2 Formalisms

2.1 STRIPS (Classical Planning)

STRIPS[3] 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.
An expanded version of STRIPS is Action Description Language (ADL)[5], 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.

2.2 PDDL

PDDL[6] is the standard de facto for planning domains description. It generalizes both STRIPS[3] and Action Description Language (ADL [5]).
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 [6] 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. The most notable are:

- PPDDL[15] is a probabilistic version of PDDL.
  It extends PDDL 2.1 with probabilistic effects, reward fluents, goal rewards and goal-achieved fluents. This allows realising Markov Decision Process (MDP) planning in a fully observable environment with uncertainty in state-transitions
- RDDL[16] 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[17] add to PDDL some constructs to represent the incomplete knowledge of the agent about the environment.
  It is discussed in next section.

2.3 $ALCK_{NF}$

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 [14].
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.

2.4 KPDDL

KPDDL[17] 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.

2.5 Answer Set Programming (ASP)

Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems. [9] [10]
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 2.2).

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 [9] 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"*.
(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 [11] and [12].
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 [13] where Markov Logic Networks (MLN) are used.

2.6 Situation Calculus

Situation Calculus[7][8] 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**.

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

  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.

2.7 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 $\pi : S \rightarrow A$ 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 2.8).

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 $C : S \times A \rightarrow \mathbb{R}$ and a reward function as $R : S \rightarrow \mathbb{R}$, 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:

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

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

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

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.

2.8 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 2.7, 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 2.7. 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:

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

2.9 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 [14]).

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 [18] and [19], where particle filters and PCA are respectively used to represent the belief state compactly or exploit its properties.