addAction(\pi_c, action, current\_state)$;
    foreach *element e of v_o*
    do
        $succ\_state = successor(e)$;
        $obs = observation(e)$;
        if $M_V[succ\_state] == 0$ **then**
            $addBranch(\pi_c, obs)$;
            $SS.push(succ\_state)$;
            $M_V.add(succ\_state)$;
        else
            $addTransitionBack(\pi_c, obs)$;
        end
    end
end

Chapter 7

Use Cases

In this section I will characterize some examples, describing the whole process from the problem definition to the plan generation.
In order to do so I will start from the motivating example introduced in section 1.1.

7.1 from PDDL to PNP

PDDL was meant to express the "physics" of a domain, and could not represent the epistemic state of an agent.
The plan is generated and then executed with "eyes closed". The advantage of this kind of approach is that is very fast, but it could easily fail due to mismatch between the planned plan and the real world.

7.1.1 Example 1: COACHES

I will present a solution in PDDL (ver. 1.2) to the example presented in 1.1. We must then characterise objects, predicates, actions, initial and final states.
I used FF [13][1](ver. 2.3) planner to check the satisfiability of the problem and to solve it. For the sake of clarity, in the domain description I will consider only the agent $h2$ of the example.

For the predicates we have:

```
(:predicates
            ;robot and person properties
            (person ?x) (robot ?x)
            (at-robot ?x) (at-person ?x)
            (close-to ?x)(wait ?x)(bags-loaded ?x)

            ;needs represents the person necessity
            (needs-help ?x) (needs-WC ?x) (needs-obj ?x)
            (needs-food ?x) (needs-clothes ?x)(satisfied ?x)

            ;desire predicates used to disambiguate which food/cloth
            (desire-italian ?x) (desire-chinese ?x)
            (desire-sportive ?x) (desire-fashion ?x)

            ;locations predicates
            (SHOP ?x) (entrance ?x) (WC ?x) (market ?x)
            (sportive-clothes ?x) (fashion-clothes ?x)
            (chinese-restaurant ?x) (italian-restaurant ?x)

)
```

Beside from predicates that model environmental properties, we notice the ones that (tries to) describe agents state. For what concern actions, I will not list them all, but I rather focus on the ones that are supposed to be sensing actions:

```
(:action sense-person
```

[1]https://fai.cs.uni-saarland.de/hoffmann/ff.html

```
                      :parameters ( ?r ?p)
                      :precondition (and(person ?p)(robot ?r)(wait ?r))
                      :effect (not(wait ?r))
)

(:action ask_clothes
           :parameters ( ?r ?p ?x)
           :precondition (and (person ?p)(robot ?r)(close-to ?p))
           :effect (and
                        (needs-clothes ?p)
                        (when
                              (and (at-person ?x)(sportive-clothes ?x))
                              (desire-sportive ?p)
                        )
                        (when
                              (and (at-person ?x)(fashion-clothes ?x))
                              (desire-fashion ?p)
                        )
                   )
)

(:action ask_food
             :parameters ( ?r ?p ?x)
             :precondition (and (person ?p)(robot ?r)(close-to ?p)(at-person ?x))
             :effect (and
                          (needs-food ?p)
                          (when
                                (italian-restaurant ?x)
                                (desire-italian ?p)
                          )
                          (when
                                (chinese-restaurant ?x)
                                (desire-chinese ?p)
                          )
                     )
)

(:action ask_object
             :parameters ( ?r ?p ?x)
             :precondition (and (person ?p)(robot ?r)(close-to ?p)(at-person ?x)(market ?x))
```

```
             :effect (needs-obj ?p)
)
```

It is clear from the specifications that they are not real sensing actions, as we have previously defined.
In fact, in standard PDDL, the construct "**when**" is the only one that allow a simple form of conditioning on actions effects. All the information must be known at planning time in order to compute a solution and there isn't the possibility of specifying actions with non-deterministic effects.
The objects are:

```
  (:objects shop1 shop2 shop3
          shop4 shop5 shop6
          entr1 entr2
          food clothes obj
          WC help none
          r h2
  )
```

where $\{food, clothes, obj, WC, help, none\}$ models the Z variables presented in 1.1.
We can then characterize the initial and final state as:

```
(:init (robot r)(person h2)
       (wait r) (desire-fashion h2)
   (at-robot shop4) (at-person shop2)
       (entrance entr1)(entrance entr2)
       (SHOP shop1)(SHOP shop2)(SHOP shop3)
       (SHOP shop4)(SHOP shop5)(SHOP shop6)
       (market shop1)(WC shop3)
       (sportive-clothes shop2)(fashion-clothes shop4)
       (chinese-restaurant shop5)(italian-restaurant shop6)
)
(:goal (satisfied h2) )
```

Given this domain and this problem instance, where the "*desire-fashion h2*" predicate is explicitly indicated in the initial state, the output of the FF planner is:

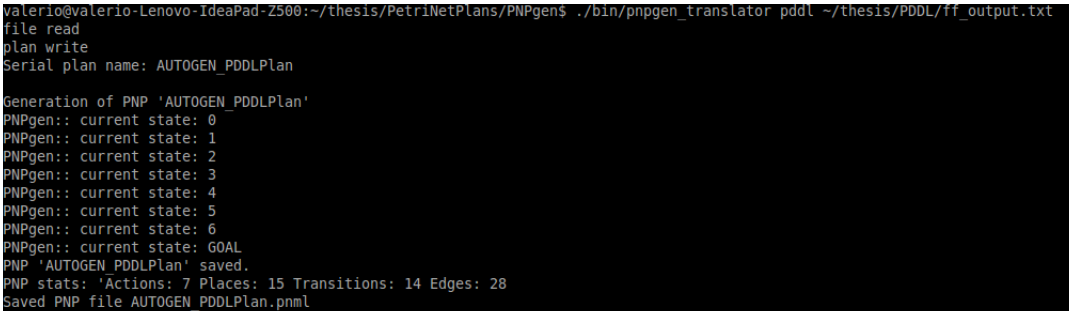
```
ff: parsing domain file
domain 'COACHES' defined
 ... done.
ff: parsing problem file
problem 'H2-PLAN' defined
 ... done.

ff: found legal plan as follows

step    0: SENSE-PERSON R H2
        1: MOVE SHOP4 SHOP2 R
        2: APPROACH-PERSON H2 SHOP2
        3: ASK_CLOTHES R H2 SHOP1
        4: BRINGTO R H2 SHOP2 SHOP4
        5: ADVERTISE-FASHION SHOP4 H2
        6: CHECK_SATISFACTION H2
```

In this scenario, we can imagine an infrastructure with a planning system responsible of the generation of the problem instances and a module responsible for gathering the informations (through sensors and physical interfaces) and that manage fluents. This module, for instance, would be the one responsible for the Knowledge Base updates so to justify the presence of "*desire-fashion h2*" ground in the initial state description.

