4 General Planning Framework

When designing intelligent systems, an important feature is the ability to make decisions and act accordingly. To act, one should plan ahead. This is why the field of automated planning is being actively researched in order to find efficient algorithms to find the best course of action in any given situation. The previous chapter allowed to lay the basis of knowledge representation. How knowledge about the planning domains are represented is a main factor to take into account in order to conceive most planning algorithm.

konwledge est pluriel ?

the way to represent the knowledge in the planning domains is an important factor

Automated planning really started being formally investigated after the creation of the Stanford Research Institute Problem Solver (STRIPS) by Fikes and Nilsson (1971). This is one of the most influential planners, not because of its algorithm but because of its input language. Any planning system needs a way to express the information related to the input problem. Any language done for this purpose is called an *action language*. STRIPS will be mostly remembered for its eponymous action language that is at the base of any modern derivatives.

All action languages are based on mainly two notions: *actions* and *states*. A state is a set of *fluents* that describe aspects of the world modeled by the domain. Each action has a logic formula over states that allows its correct execution. This requirement is called *precondition*. The mirror image of this notion is called possible *effects* which are logic formulas that are enforced on the current state after the action is executed. The domain is completed with a problem, most of the time specified in a separate file. The problem basically contains two states: the *initial* and *goal* states.

PLAN du chapitre à présenter

4.1 Illustration

To illustrate how automated planners works, we introduce a typical planning problem called **block world**.

Example 19. In this example, a robotic grabbing arm tries to stack blocks on a table in a specific order. The arm is only capable of handling one block at a time. We suppose that the table is large enough so that all the blocks can be put on it without any stacks. Figure 4.1 illustrates the setup of this domain.

The possible actions are `pickup`, `putdown`, `stack` and `unstack`. There are at least three fluents needed:

- one to state if a given block is `down` on the table,
- one to specify which block is `held` at any moment and
- one to describe which block is stacked `on` which block.

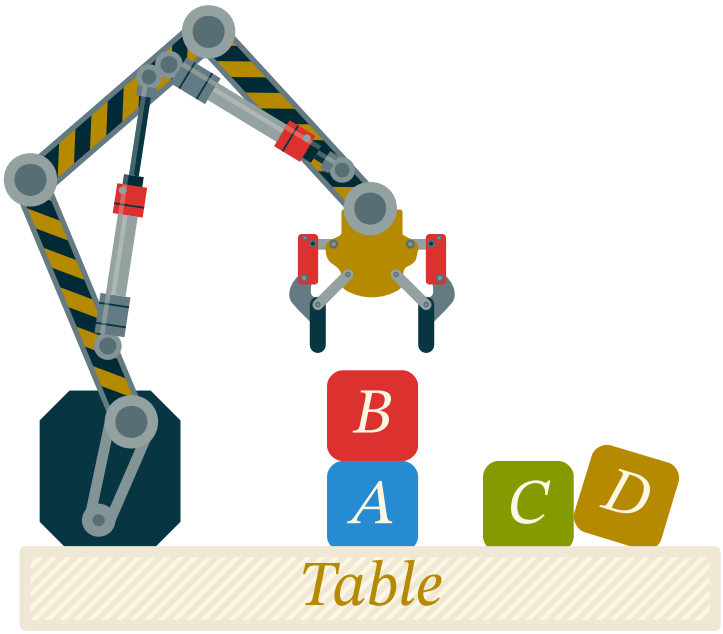


Figure 4.1: The block world domain setup.

We also need a special block to state when `noblock` is held or on top of another block. This block is a constant.

The knowledge we just described is called *planning domain*.

In that example, the initial state is described as stacks and a set of blocks directly on the table. The goal state is usually the specification of one or many stacks that must be present on the table. This part of the description is called *planning problem*.

In order to solve it we must find a valid sequence of actions called a *plan*. If this plan can be executed in the initial state and result in the goal state, it is called a *solution* of the planning problem. To be executed, each action must be done in a state satisfying its precondition and will alter that state according to its effects. A plan can be executed if all its actions can be executed in the sequence of the plan.

Example 20. For example, in the block world domain we can have an initial state with the $blockB$ on top of $blockA$ and the $blockC$ being on the table. In figure 4.2, we give a plan solution to the problem consisting of having the stack $\langle blockA, blockB, blockC \rangle$ from that initial state.

je ne comprends pas

Every automated planner aims to find at least one such solution in any way shape or form in the least amount of time with the best plan quality. The quality of a plan is often measured by how hard it is to execute, whether by its execution time or by the resources needed to accomplish it. This metric is often called *cost* of a plan and is often simply the sum of the costs of its actions.

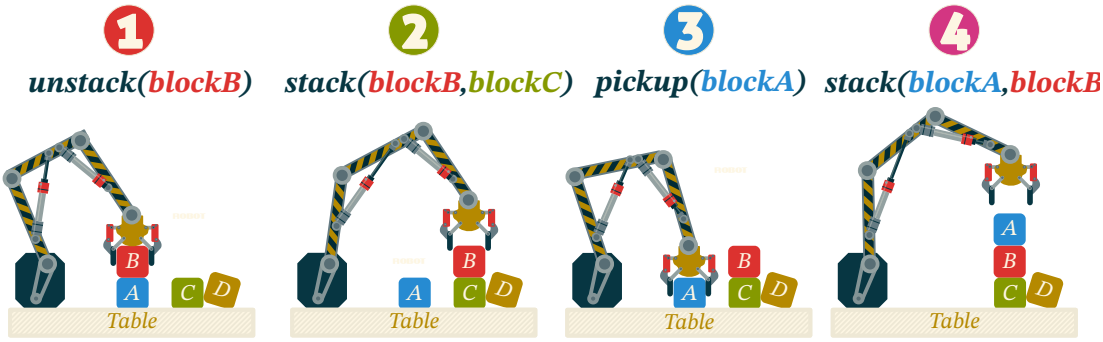


Figure 4.2: An example of a solution to a planning problem with a goal that requires three blocks stacked in alphabetical order.

Automated planning is very diverse. A lot of paradigms shift the definition of the domain, actions and even plan to widely varying extents. This is the reason why making a general planning formalism was deemed so hard or even impossible:

> "*It would be unreasonable to assume there is one single compact and correct syntax for specifying all useful planning problems.*" Sanner (2010)

Indeed, the block world example domain we give is mostly theoretical since there is infinitely more subtlety into this problem such as mechatronic engineering, balancing issues and partial ability to observe the environment and predict its evolution as well as failure in the execution. In our example, we didn't mention the misplaced $blockD$ that could very well interfere with any execution in unpredictable ways. This is why so many planning paradigms exist and why they are all so diverse: they try to address an infinitely complex problem, one sub-problem at a time. In doing so we lose the general view of the problem and by simply stating that this is the only way to resolve it we close ourselves to other approaches that can become successful. Like once said:

> "*The easiest way to solve a problem is to deny it exists.*" Asimov (1973)

define/design

provide

However, in the next section we aim to create such a general planning formalism. The main goal is to design the automated planning community with a general unifying framework ~~it so badly needs.~~ INUTILE

4.2 Formalism

pas clair: dans cette seciton, tu proposes un formalisme OU tu expliques ce qu'est le planning? —> après lecture j'en ai compris que tu présentes les éléments de la planif classiques en utilisant ton formalisme

In this section, a general formalism of automated planning is proposed. The goal is to explain what is planning and how it works. First we must express the knowledge domain formalism, then we describe how problems are represented and lastly how a general planning algorithm can be envisioned.