Once the solution is computed and stored in a file, the PDDL translator fills its internal structure to represent the plan and subsequently expose it to PNPgen. The correspondent translation is the following:

```
plan{
 n_states=7
 0[label=0,actions=sense-person_r_h2]
 1[label=1,actions=move_shop4_shop2_r]
 2[label=2,actions=approach-person_h2_shop2]
 3[label=3,actions=ask_clothes_r_h2_shop1]
 4[label=4,actions=bringto_r_h2_shop2_shop4]
 5[label=5,actions=advertise-fashion_shop4_h2]
 6[label=6,actions=check_satisfaction_h2]
```

```
 "0" -> "1"
 "1" -> "2"
 "2" -> "3"
 "3" -> "4"
 "4" -> "5"
 "5" -> "6"
}
```



```
valerio@valerio-Lenovo-IdeaPad-Z500:~/thesis/PetriNetPlans/PNPgen$ ./bin/pnpgen_translator pddl ~/thesis/PDDL/ff_output.txt
file read
plan write
Serial plan name: AUTOGEN_PDDLPlan

Generation of PNP 'AUTOGEN_PDDLPlan'
PNPgen:: current state: 0
PNPgen:: current state: 1
PNPgen:: current state: 2
PNPgen:: current state: 3
PNPgen:: current state: 4
PNPgen:: current state: 5
PNPgen:: current state: 6
PNPgen:: current state: GOAL
PNP 'AUTOGEN_PDDLPlan' saved.
PNP stats: 'Actions: 7 Places: 15 Transitions: 14 Edges: 28
Saved PNP file AUTOGEN_PDDLPlan.pnml
```

Figure 7.1: PNPgen generates the PNP from FF output.

When the PNP is generated, it could be visualized and manipulated using Jarp. The correspondent PNP in showed in 7.2.
The whole process, translation+generation, take place in 0,005s (average time). The planning runs in only 0,007s.
All the results are obtained using the linux command **time**[2].

[2]https://linux.die.net/man/1/time

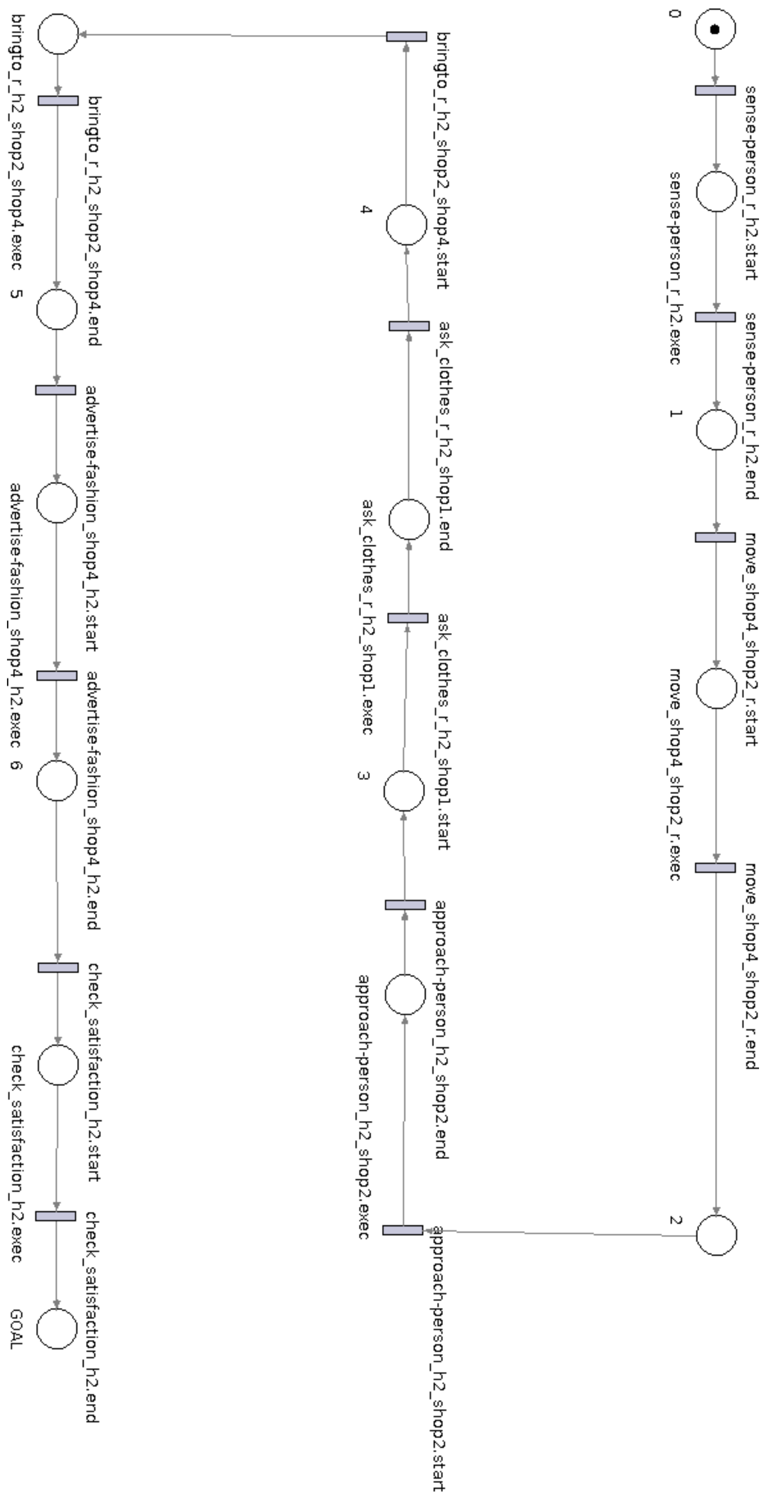


Figure 7.2: the PNP generated from the coaches PDDL domain. Notice that we had to modify the PNP for a better visualization.