4.2.1 Planning domain

In order to conceive a general formalism for planning domains, we base its definition on the formalism of SELF. This means that all parts of the domain must be a member of the universe of discourse $\mathbb{U}$.

4.2.1.1 Fluents

First, we need to define the smallest unit of knowledge in planning, the fluents.

Definition 30 (Fluent)**.** A planning fluent is a predicate $f \in F$.

Fluents are signed. Negative fluents are noted $\neg f$ and behave as a logical complement. We do not use the closed world hypothesis: fluents are only satisfied when another compatible fluent is provided.

The name "fluent" comes from their fluctuating value. Indeed the truth value of a fluent is meant to vary with time and specifically by acting on it. In this formalism we represent fluents using either parameterized entities or using statements for binary fluents.

our domain block wolrd example les préciser countless fluents:non défini

Example 21. In our example we have four predicates. They can form countless fluents like $held(no-block), on(blockA, blockB)$ or $\neg down(blockA)$. Their when expressing a fluent we suppose its truth value is $\top$ and denote falsehood using the negation $\neg$.

Then ?

4.2.1.2 States

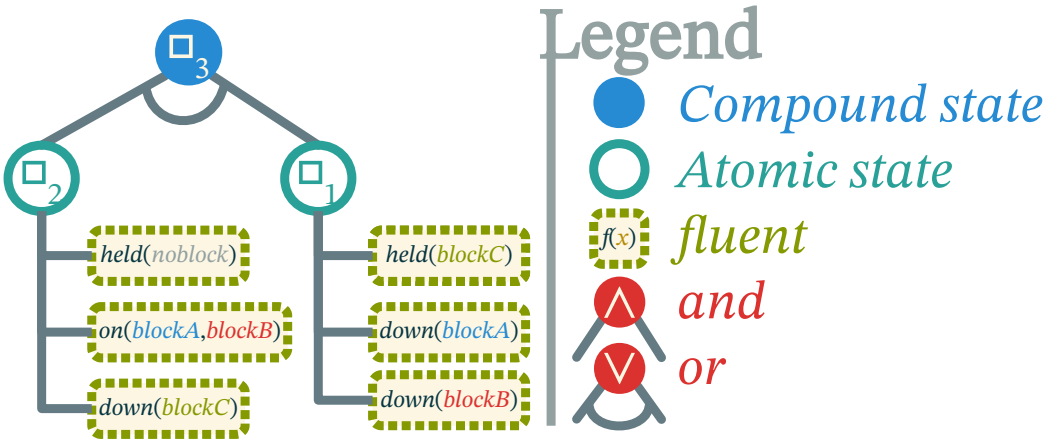
When expressing states, we need a formalism to express sets of fluents as formulas.

Definition 31 (State)**.** A state is a logical formula of fluents. Since all logical formulas can be reduced to a simple form using only $\wedge$, $\vee$, and $\neg$, we can represent states as *and/or trees*. This means that the leaves are fluents and the other nodes are states. We note states using small squares $\square$ as it is often the symbol used in the representation of automates and grafcets.

Example 22. In the domain block world, we can express a couple of states as :

- $\square_1 = held(noblock) \wedge on(blockA, blockB) \wedge down(blockC)$
- $\square_2 = held(blockC) \wedge down(blockA) \wedge down(blockB)$

In such a case, both state $\square_1$ and $\square_2$ have their truth value being the conjunction of all their fluents. We can express a disjunction in the following way: $\square_3 = \square_1 \vee \square_2$. In that case $\square_3$ is the root of the and/or tree and all its direct children are or vertices. The states $\square_1$ and $\square_2$ have their children as *and vertices*. All the leaves are fluents.



tu dois citer les figures dans ton texte !

Figure 4.3: Example of a state encoded as an and/or tree.

4.2.1.3 Verification and binding constraints

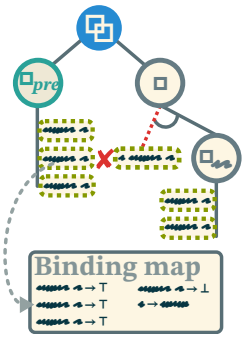


Figure 4.4: Example of verification and/or tree with binding constraints

When planning, there are two operations that are usually done on states: verify if a precondition fits a given state and then apply the effects of an action. In our model we consider preconditions and effects as states.

dire pourquoi tu fais ca (es-tu le seul ? quel avantage par rapport aux auttres manières (qui font quoi ?))

The verification is the operation $\square_{pre} \models \square$ that has either no value when the verification fails or a binding map for variables and fluents with their respective values.

The algorithm is a regular and/or tree exploration and evaluation applied on the state $\square = \square_{pre} \wedge \square$. During the valuation, if an inconsistency is found then the algorithm returns nothing. Otherwise, at each node of the tree, the algorithm will populate the binding map and verify if the truth value of the node holds under those constraints. All quantified variables are also registered in the binding map to enforce coherence in the root state. If the node is a state, the algorithm recursively applies until it reaches fluents. Once $\square$ is valuated as true, the binding map is returned.

Evaluation ?

Example 23. Using previously defined example states $\square_{1,2,3}$, and adding the following:

- $\square_4(x) = \{held(noblock), down(x)\}$ and
- $\square_5(y) = \{held(y), \neg down(y)\}$,

We can express a few examples of fluent verification:

- $held(noblock) \models held(x) = \{x = noblock\}$
- $\neg held(x) \models held(x) = \varnothing$

la figure n'est pas citée et on ne comprends pas à quoi elle sert / ce qu'elle est censée représenter (ton texte à gauche est compréhensible , si cette figure ne sert qu'à l'illustrer l'enlever elle ne sert à rien; si c'est une sorte d'exemple fil rouge la mettre en plus grosse et bien expliuer dans la légende) + Cest DEJA UNE REMARQUE FAITE PAR SAMIR ET QUE TU N'AS PAS CORRIGE !!!!

4.2.2 Application

? application of the action ? of effect state ?
je ne comprends pas pourquoi c'est séparé du 4.2.1 ?

Once the verification is done, the binding map is kept until the planner needs to apply the action to the state. The application of an effect state is noted $\square_{eff}(\square) = \square'$ and is very similar to the verification. The algorithm will traverse the state $\square$ and use the

c'est quoi ? Ca n'a pas été défini

idem: illisible, incompréhensible (car non cité dans le texte
qui y fait référence) —> DEJA REMARQUE DE SAMIR NON CORRIGE
L'idée d'illustrer sur les graphs est bien, mais mettre en plus gros, et expliquer l'exemple

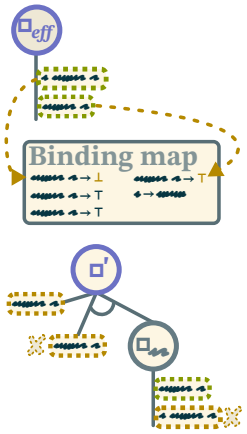


Figure 4.5: Example of application of a state with the updated binding constraints

binding map to force the values inside it. The binding map is previously completed using $\square_{eff}$ to enforce the application of its new value in the current state. This leads to changing the state $\square$ progressively into $\square'$ and the application algorithm will return this state.

4.2.2.1 Actions

Actions are the main mechanism behind automated planning, they describe what can be done and how it can be done.

Definition 32 (Action)**.** An action is a parameterized tuple $a(args) = \langle pre, eff, \gamma, ¢, d, \mathbb{P}, \mathbb{\Pi} \rangle$ where:

- *pre* and *eff* are states that are respectively the **preconditions and the effects** of the action.
- γ is the state representing the **constraints**.
- $¢$ is the intrinsic **cost** of the action.
- d is the intrinsic **duration** of the action.
- $\mathbb{P}$ is the prior **probability** of the action succeeding.
- $\mathbb{\Pi}$ is a set of **methods** that decompose the action into smaller simpler ones.

different

Operators take many names in difference planning paradigm: actions, steps, tasks, etc. In our case we call operators, all fully lifted actions and actions are all the possible instances (including operators).

In order to be more generalist, we allow in the constraints description, any time constraints, equalities or inequalities, as well as probabilistic distributions. These constraints can also express derived predicates. It is even possible to place arbitrary constraints on order and selection of actions.

Actions are often represented as state operators that can be applied in a given state to alter it. The application of actions is done by using the action as a relation on the set of states $a : \square \to \square$ defined as follows:

$$a(\square) = \begin{cases} \varnothing, & \text{if } pre \nvDash \square = \varnothing \\ eff(\square), & \text{using the binding map otherwise} \end{cases}$$

Example 24. A useful action we can define from previously defined states is the following:

$$pickup(x) = \langle \square_4(x), \square_5(x), (x : Block), 1.0¢, 3.5s, 75\%, \varnothing \rangle$$

That action can pick up a block x in 3.5 seconds using a cost of 1.0 with a prior success probability of 75%.

c'est franchement pas intuitif :
- il faut expliquer ta figure dire que c'est un diag de Vann ne suffit pas: dans un diag de Venn basique il y a juste des cercles , ici il y a plein d'autres symboles (flèches, pointillé) : c'est quoi ? tous tes cercles représentent des ensembles ?
- une image doit aider à la compréhension, si elle rajoute de l'incompréhension ou du travail de déchiffrage pour le lecteur, ca n'a pas vraiment d'intérêt. Ton image a pour objectif de synthétiser les éléments de SELF, mais il faut qu'elle soit simple et l'expliquer !
- cette remarque vaut aussi pour ton chapitre précédent qui introduit déjà l'image pour SELF!
+ C est DEJA UNE REMARQUE DE SAMIR NON PRISES EN COMTPE

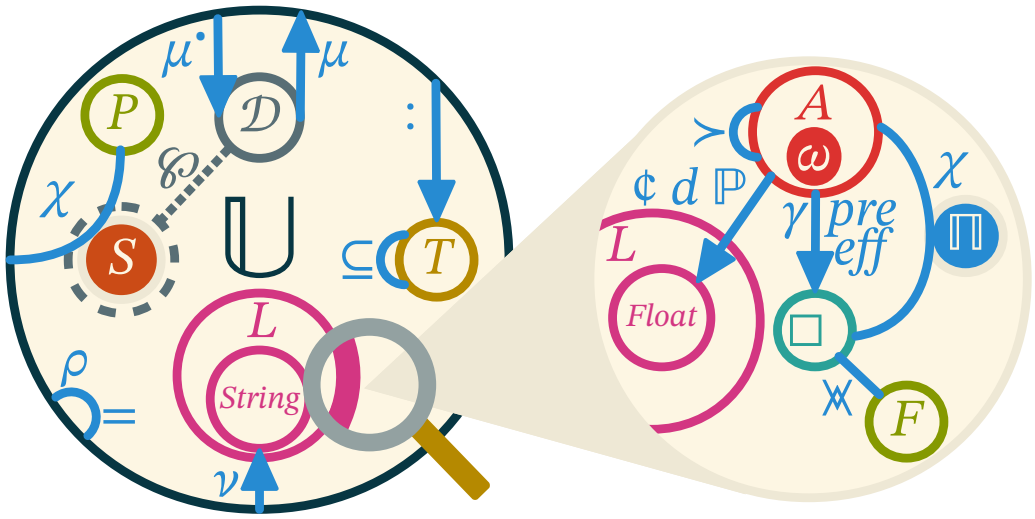


Figure 4.6: Venn diagram extended from the one from SELF to add all planning knowledge representation.

pourquoi elle apparait ici alors que tu ne la cites qu'à la fin du 4.2 ???

4.2.2.2 Domain

The planning domain specifies the allowed operators that can be used to plan and all the fluents they use as preconditions and effects.

Definition 33 (Domain)**.** A planning domain $\mathcal{D}$ is a set of **operators** which are fully lifted *actions*, along with all the relations and entities needed to describe their pre-conditions and effects.

Example 25. In the previous examples the domain was named block world. It consists in four actions: $pickup$, $putdown$, $stack$ and $unstack$. Usually the domain is self contained, meaning that all fluents, types, constants and operators are contained in it.

4.2.3 Planning problem

The aim of an automated planner is to find a plan to satisfy the goal. This plan can be of multiple forms, and there can even be multiple plans that meet the demand of the problem.

4.2.3.1 Solution to Planning Problems

Definition 34 (Partial Plan / Method)**.** A partially ordered plan is an *acyclic* directed graph $\pi = (A_\pi, E)$, with:

- A_π the set of **steps** of the plan as vertices. A step is an action belonging in the plan. A_π must contain an initial step a^0_π and goal step a^*_π as convenience for certain planning paradigms.

- E the set of **causal links** of the plan as edges. We note $l = a_s \xrightarrow{\square} a_t$ the link between its source a_s and its target a_t caused by the set of fluents $\square$. If $\square = \varnothing$ then the link is used as an ordering constraint.

With this def any kind of plans can be expressed

This definition can express any kind of plans, either temporal, fully or partially ordered or even hierarchical plans (using the methods of the actions $\mathbb{M}$). It can even express diverse planning results.

The notation can be reminicent of functional affectation and it is on purpose. Indeed, those links can be seen as relations that only affect their source to their target and the plan is a graph with its adjacence function being the combination of all links.

In our framework, *ordering constraints* are defined as the transitive cover of causal links over the set of steps. We note ordering constraints: $a_a \succ a_s$, with a_a being *anterior* to its *successor* a_s. Ordering constraints cannot form cycles, meaning that the steps must be different and that the successor cannot also be anterior to its anterior steps: $a_a \neq a_s \wedge a_s \not\succ a_a$. If we need to enforce order, we simply add a link without specifying a cause. The use of graphs and implicit order constraints help to simplify the model while maintaining its properties. Totally ordered plans are made by specifying links between all successive actions of the sequence.

s (c'est the use le sujet ?)

Example 26. In the section 4.1, we described a classical fully ordered plan, illustrated in figure 4.2. A partially ordered plan has a tree-like structure except that it also meets in a "sink" vertex (goal step). We explicit this structure in figure 4.7.