7.1.2 Example 2: Gripper Problem

The problem consists in a robot with two hands, moving balls from a room A to a room B. The size of the problem is expressed terms of balls to move from one room to another. We show that doubling the problem size the generation algorithm double its runtime, i.e. the algorithm is linear in the problem size. This is demonstrated in the following table:

Problem Size	Time (s)
50	0,013
100	0,024
200	0,04

Figure 7.3: the time for PNPgen to run for an increasing size problem.

Where the time is the average among five algorithms runs. Obviously also PNPs double their sizes, as we can see from the generation stats:

1. **50 balls:** Actions: 149 Places: 299 Transitions: 298 Edges: 596.
2. **100 balls:** Actions: 299 Places: 599 Transitions: 598 Edges: 1196.
3. **200 balls:** Actions: 599 Places: 1199 Transitions: 1198 Edges: 2396.

7.2 from KPDDL to PNP

KPDDL permits to represent conditional plans. The cons reside in the fact that depending on the written domain, the planner could take much time to compute a solution. The author provides the possibility to drive the solution search by the means of a heuristic factor that varies in the range of $[0.0 - 1.0]$, indicating with 0 a breadth-first search while with 1 a depth-first search.
In any case, this not represents a safe solution for an on-line approach as there are no ways to know if a solution exist (see 3.2).

7.2.1 Example 1: COACHES

In KPDDL sensing actions could be modelled using the syntactic construct **:sense**. Most of the predicates, actions and constants remain unchanged with respect to the PDDL case:

```
  (:action sense_person
           :parameters ( ?r - robot ?x -location)
           :precondition (sensing ?r)
           :sense (at-person ?x)
  )

  (:action ask_object
           :parameters ( ?p - person )
           :precondition (close-to ?p)
           :sense (needs-obj ?p)
  )

  (:action ask_clothes
           :parameters ( ?p - person )
           :precondition (close-to ?p)
           :sense (needs-clothes ?p)
  )


 (:action ask_food
          :parameters ( ?p - person )
          :precondition (close-to ?p)
          :sense (needs-food ?p)
 )
```

The substantial difference is that after the execution of such actions the agent knows whether the property sensed by the action is true or false. We can then define the initial state and the goal as following:

```
(define
  (problem h2-plan)
  (:domain coaches)
  (:formula-init (and
                   (at-robot shop4)
```

```
                    (at-person shop2)
                    (market shop1)
                    (sportive-clothes shop2)
                    (WC shop3)
                    (fashion-clothes shop4)
                    (chinese-restaurant shop5)
                    (italian-restaurant shop6)
                )
  )

  (:goal
      (satisfied h2)
  )
)
```

As we can see, we do not need to model explicitly in the initial state the knowledge about the person needs.
satisfied is a predicate to indicate a final state and became true after the action *bye*, when the robot as finished to acquire information from the human:

```
(:action bye
            :parameters (?p - person)
            :precondition (not (needs-food ?p))
            :effect (satisfied ?p)
)
```

As we can see, in this case the property to sense is *needs-food*. When given as input to KPlanner, the domain and the problem instance produce the following output:

```
INIT: 7
  1 :    approach-person_h2_shop2 --> 2
  2 :    ask_food_h2 (F) --> 3    ask_food_h2 (T) --> 4
  3 :    bye_h2 --> 0
  4 :    bringto_h2_shop2_shop6 --> 5
  5 :    advertise-italian_h2_shop6 --> 6
  6 :    bye_h2 --> 0
  7 :    move_shop4_shop2_r --> 1
  0 : GOAL
```

The state description is the conjunction of the properties that holds true in a state.

Consider, for instance, the state $2, 3, 4$ corresponding to the sensing action *ask_food* (the ground predicates corresponding to the shops are omitted):

```
4 = ( at-robot_shop2  at-person_shop2  close-to_h2 )
5 = ( at-robot_shop2  at-person_shop2  close-to_h2 -needs-food_h2 )
6 = ( at-robot_shop2  at-person_shop2  close-to_h2 needs-food_h2 )
```

Here we can notice how the epistemic state changes.

In particular, after *ask_food* execution, we can observe two distinct state: one in which **needs-food_h2** holds true, and one in which not.

```
valerio@valerio-Lenovo-IdeaPad-Z500:~/thesis/PetriNetPlans/PNPgen$ ./bin/pnpgen_translator kpddl ~/thesis/PDDL/planners/KPlanner/plans/coaches.txt
KPDDL Conditional Plan name: AUTOGEN_KPDDLPlan

Generation of PNP 'AUTOGEN_KPDDLPlan'
Initial state 7
Final state GOAL
PNPgen::  7 : move_shop4_shop2_r -> [] 1
PNPgen::  1 : approach-person_h2_shop2 -> [] 2
PNPgen::  2 : ask_food_h2 -> [f] 3    [t] 4
PNPgen::  4 : bringto_h2_shop2_shop6 -> [] 5
PNPgen::  5 : advertise-italian_h2_shop6 -> [] 6
PNPgen::  6 : bye_h2 -> [] GOAL
PNPgen::  GOAL : PNPgen::  3 : bye_h2 -> [] GOAL
PNP 'AUTOGEN_KPDDLPlan' saved.
PNP stats: 'Actions: 7 Places: 23 Transitions: 23 Edges: 46
Saved PNP file AUTOGEN_KPDDLPlan.pnml
```

Figure 7.4: PNPgen generates the PNP from KPlanner output. the observation are enclosed between square brackets.

We translate the solution in our conditional plan representation format. Notice that even the KPlanner output doesn't differ so much from our representation, this is meant to be the same for every considered formalisms (present and future).

```
plan{

 n_states=8

 0[label=7,actions=move_shop4_shop2_r]
 1[label=1,actions=approach-person_h2_shop2]
 2[label=2,actions=ask_food_h2]
 3[label=3,actions=bye_h2]
 4[label=4,actions=bringto_h2_shop2_shop6]
 5[label=5,actions=advertise-italian_h2_shop6]
```

```
6[label=6,actions=bye_h2]
7[label=GOAL,actions=]

"7" -> "1"
"1" -> "2"
"2" [f] -> "3" ; "2" [t] -> "4"
"3" -> "GOAL"
"4" -> "5"
"5" -> "6"
"6" -> "GOAL"

}
```

Once represented, we can translate the conditional plan into a PNP. Action with no conditions are intend to be deterministic, while every time we sense for a property we branch on the boolean result of the action (true or false) as we can see from figure 7.5.

The planner takes 0.569s on average to run given this problem instance, most of these spent in the search. The translation+generation takes only 0.008s time to run. We will compare this results with the ones of the next example.

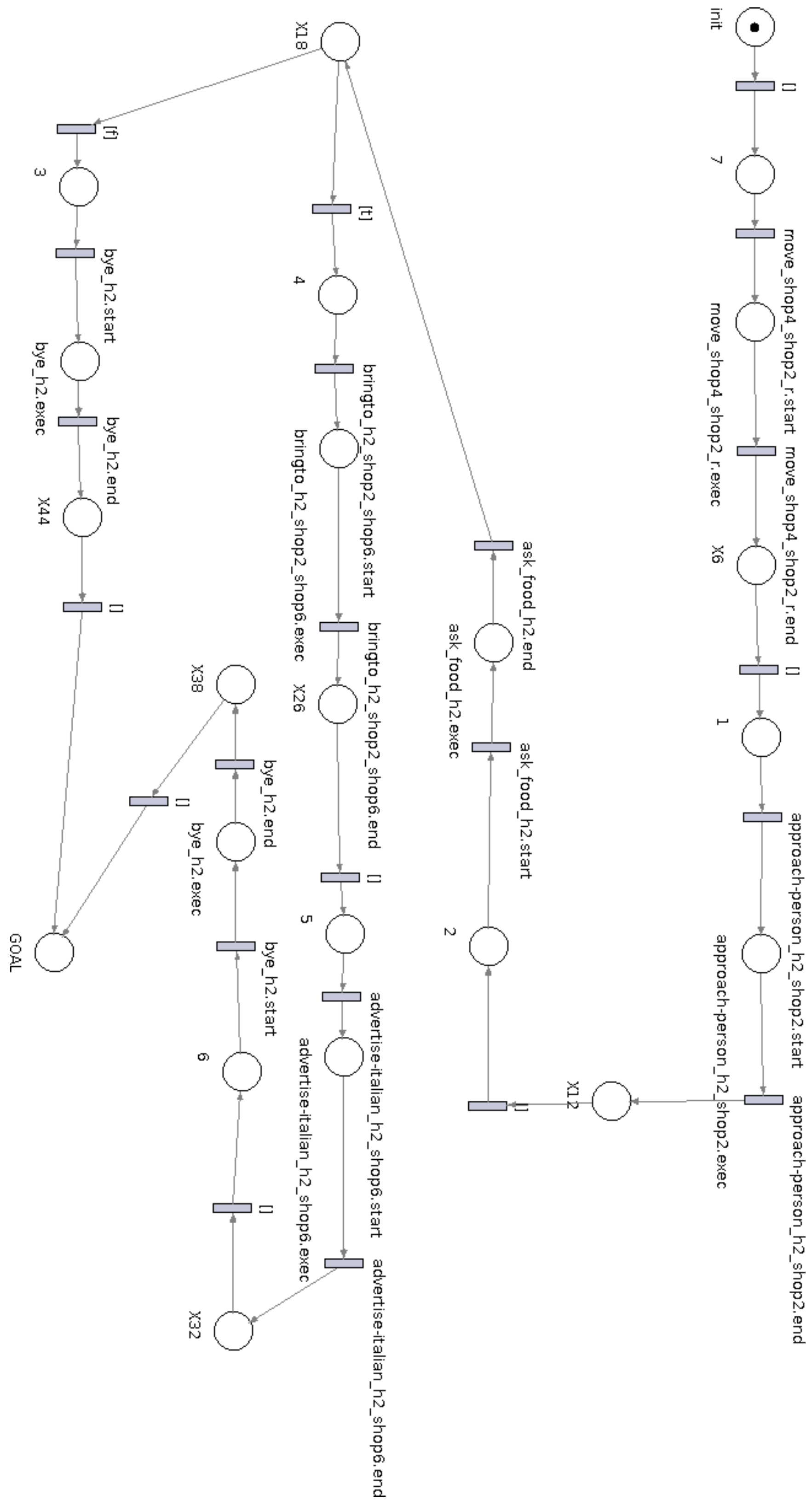


Figure 7.5: the PNP generated from the coaches KPDDL domain. Notice that we had to modify the PNP for a better visualization.

7.2.2 Example 2: COACHES enhanced

As counterexample, consider the following domain in which we have modified the bye action and we ask the agent to sense for two properties:

```
(:action bye
            :parameters (?p - person)
            :precondition (   and
                              (not (needs-obj ?p))
                              (not (needs-food ?p))
                          )
            :effect (satisfied ?p)
)
```

The conditional plan correspondent to the computed solution is the following:

```
plan{

 n_states=12

 0[label=11,actions=move_shop4_shop2_r]
 1[label=1,actions=approach-person_h2_shop2]
 2[label=2,actions=ask_object_h2]
 3[label=3,actions=ask_food_h2]
 4[label=4,actions=bye_h2]
 5[label=5,actions=bringto_h2_shop2_shop6]
 6[label=6,actions=advertise-italian_h2_shop6]
 7[label=7,actions=bye_h2]
 8[label=8,actions=bringto_h2_shop2_shop1]
 9[label=9,actions=advertise-objects_shop1_h2]
 10[label=10,actions=bringto_h2_shop1_shop2]
 11[label=GOAL,actions=]

 "11" -> "1"
 "1" -> "2"
 "2" [f] -> "3" ; "2" [t] -> "8"
 "3" [f] -> "4" ; "3" [t] -> "5"
 "4" -> "GOAL"
 "5" -> "6"
 "6" -> "7"
 "7" -> "GOAL"
```

```
 "8" -> "9"
 "9" -> "10"
 "10" -> "3"

}
```

As we can see is significantly more complex than the previous, as we can also see from the correspondent PNP depicted in 7.6.
Such complex PNPs are very time consuming to be manually constructed and error prone. However, we can notice that doubling the number of epistemic variables, the time for the planner to run increases exponentially from 0.569s to 23,8252s! The run time of our system however remain almost the same (from 0,008s to 0,009). It is clear that if we double the epistemic states, the planner runs indefinitely.