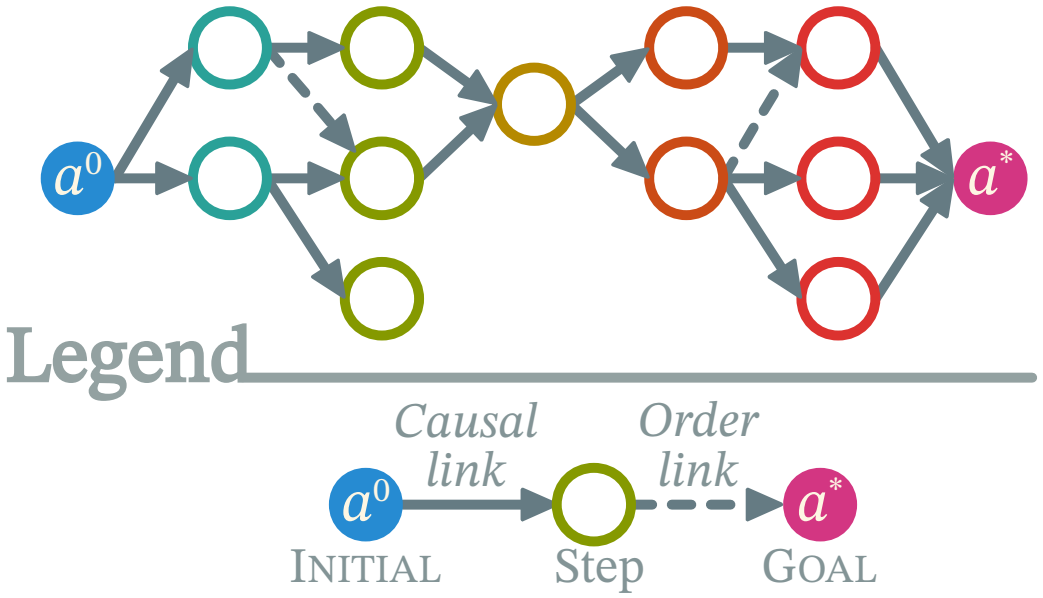


Figure 4.7: Structure of a partially ordered plan.

illustres le plan partiellement ordonné su rl'exemple blockworld

4.2.3.2 Planning Problem

With this formalism, the problem is very simplified but still general.

planning problem

Definition 35 (Problem)**.** The planning problem is defined as the **root operator** ω which methods are potential solutions of the problem. Its preconditions and effects are respectively used as initial state and goal description.

CE SONT DES REMARQUES FAITES PAR SAMIR NON CORRIGE

pas clair: tu veux dire quoi par des space homogeneous ? pourquoi actuellement ca ne l'est pas ?

As actions are very general, it is interesting to make the problem and domain space homogenous by using an action to describe any problem. Most of the specific notions of this framework are optional. Any planner using it will probably define what features it supports when compiling input domains and problems.

illustrer ca n'est pas évident ca : comment tu peux utliser une suele action pour décrire tout type de problem ????

All notions explained so far are represented in the figure 4.6 adding to the SELF Venn diagram.

il faut préciser « most of the specifi notions» c'est trop flou (et c'est quoi que tu appelles « notion »?): on peut avoir un planner qui ne définit pas le but par exemple?

4.3 Planning search

A general planning algorithm can be described as a guided exploration of a search space. The detailed structure of the search space as well as search iterators is dependent on the planning paradigm used and therefore are parameters of the algorithm.

oarler de ce que tu vas présenter après (contraintes temporelle ou sur la solution)

4.3.1 Search space

Definition 36 (Planner)**.** A planning algorithm, often called planner, is an exploration of a search space $\mathbb{S}$ partially ordered by an iterator $\chi_{\mathbb{S}}$ guided by a heuristic h. From any problem $\mathbb{p}$ every planner can derive two pieces of information immediately:

- the starting point $\mathbb{s}_0 \in \mathbb{S}$ and
- the solution predicate $?_{\mathbb{s}^*}$ that gives the validity of any potential solution in the search space.

Formally the problem can be viewed as a path-finding problem in the directed graph $g_{\mathbb{S}}$ formed by the vertex set $\mathbb{S}$ and the adjacency function $\chi_{\mathbb{S}}$. The set of solutions is therefore expressed as:

$$\mathbb{S}^* = \left\{ \mathbb{s}^* : \langle \mathbb{s}_0, \mathbb{s}^* \rangle \in \chi_{\mathbb{S}}^+(\mathbb{s}_0) \wedge ?_{\mathbb{s}^*} \right\}$$

We note a provided heuristic $h(\mathbb{s})$. It gives off the shortest predicted distance to any point of the solution space. The exploration is guided by it by minimizing its value.

The search in automated planning can have a lot of requirements. Often time is limited and results may need to meet a certain set of specifications.

lesquelles ? c'est ce que tu présentes après (solution constraints, temporal constraints , … ????)

4.3.2 Solution constraints

As finding a plan is computationally expensive, it is sometimes better to try to find either a more generally applicable plan or a set of alternatives. This is especially important in the case of execution monitoring or human interactions as proposing several relevant solutions to pick from is a very interesting feature.

For this, one will prefer either probabilistic or diverse planning. The main problem is the additional parameters required and the change of behavior of the algorithm. In order to handle such formalism, the planning algorithm needs additional specifications.

c'est à dire ? il faudrait peut être présenté probabilistic planninf et diverse planning pour que l'on comprenne

We note $\gamma_{\mathbb{S}}$ the set of constraints on the nature of the solution. This contains notably the following optional elements:

- Δ the plan deviation metric to compare how much two plans are different.
- k is the number of expected different solutions. This simply makes the process return when either it found k solutions or when it determined that $k > |\mathbb{S}^*|$.

For probabilistic planning, all elements of the probability distributions used are typically included in the domain but the solution constraints parameter can also contain all required information from any planning paradigm, present or future as long as it is expressible in SELF.

on comprends rien ici, en plus tu parles de distirbutions de proba pour probabilistic planning
mais on ne sait pas ce que c'est, tu n'as pas encore présenté cela ! De même pour divers eplanning !

4.3.3 Temporal constraints

Another aspect of planning lies in its timing. Indeed sometimes acting needs to be done before a deadline and planning are useful only during a finite timeframe. This is done even in optimal planning as researchers evaluating algorithms often need to set a timeout in order to be able to complete a study in a reasonable amount of time. Indeed, often in efficiency graphs, planning instances are stopped after a defined amount of time.

donner un exemple pour qu'on comprenne mieux

This time component is quite important as it often determines the planning paradigm used. It is expressed as two parameters:

fitTing

- $t_{\mathbb{S}}$ the allotted time for the algorithm to find at least a fiting solution.
- t^* additional time for plan optimization.

This means that if the planner cannot find a fitting solution in time it will either return a timeout error or a partial or abstract solution that needs to be refined. Anytime planners will also use the extra time parameter to optimize the solution some more. If the amount of time is either unknown or unrestricted the parameters can be omitted and their value will be set to infinity.

c'est quoi ca ?

tu veux dire quoi ici: un planner general peut trouver une solution à tout type de pb de planning ? quels sont ces différents types de problèmes (—> ca pourrait être quelque chose que tu présentes en synthèse dans une conclusion du 4.3 par exemple …. les conclusions manquent cruellement dans ton document pour récapituler tout ce que tu dis !!!) ?

4.4 General planner

A general planner $\mathbb{\Pi}^*(g_{\mathbb{S}}, \mathbb{s}_0, ?_{\mathbb{s}^*}, h, \gamma_{\mathbb{S}}, \mathcal{D}, t_{\mathbb{S}}, t^*)$ is an algorithm that can find solutions using any valid instance of the planning formalism.

je ne comprends pas,tu parles de quoi ici: de l'algo de planning que tu utilises ensuite dans ta thèse ?