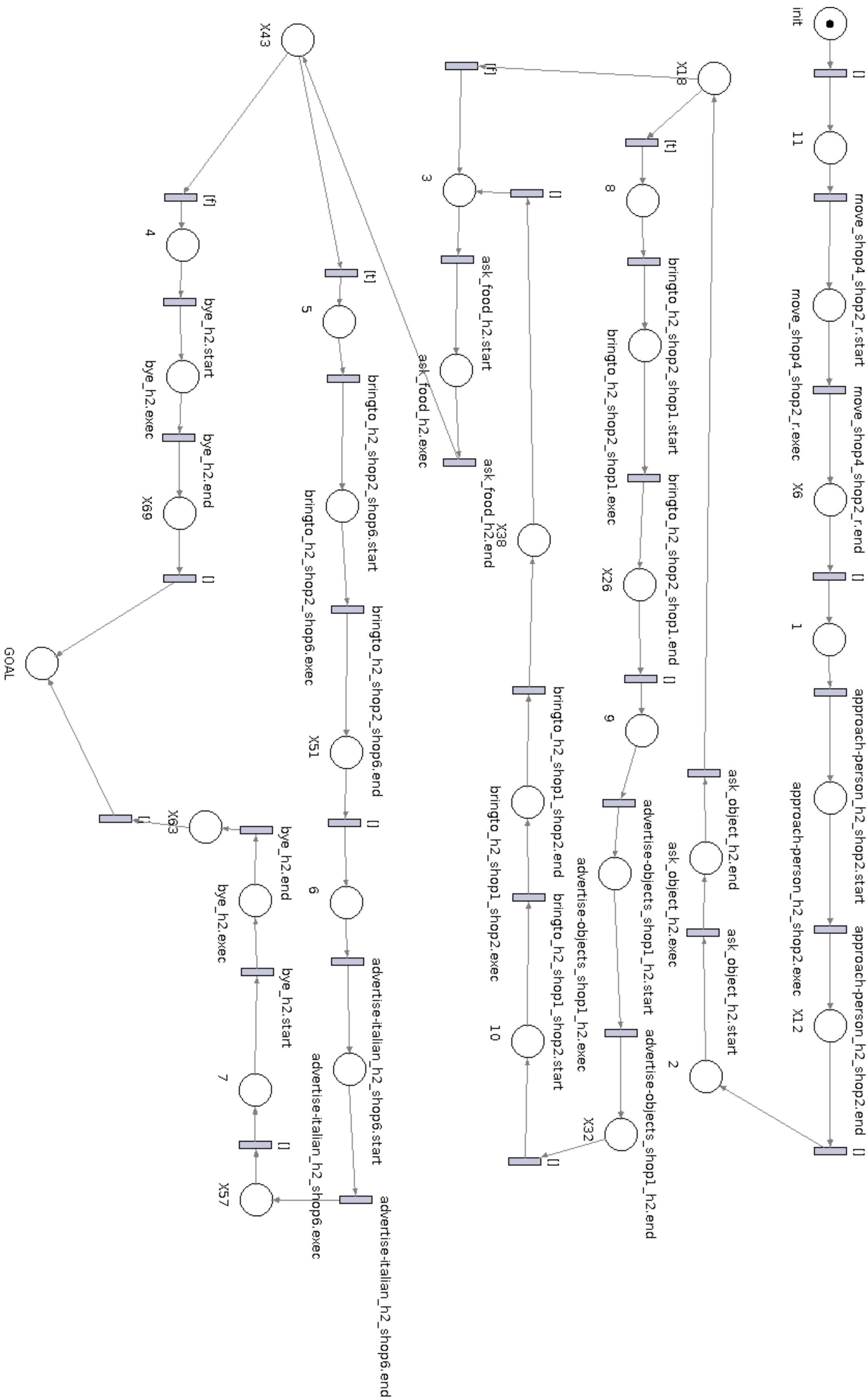


Figure 7.6: the PNP generated from the modified coaches KPDDL domain. Notice that we had to modify the PNP for a better visualization.

7.3 from ROSPlan PDDL to PNP

ROSPlan represents the most prominent solution for an on-line approach. The used planner, CLG, has a very intelligent heuristic that can present almost instantaneously a solution, when present, or communicate that this not exist. The ROSPlan framework is perfectly integrated into ROS and, due to its modularity, it can be extended to consider other formalisms (like we have done for this work).

The infrastructure provides a safe way to handle planning failures, since even when a solution is not found in the first place, it can replan a certain number of times based on a tunable parameter.
This comes in handy since if the PNP execution fails, we have a second level of redundancy to make sure that the robot doesn't suddenly stop.
The integration between ROSPlan and PNP is an ongoing work and I will discuss possible future implementations in the next chapter.

7.3.1 Example 1: simple COACHES

This is a simplified version of the coaches example, in which we inspect only the person position. We must introduce a slightly different version of PDDL, similar to KPDDL, that is used in the ROSPlan Framework.
As in KPDDL, in this case PDDL is extended with introduction of the **:observe** construct for action effects, used to define sensing actions.
For instance, a sensing action has the following form:

```
(:action sense_person
                      :parameters (?x -location)
                      :precondition (at-robot ?x)
                      :observe (at-person ?x)
)
```

Beside from the *observe* construct, we must provide the problem specification with the following expression:

```
(oneof
```

```
    (at-person shop1)
    (at-person shop2)
    (at-person shop3)
    (at-person shop4)
    (at-person shop5)
    (at-person shop6)
 )
```

These are necessary for the planner to derive a supplementary heuristic used while searching for the solution. We avoid to present the complete domain as this is essentially the same of the previous sections.