For the planning algorithm itself, we simply use a parameterized instance of any search algorithms. In our case we chose the K* algorithm (Aljazzar and Leue 2011, alg. 1). This algorithm uses the classical algorithm A* to explore the graph while using Dijkstra on some sections to find the k shortest paths. The parameters are as follows: $K^*(g_{\mathbb{S}}, \mathbb{s}_0, ?_{\mathbb{s}^*}, h)$. In this case, the solution predicate contains the solution and time constraints.

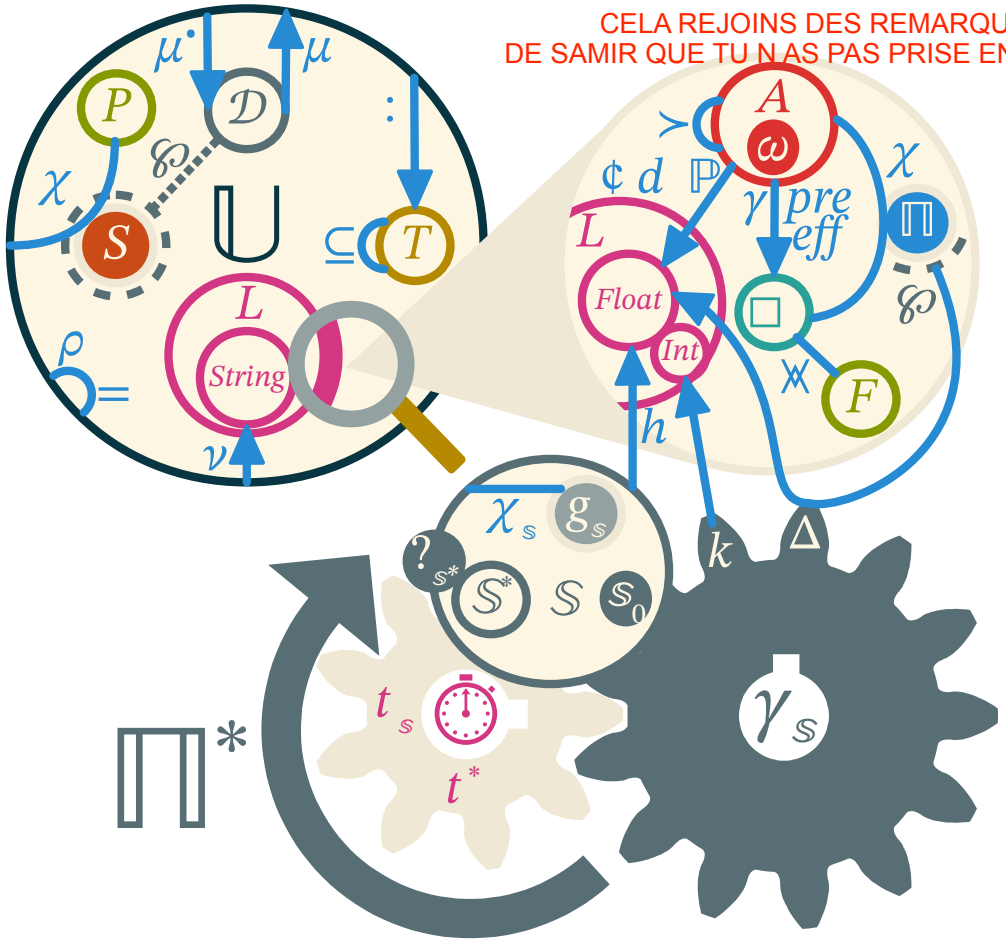
tu utilise comme instance K^* (à quoi sert de dire « any search algo » alors que tu en choisit un ?)

K* ???

Of course this algorithm is merely an example of a general planner algorithm. The algorithm has been chosen to be general and its efficiency hasn't been tested.

je ne comprends pas ce que tu veux dire dans le 4.4. Tu définis un general planner, puis tu donnes un exemple d'un general planner qui est K* ???? à quel type de pb K* permet de trouver une solution?

CELA REJOINS DES REMARQUES DE SAMIR QUE TU N'AS PAS PRISE EN COMPTE



OK belle figure mais ca sert à quoi si on ne comprends pas ??? (CF REMARQUE DE SAMIR NON CORRIG ENCORE ET ENCORE…)

Figure 4.8: Venn diagram extended with general planning formalism.

Est-ce vraiment le bon titre de section ? pourquoi tu présentes le formalisme classique APRES avoir présenté le tien??? est-ce que tu présentes la même chose que précédemment ?
—> après lecture, tu ne présentes pas le formalisme classique ici, mais les différents types d'automated planning / planning paradigm ?
Pourquoi on ne retrouve pas tous ceux du 4.6 ?

4.5 Classical Formalism

One of the most comprehensive work on summarizing the automated planning domain was done by Ghallab *et al.* (2004). This book explains the different planning paradigm of its time and gives formal description of some of them. This work has been updated later (Ghallab *et al.* 2016) to reflect the changes occurring in the planning community.

4.5.1 State-transition planning

The most classical representation of automated planning is using the state transition approach: actions are operators on the set of states and a plan is a finite-state automaton. We can also see any planning problem as either a graph exploration problem or even a constraint satisfaction problem. In any way that problem is isomorph to its original formulation and most efficient algorithms use a derivative of A* exploration techniques on the state space.

The parameters for state space planning are trivial:

$$\mathbb{\Pi}^*_{\square} = \mathbb{\Pi}^* ((\square, A), \mathit{pre}(\omega), \mathit{eff}(\omega))$$

This formulation takes advantage of several tools previously described. It uses the partial application of a function to omit the last parameters. It also defines the search graph using the set of all states $\square$ as the vertices set and the set of all available actions A as the set of edges while considering actions as relations that can be applied to states to make the search progress toward an eventual solution. We also use the binary nature of states to use the effects of the root operator as the solution predicate.

Usually, we would set both $t_{\mathbb{S}}$ and t^* to an infinite amount as it is often the case for such planners. These parameters are left to the user of the planner.

State based planning usually suppose[S] total knowledge of the state space and action behavior. No concurrence or time constraints are expressed and the state and action space must be finite as well as the resulting state graph. This process is also deterministic and doesn't allow uncertainty. The result of such planning is a totally ordered sequence of actions called a plan. The total order needs to be enforced even if it is unnecessary.

All those features are important in practice and lead to other planning paradigms that are more complex than classical state-based planning.

4.5.2 Plan space planning

Plan Space Planning (PSP) is a form of planning that uses plan space as its search space. It starts with an empty plan and tries to iteratively refine that plan into a solution.

The transformation into a general planner is more complicated than for state-based planning as the progress is made through refinements. We note the set of possible

refinement of a given plan $r = \mathbb{m} \rightarrow \{\!|\odot : \otimes(\mathbb{m})|\!\}$. Each refinement is a new plan in which we fixed a *flaw* using one of the possible *resolvers* (see **LATER**).

mettre ref du numéro de section

$$\mathbb{\Pi}^*_{\mathbb{\Pi}} = \mathbb{\Pi}^* \left((\mathbb{\Pi}, \{r(\mathbb{m}) : \mathbb{m} \in \mathbb{\Pi}\}), (\{a^0, a^*\}, \{a^0 \rightarrow a^*\}), \otimes(\mathbb{s}) = \varnothing \right)$$

with a^0 and a^* being the initial and goal steps of the plan corresponding $\mathbb{s}_0$ such that $\mathit{eff}(a^0) = \mathit{pre}(\omega)$ and $\mathit{pre}(a^*) = \mathit{eff}(\omega)$. The iterator is all the possible resolutions of all flaws on any plan in the search space and the solution predicate is true when the plan has no more flaws.