A ROS node serves as interface between ROSPlan and PNPgen. This node actively listens the topic where the plan is published by the planning system. The planning system fetch the PDDL model from the Knowledge Base and generates a problem instance. After that, he calls the planner and dispatch its output on the topic "*/kcl_rosplan/plan_graph*"

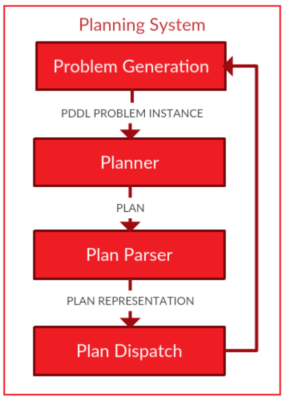


Figure 7.7: The planning system architecture.

The output is represented as a direct graph, as described in section 6.2.3:

```
digraph plan_0 {
0[ label="sense_person_shop4" style="fill: #fff; "];
1[ label="move_shop4_shop5_r" style="fill: #fff; "];
2[ label="sense_person_shop5" style="fill: #fff; "];
3[ label="move_shop5_shop3_r" style="fill: #fff; "];
```

```
4[ label="sense_person_shop3" style="fill: #fff; "];
5[ label="move_shop3_shop2_r" style="fill: #fff; "];
6[ label="sense_person_shop2" style="fill: #fff; "];
7[ label="move_shop2_shop1_r" style="fill: #fff; "];
8[ label="sense_person_shop1" style="fill: #fff; "];
9[ label="move_shop1_shop6_r" style="fill: #fff; "];
10[ label="approach-person_h2_shop6" style="fill: #fff; "];
11[ label="bye_h2" style="fill: #fff; "];
12[ label="approach-person_h2_shop1" style="fill: #fff; "];
13[ label="bye_h2" style="fill: #fff; "];
14[ label="approach-person_h2_shop2" style="fill: #fff; "];
15[ label="bye_h2" style="fill: #fff; "];
16[ label="approach-person_h2_shop3" style="fill: #fff; "];
17[ label="bye_h2" style="fill: #fff; "];
18[ label="approach-person_h2_shop5" style="fill: #fff; "];
19[ label="bye_h2" style="fill: #fff; "];
20[ label="approach-person_h2_shop4" style="fill: #fff; "];
21[ label="bye_h2" style="fill: #fff; "];
"0" -> "20" [ label="at-person shop4" ];
"0" -> "1" [ label="(not (at-person shop4))" ];
"1" -> "2"
"2" -> "18" [ label="at-person shop5" ];
"2" -> "3" [ label="(not (at-person shop5))" ];
"3" -> "4"
"4" -> "16" [ label="at-person shop3" ];
"4" -> "5" [ label="(not (at-person shop3))" ];
"5" -> "6"
"6" -> "14" [ label="at-person shop2" ];
"6" -> "7" [ label="(not (at-person shop2))" ];
"7" -> "8"
"8" -> "12" [ label="at-person shop1" ];
"8" -> "9" [ label="(not (at-person shop1))" ];
"9" -> "10"
"10" -> "11"
"12" -> "13"
"14" -> "15"
"16" -> "17"
"18" -> "19"
"20" -> "21"
```

```
}
```

As soon as a plan is published, the node writes it to a file and calls the PNPgen class responsible for the translation and generation of the actual PNP.
The following picture depict the connection between our node (*pnp_rosplan_interface*) and the rest of the architecture:

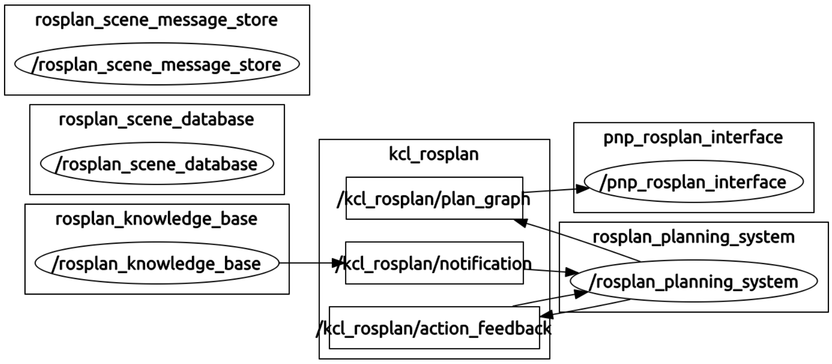


Figure 7.8: A snapshot of the nodes used to translate a digraph into a PNP.

You will find a portion of the correspondent PNP in the next page.