Details about flaws, resolvers and the overall Partial Order Causal Links (POCL) algorithm will be presented **LATER**.

This approach can usually give a partial plan if we set $t_{\mathbb{s}}$ too low for the algorithm to complete. This plan is not a solution but can eventually be usefull ~~as~~ an approximation for certain use cases (like intent recognition, see **LATER**).

4.5.3 Case based planning

Another plan oriented planning is called Case-Based Planning (CBP). This kind of planning relies on a library $\mathcal{C}$ of already complete plans and try to find the most appropriate one to repair.

$$\mathbb{\Pi}_{\mathcal{C}} = \mathbb{\Pi}^* \left((\mathcal{C}, \odot), (\mathbb{m} : \mathbb{m} \in \mathcal{C} \wedge \Delta(\mathbb{m}, \omega) = \{\!|min : \mathcal{C} \times \{\omega\}|\!\}, \mathbb{s}(\mathit{pre}(\omega)) \neq \varnothing \right)$$

The planner selects a plan that fits the best efficiently with the initial and goal state of the problem. This plan is then repaired and validated iteratively. The problem with this approach is that it may be unable to find a valid plan or might need to populate and maintain a good plan library. For such case an auxiliary planner is used (preferably a diverse planner; that gives several solution).

tu n'as jamais dis avant ce qu'était un diverse planner

4.5.4 Probabilistic planning

Probabilistic planning tries to deal with uncertainty by generating a policy instead of a plan. The initial problem holds probability laws that govern the execution of any actions. It is sometimes accompanied with a reward function instead of a deterministic goal. We use the set of a pair of states with applicable actions as the search space: $\square + A = \{\langle \square, a \rangle : a \in A \wedge \square \in \square \wedge a(\square) \neq \varnothing\}$. We can also note the policy iteration $pol = \langle \square, a \rangle \rightarrow \langle a(\square), \{\!|a' \in A \wedge a'(a(\square)) \neq \varnothing|\!\} \rangle$

$$\mathbb{\Pi}_{\mathbb{p}} = \mathbb{\Pi}^* \left((\square + A, pol), \langle \mathit{pre}(\omega), \{\!|a \in A \wedge a(\mathit{pre}(\omega)) \neq \varnothing|\!\} \rangle, \forall \mathbb{s}_1 \vDash \mathit{eff}(\omega) \right)$$

At each iteration a state is chosen from the frontier. The frontier is updated with the application of a non-deterministically chosen pair of the last policy insertion. The search stops when all elements in the frontier are goal states.

4.5.5 Hierarchical planning

Hierarchical Task Networks (HTN) are a totally different kind of planning paradigm. Instead of a goal description, HTN uses a root task that needs to be decomposed. The task decomposition is an operation that replaces a task (action) by one of its methods $\mathbb{\Pi}$.

$$\mathbb{\Pi}_\omega = \mathbb{\Pi}^* = ((\mathbb{\Pi}, \mathbb{s} \rightarrow \{\!\{\mathbb{\pi} \in \mathbb{\Pi}(\{\!\{a \in A_\mathbb{s} \wedge \mathbb{\Pi}(a) \neq \}\!\})\}\!\},), \omega, \forall a \in A_\mathbb{s} : \mathbb{\Pi}(a) = \varnothing$$

TODO: Explain more Oui ca devrait être déjà fait

4.6 Existing Languages and Frameworks

4.6.1 Classical

préciser : classical quoi ?

donner nom complet de ADL (c'est valable partout, aussi pour PPDL, RDDL, MAPL, …)

is ? y'a pas de verbe dans ta phrase

After STRIPS, one of the first languages to be introduced to express planning domains like ADL (Pednault 1989). That formalism adds negation and conjunctions into literals to STRIPS. It also drops the closed world hypothesis for an open world one: anything not stated in conditions (initials or action effects) is unknown.

The current standard was strongly inspired by Penberthy *et al.* (1992) and his UCPOP planner. Like STRIPS, UCPOP had a planning domain language that was probably the most expressive of its time. It differs from ADL by merging the add and delete lists in effects and to change both preconditions and effects of actions into logic formula instead of simple states.

The PDDL language was created for the first major automated planning competition hosted by AIPS in 1998 (Ghallab *et al.* 1998). It came along with syntax and solution checker written in Lisp. It was introduced as a way to standardize the notation of planning domains and problems so that libraries of standard problems can be used for benchmarks. The main goal of the language was to be able to express most of the planning problems of the time.

With time, the planning competitions became known under the name of International Planning Competitions (IPC) regularly hosted by the ICAPS conference. With each installment, the language evolved to address issues encountered the previous years. The current version of PDDL is 3.1 (Kovacs 2011). Its syntax goes similarly as described in **??** 4.1.

```
1 (define (domain <domain-name>)
2   (:requirements :<requirement-name>)
3   (:types <type-name>)
4   (:constants <constant-name> - <constant-type>)
5   (:predicates (<predicate-name> ?<var> - <var-type>))
6   (:functions (<function-name> ?<var> - <var-type>) - <function-type>)
7
8   (:action <action-name>
9       :parameters (?<var> - <var-type>)
```

```
10        :precondition (and (= (<function-name> ?<var>) <value>)
      (<predicate-name> ?<var>))
11        :effect
12        (and (not (<predicate-name> ?<var>))
13       (assign (<function-name> ?<var>) ?<var>)))
```

Listing 4.1: Simplified explanation of the syntax of PDDL.

PDDL uses the functional notation style of LISP. It defines usually two files: one for the domain and one for the problem instance. The domain describes constants, fluents and all possible actions. The problem lays the initial and goal states description.

Example 27. For example, consider the classic block world domain expressed in **??** 4.2. It uses a predicate to express whether a block is on the table because several blocks can be on the table at once. However it uses a 0-ary function to describe the one block allowed to be held at a time. The description of the stack of blocks is done with an unary function to give the block that is on top of another one. To be able to express the absence of blocks it uses a constant named `no-block`. All the actions described are pretty straightforward: `stack` and `unstack` make sure it is possible to add or remove a block before doing it and `pick-up` and `put-down` manages the handling operations.

oréciser la ligne où cela apparait

```
1  (define (domain BLOCKS-object-fluents)
2    (:requirements :typing :equality :object-fluents)
3    (:types block)
4    (:constants no-block - block)
5    (:predicates (on-table ?x - block))
6    (:functions (in-hand) - block
7    (on-block ?x - block) - block) ;;what is in top of block ?x
8
9    (:action pick-up
10       :parameters (?x - block)
11       :precondition (and (= (on-block ?x) no-block) (on-table ?x) (=
       (in-hand) no-block))
12       :effect
13       (and (not (on-table ?x))
14        (assign (in-hand) ?x)))
15
16   (:action put-down
17       :parameters (?x - block)
18       :precondition (= (in-hand) ?x)
19       :effect
20       (and (assign (in-hand) no-block)
21        (on-table ?x)))
22
23   (:action stack
24       :parameters (?x - block ?y - block)
25       :precondition (and (= (in-hand) ?x) (= (on-block ?y) no-block))
26       :effect
27       (and (assign (in-hand) no-block)
28        (assign (on-block ?y) ?x)))
29
30   (:action unstack
31       :parameters (?x - block ?y - block)
32       :precondition (and (= (on-block ?y) ?x) (= (on-block ?x) no-block)
       (= (in-hand) no-block))
33       :effect
34       (and (assign (in-hand) ?x)
```

```
35      (assign (on-block ?y) no-block))))
```