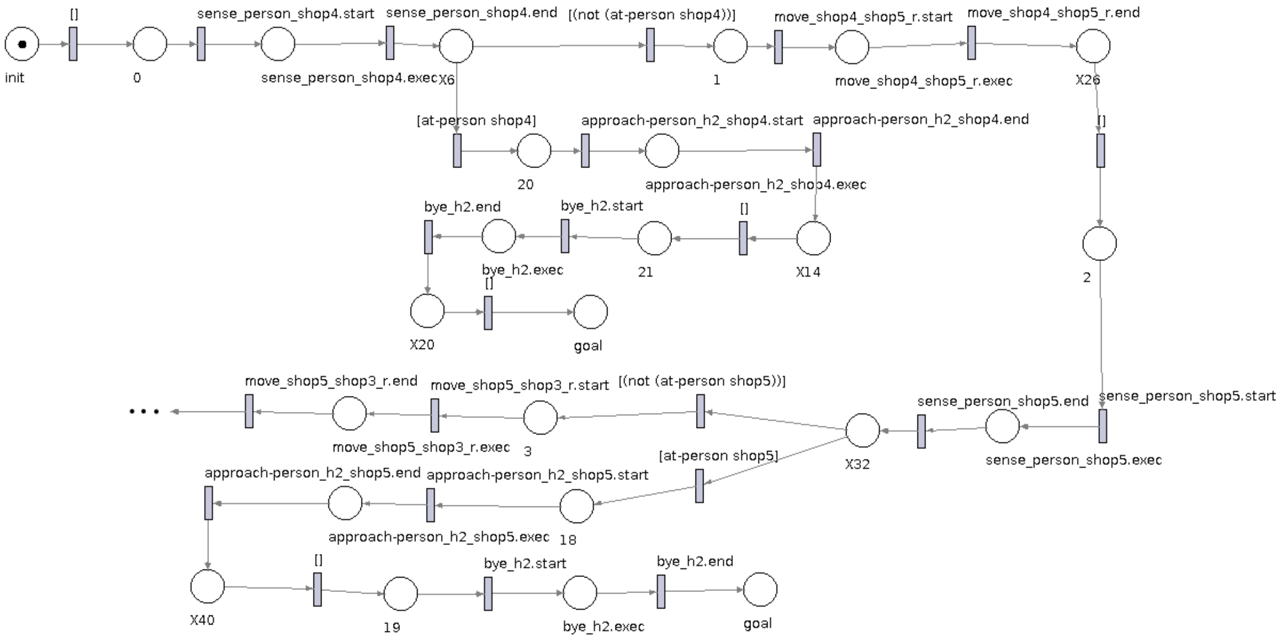


Figure 7.9: a portion of the PNP generated from the ROSPlan Digraph. Notice that we had to move some places/transitions for a better visualization.

7.3.2 Example 2: COACHES complete

This is the complete coaches example, as presented in the previous sections. Namely, we try to infer also the person needs using the following sensing action:

```
(:action ask
  :parameters ( ?p - person ?x - desire)
  :precondition (close-to ?p)
  :observe (needs ?x)
)
```

The predicate *needs* can take five values: obj, food, clothes, WC and help. As before, we must specify the possible outcomes in the initial state:

```
(:init (and
           (at-robot shop4)
           (oneof
             (at-person shop1)
```

```
              (at-person shop2)
              (at-person shop3)
              (at-person shop4)
              (at-person shop5)
              (at-person shop6)
            )
            (oneof
              (needs obj)
              (needs food)
              (needs clothes)
              (needs help)
              (needs WC)
            )
        )
  )
```

We do not show the complete PNP since it is too big to fit in a page. Consider that for this case there are 211 places and 422 edges. As we can see from the image 7.10, the generated PNP is complete in the sense that can represent the epistemic knowledge of the robot and the generation happens almost instantaneously. This example concludes our analysis.

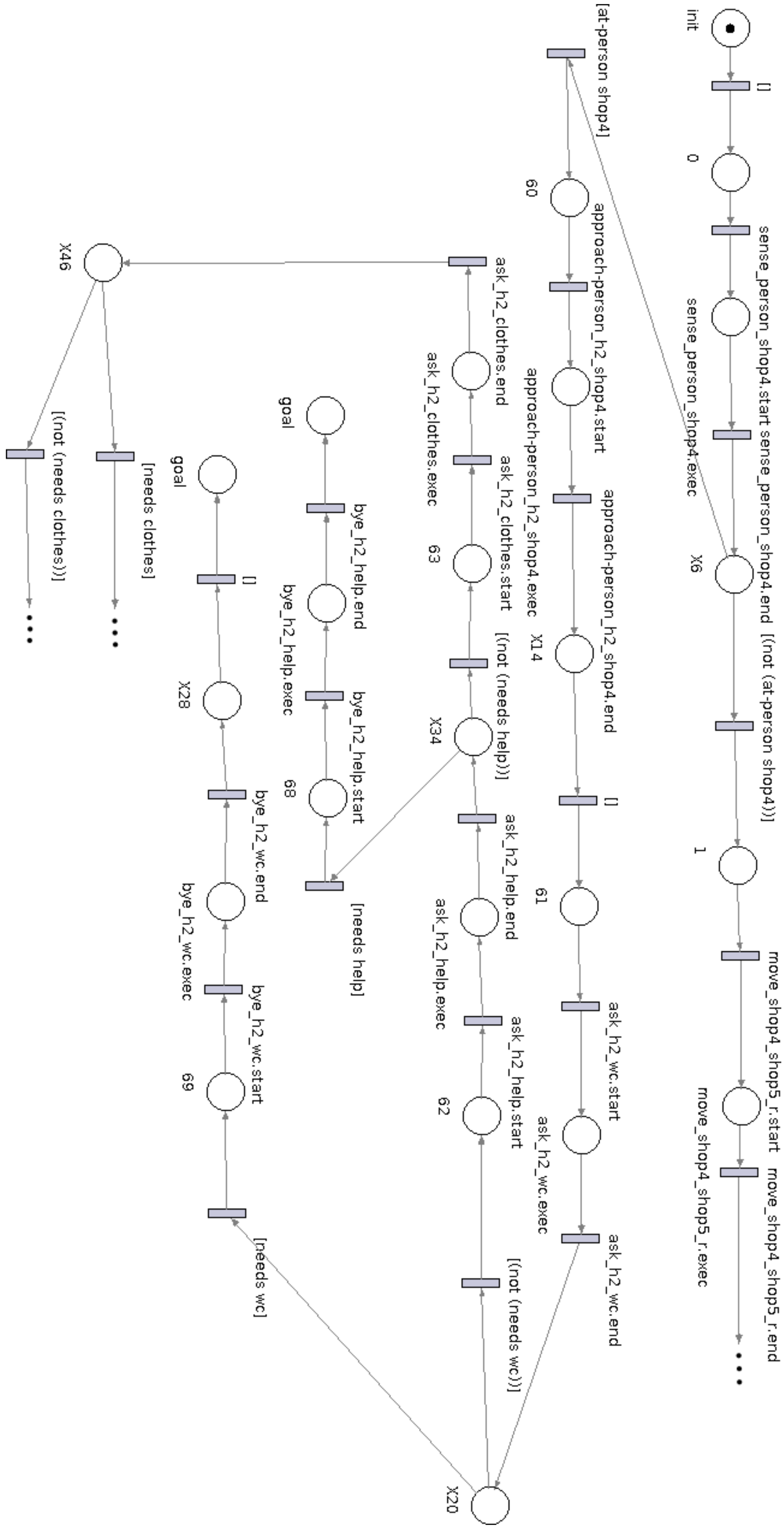


Figure 7.10: a small portion of the PNP generated from the complete coaches domain. Notice that we had to modify the PNP for a better visualization.