Listing 4.2: Classical PDDL 3.0 definition of the domain Block world

déjà dit avant

However, PDDL is far from a universal standard. Some efforts have been made to try and standardize the domain of automated planning in the form of optional requirements. The latest of the PDDL standard is the version 3.1 (Kovacs 2011). It has 18 atomic requirements as represented in figure 4.9. Most requirements are parts of PDDL that either increase the complexity of planning significantly or that require extra implementation effort to meet.

il y a plein d'éléments dans la figure qui ne sont pas compréhensible et non évident :
il faut expliquer ta figure

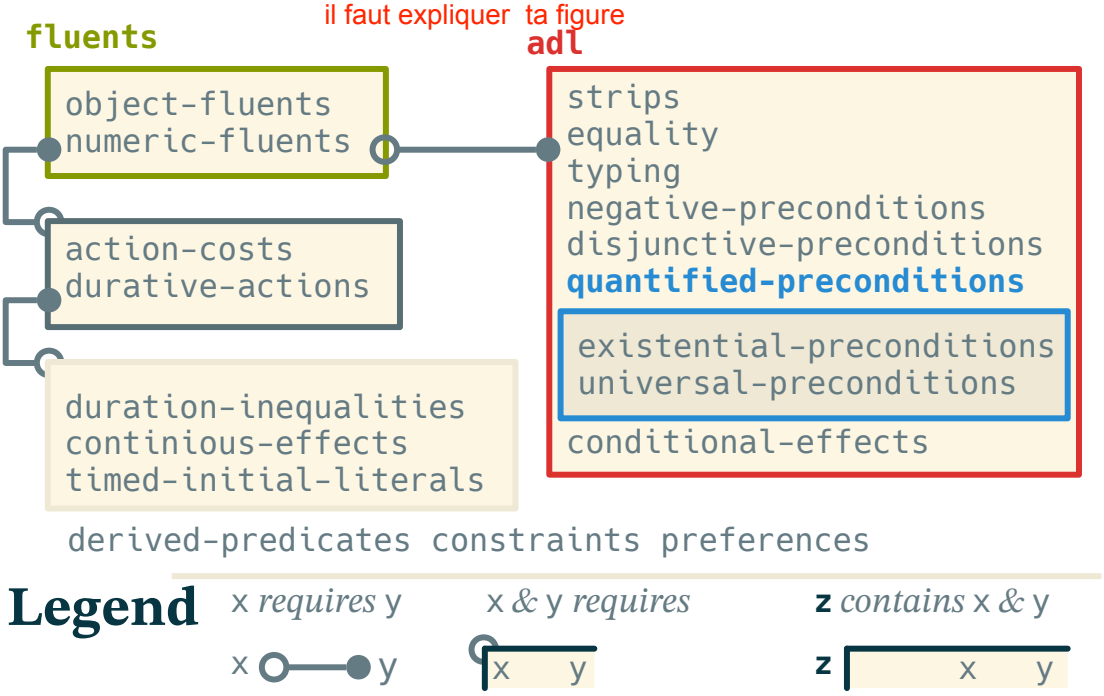


Figure 4.9: Dependencies and grouping of PDDL requirements.

donner un exemple pour justifier tes propos

donner un exemple

Even with that flexibility, PDDL is unable to cover all of automated planning paradigms. This caused most subdomains of automated planning to be left in a state similar to before PDDL: a collection of languages and derivatives that aren't interoperable. The reason for this is the fact that PDDL isn't expressive enough to encode more than a limited variation in action and fluent description.

Another problem is that PDDL isn't made to be used by planners to help with their planning process. Most planners will totally separate the compilation of PDDL before doing any planning, so much so that most planners of the latest IPC used a framework that translates PDDL into a useful form before planning, adding computation time to the planning process. The list of participating planners and their use of language is presented in table 4.1.

The domain is so diverse that attempts to unify it haven't succeeded so far. The main reason behind this is that some paradigms are vastly different from the classical planning description. Sometimes just adding a seemingly small feature like probabilities or plan reuse can make for a totally different planning problem. In the next section we

Table 4.1: Planners participating in the Classic track of the 2018 International Planning Competition (IPC). The table states whether the planner used a translation and a pre-processing system to handle PDDL. Most of the planners are based on FastDownward directly.

Name	Trans	Pre	Lang	Base	Rank
Delfi	Yes	Yes	C++	FD	1
Complementary	Yes	Yes	C++	FD	2
Planning-PDBs	Yes	Yes	C++	FD	3
Scorpion	Yes	Yes	C++	FD	4
FDMS	Yes	Yes	C++	FD	5
DecStar	Yes	Yes	C++	LAMA	6
Metis	Yes	Yes	C++	FD	7
MSP	Yes	Yes	Lisp	FD	8
Symple	Yes	Yes	C++	FD	9
Ma-plan	No	Yes	C	None	10

Dire ce que veut dire chaque colonne
Donner une ref pour chaque planner

describe planning paradigms and how they differ from classical planning along with their associated languages.

4.6.2 Temporality oriented

When planning, time can become a sensitive constraint. Some critical tasks may require to be completed within a certain time. Actions with duration are already a feature of PDDL 3.1. However, PDDL might not provide support for external events (i.e. events occurring independent from the agent). To do this one must use another language.

4.6.2.1 PDDL+

tu ne peux pas avoir 4.6.2.1 sans 4.6.2.2 ….

PDDL+ is an extension of PDDL 2.1 that handles process and events (Fox and Long 2002). It can be viewed as similar to PDDL 3.1 continuous effects but it differs on the expressivity. A process can have an effect on fluents at any time. They can happen either from the agent's own doing or being purely environmental. It might be possible in certain cases to model this using the durative actions, continuous effects and timed initial literals of PDDL 3.1.

In ?? 4.3, we reproduce an example from Fox and Long (2002). It shows the syntax of durative actions in PDDL+. The timed preconditions are also available in PDDL 3.1, but the `increase` and `decrease` rate of fluents is an exclusive feature of PDDL+.

```
1 (:durative-action downlink
2     :parameters (?r - recorder ?g - groundStation)
3     :duration (> ?duration 0)
4     :condition (and (at start (inView ?g))
5                     (over all (inView ?g))
6                     (over all (> (data ?r) 0)))
7     :effect (and (increase (downlinked)
8                            (* #t (transmissionRate ?g)))
9                  (decrease (data ?r)
10                           (* #t (transmissionRate ?g)))))
```

Listing 4.3: Example of PDDL+ durative action from Fox's paper.

détailler plus ton exemple que l'on voit où/comment sont exprimés les actions durative

The main issue with durative actions is that time becomes a continuous resource that may change the values of fluents. The search for a plan in that context has a higher complexity than regular planning.