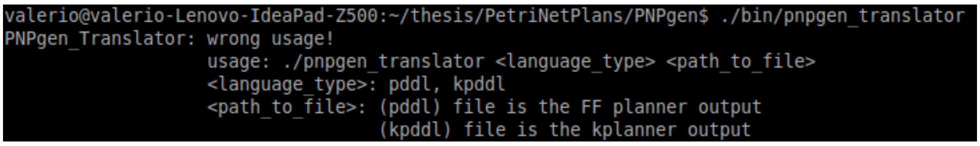
Chapter 8

Conclusion

One of the fundamental aspects of moving robots from laboratories to everyday environments is that the interaction with unaware persons should be the most natural and smooth as possible.
In fact, a robot that "freeze" during an interaction or that could not response actively to the external stimuli is very far from the definition of social.
In this work we have discussed an approach that tries to take advantage of the expressivity of some of the languages and formalisms present in the literature to represent, generate and execute conditional plans, i.e. plans that incorporates observations.
This kind of plans are robust with respect to action failures since we can represent very complex plans that are otherwise impossible to write "manually".
Using the PNP framework we can easily visualize and eventually modify a plan.
Furthermore, we have demonstrated the integration of our work with an infrastructure that runs on-line on the robot and how promising this approach is. We are indeed collaborating with ROSPlan5.5 authors to extend further this work.

8.1 Future Development

There are some things still missing in our framework that can be considered for future implementations.
For instance, it is evident the lack of a graphical interface. It is clear that if we want to expand our work considering more formalisms, a tool that automatically selects one or another based on desired properties becomes fundamental.



```
valerio@valerio-Lenovo-IdeaPad-Z500:~/thesis/PetriNetPlans/PNPgen$ ./bin/pnpgen_translator
PNPgen_Translator: wrong usage!
                  usage: ./pnpgen_translator <language_type> <path_to_file>
                  <language_type>: pddl, kpddl
                  <path_to_file>: (pddl) file is the FF planner output
                                  (kpddl) file is the kplanner output
```

Figure 8.1: the current interface to generate PNPs from (K)PDDL.

Another important aspect

- Add More Formalisms – PDDL extensions in the translation
- Planner selection/study
- from Boolean Evaluation to an Higher Logic
- Merging plans
- ER rules

Bibliography

[1] Alexandre Albore. Translation-based approaches to automated planning with incomplete information and sensing. phd, Universitat Pompeu Fabra, 2011.

[2] Emanuele Bastianelli, Domenico Daniele Bloisi, Roberto Capobianco, Fabrizio Cossu, Guglielmo Gemignani, Luca Iocchi, and Daniele Nardi. On-line semantic mapping. In *Advanced Robotics (ICAR), 2013 16th International Conference on*, pages 1–6. IEEE, 2013.

[3] Ronen Brafman, Alexander Gorohovski, and Guy Shani. A contingent planning-based pomdp replanner. *Models and Paradigms for Planning under Uncertainty: a Broad Perspective*, page 44, 2014.

[4] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1):165–204, 1994.

[5] Fabio Maria Carlucci, Lorenzo Nardi, Luca Iocchi, and Daniele Nardi. Explicit representation of social norms for social robots. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 4191–4196. IEEE, 2015.

[6] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, Narcís Palomeras, Natalia Hurtos, and Marc Carreras. Rosplan: Planning in the robot operating system. In *ICAPS*, pages 333–341, 2015.

[7] Xiaoping Chen, Jianmin Ji, Jiehui Jiang, Guoqiang Jin, Feng Wang, and Jiongkun Xie. Developing high-level cognitive functions for ser-

vice robots. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 989–996. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[8] Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *ICAPS*, pages 42–49, 2010.

[9] Francesco M Donini, Daniele Nardi, and Riccardo Rosati. Autoepistemic description logics. In *IJCAI (1)*, pages 136–141, 1997.

[10] Esra Erdem, Erdi Aker, and Volkan Patoglu. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics*, 5(4):275–291, 2012.

[11] Alessandro Farinelli, Luca Iocchi, Daniele Nardi, and Vittorio Amos Ziparo. Assignment of dynamically perceived tasks by token passing in multirobot systems. *Proceedings of the IEEE*, 94(7):1271–1288, 2006.

[12] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[13] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. 14:253–302, 2001.

[14] Luca Iocchi, Laurent Jeanpierre, Maria Teresa Lazaro, and Abdel-Illah Mouaddib. A practical framework for robust decision-theoretic planning and execution for service robots. In *Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016.

[15] Luca Iocchi, Daniele Nardi, and Riccardo Rosati. A pddl extension for describing planning domains with incomplete information and sensing.

[16] Vladimir Lifschitz. What is answer set programming?. In *AAAI*, volume 8, pages 1594–1597, 2008.

[17] John McCarthy and Patrick J Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Readings in artificial intelligence*, pages 431–450, 1969.

[18] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. 1998.

[19] Pier Francesco Palamara, Vittorio A Ziparo, D Nardi, P Lima, H Costelha, et al. A robotic soccer passing task using petri net plans. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: demo papers*, pages 1711–1712. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[20] Edwin PD Pednault. Formulating multiagent, dynamic-world problems in the classical planning framework. *Reasoning about actions and plans*, pages 47–82, 1987.

[21] Raymond Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64(2):337–351, 1993.

[22] Jussi Rintanen. Complexity of planning with partial observability. In *ICAPS*, pages 345–354, 2004.

[23] Nicholas Roy, Geoffrey J Gordon, and Sebastian Thrun. Finding approximate pomdp solutions through belief compression. *J. Artif. Intell. Res.(JAIR)*, 23:1–40, 2005.

[24] Stuart Russell and Peter Norvig. Ai a modern approach. *Learning*, 2(3):4, 2005.

[25] Scott Sanner. Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. Australian National University*, 2010.

[26] Yi Wang and Joohyung Lee. Handling uncertainty in answer set programming. In *AAAI*, pages 4218–4219, 2015.

[27] Håkan LS Younes and Michael L Littman. Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162*, 2004.

[28] VA Ziparo, L Iocchi, D Nardi, PF Palamara, and H Costelha. Pnp: A formal model for representation and execution of multi-robot plans. In *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 79–86.

[29] Vittorio Amos Ziparo and Luca Iocchi. Petri net plans. In *Proceedings of Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA)*, pages 267–290, 2006.