4.6.3 Probabilistic

Sometimes, acting can become unpredictable. An action can fail for many reasons, from logical errors down to physical constraints. This call for a way to plan using probabilities with the ability to recover from any predicted failures. PDDL doesn't support using probabilities. That is why all IPC's tracks dealing with it always used another language than PDDL.

4.6.3.1 PPDDL

PPDDL is such a language. It was used during the 4th and 5th IPC for its probabilistic track (Younes and Littman 2004). It allows for probabilistic effects as demonstrated in **??** 4.4. The planner must take into account the probability when choosing an action. The plan must be the most likely to succeed. But even with the best plan, failure can occur. This is why probabilistic planning often gives policies instead of a plan. A policy dictates the best choice in any given state, failure or not. While this allows for much more resilient execution, computation of policies are exponentially harder than classical planning. Indeed the planner needs to take into account every outcome of every action in the plan and react accordingly.

```
(define (domain bomb-and-toilet)
    (:requirements :conditional-effects :probabilistic-effects)
    (:predicates (bomb-in-package ?pkg) (toilet-clogged)
                 (bomb-defused))
    (:action dunk-package
             :parameters (?pkg)
             :effect (and (when (bomb-in-package ?pkg)
                                (bomb-defused))
                          (probabilistic 0.05 (toilet-clogged)))))
```

Listing 4.4: Example of PPDDL use of probabilistic effects from Younes's paper.

détailler plus ton exemple que l'on voit où/comment sont exprimés les effets probabilistes

4.6.3.2 RDDL

Another language used by the 7th IPC's uncertainty track is RDDL (Sanner 2010). This language has been chosen because of its ability to express problems that are hard to encode in PDDL or PPDDL. Indeed, RDDL is capable of expressing Partially Observable Markovian Decision Process (POMDP) and Dynamic Bayesian Networks (DBN) in planning domains. This along with complex probability laws allows for easy implementation of most probabilistic planning problems. Its syntax differs greatly from PDDL, and seems closer to Scala or C++. An example is provided in **??** 4.5 from Sanner (2010). In it, we can see that actions in RDDL don't need preconditions or effects. In that case

the reward is the closest information to the classical goal and the action is simply a parameter that will influence the probability distribution of the events that conditioned the reward.

```
////////////////////////////////////////////////////////////////////////
// A simple propositional 2-slice DBN (variables are not parameterized).
//
// Author: Scott Sanner (ssanner [at] gmail.com)
////////////////////////////////////////////////////////////////////////
domain prop_dbn {

 requirements = { reward-deterministic };

 pvariables {
  p : { state-fluent,  bool, default = false };
  q : { state-fluent,  bool, default = false };
  r : { state-fluent,  bool, default = false };
  a : { action-fluent, bool, default = false };
 };

 cpfs {
  // Some standard Bernoulli conditional probability tables
  p' = if (p ^ r) then Bernoulli(.9) else Bernoulli(.3);

  q' = if (q ^ r) then Bernoulli(.9)
       else if (a) then Bernoulli(.3) else Bernoulli(.8);

  // KronDelta is like a DiracDelta, but for discrete data (boolean or
     int)
  r' = if (~q) then KronDelta(r) else KronDelta(r <=> q);
 };

 // A boolean functions as a 0/1 integer when a numerical value is needed
 reward = p + q - r; // a boolean functions as a 0/1 integer when a
    numerical value is needed
}

instance inst_dbn {

 domain = prop_dbn;
 init-state {
  p = true;  // could also just say 'p' by itself
  q = false; // default so unnecessary, could also say '~q' by itself
  r;         // same as r = true
 };

 max-nondef-actions = 1;
 horizon  = 20;
 discount = 0.9;
}
```

Listing 4.5: Example of RDDL syntax by Sanner.

idem expliquer un peu plus l'exemple,en tout cas ce qui t'intéresse ici (les actions probabilistes) sinon, on ne vas pas essayer de le comprendre ….

4.6.4 Multi-agent

Planning can also be a collective effort. In some cases, a system must account for other agents trying to either cooperate or compete in achieving similar goals. The problem

that arise is coordination. How to make a plan meant to be executed with several agents concurently ? Several multi-agent action languages have been proposed to answer that question.

4.6.4.1 MAPL

Another extension of PDDL 2.1, MAPL was introduced to handle synchronization of actions (Brenner 2003). This is done using modal operators over fluents. In that regard, MAPL is closer to the PDDL+ extension proposed earlier. It encodes durative actions that will later be integrated into the PDDL 3.0 standard. MAPL also introduce a synchronization mechanism using speech as a comunication vector. This seems very specific as explicit comunication isn't a requirement of collaborative work. ?? 4.6 is an example of the syntax of MAPL domains. PDDL 3.0 seems to share a similar syntax.

```
(:state-variables
  (pos ?a - agent) - location
  (connection ?p1 ?p2 - place) - road
  (clear ?r - road) - boolean)
(:durative-action Move
  :parameters (?a - agent ?dst - place)
  :duration (:= ?duration (interval 2 4))
  :condition
    (at start (clear (connection (pos ?a) ?dst)))
  :effect (and
    (at start (:= (pos ?a) (connection (pos ?a) ?dst)))
    (at end (:= (pos ?a) ?dst))))
```

Listing 4.6: Example of MAPL syntax by Brenner.

4.6.4.2 MA-PDDL

Another aspect of multi-agent planning is the ability to affect tasks and to manage interactions between agents efficiently. For this MA-PDDL seems more adapted than MAPL. It is an extension of PDDL 3.1, that makes easier to plan for a team of heteroneous agents (Kovács 2012). In the example in ?? 4.7, we can see how action can be affected to agents. While it makes the representation easier, it is possible to obtain similar effect by passing an agent object as parameters of an action in PDDL 3.1. More complex expressions are possible in MA-PDDL, like referencing the action of other agents in the preconditions of actions or the ability to affect different goals to different agents. Later on, MA-PDDL was extended with probabilistic capabilities inspired by PPDDL (Kovács and Dobrowiecki 2013).

```
(define (domain ma-lift-table)
(:requirements :equality :negative-preconditions
               :existential-preconditions :typing :multi-agent)
(:types agent) (:constants table)
(:predicates (lifted (?x - object) (at ?a - agent ?o - object))
(:action lift :agent ?a - agent :parameters ()
:precondition (and (not (lifted table)) (at ?a table)
              (exists (?b - agent)
               (and (not (= ?a ?b)) (at ?b table) (lift ?b))))
```

```
10 :effect (lifted table)))
```

Listing 4.7: Example of MA-PDDL syntax by Kovacs.
exlpiquer plus

4.6.5 Hierarchical

Another approach to planning is using Hierarchical Tasks Networks (HTN) to resolve some planning problems. Instead of searching to satisfy a goal, HTNs try to find a decomposition to a root task that fits the initial state requirements and that generate S an executable plan.

4.6.5.1 UMCP

One of the first planner to support HTN domains was UCMP by Erol *et al.* (1994). It uses Lisp like most of the early planning systems. Apparently PDDL was in part inspired by UCMP's syntax. Like for PDDL, the domain file describes action (called operators here) and their preconditions and effects (called postconditions). The syntax is exposed in **??** 4.8. The interesting part of that language is the way decomposition is handled. Each task is expressed as a set of methods. Each method has an expansion expression that specifies how the plan should be constructed. It also has a pseudo precondition with modal operators on the temporality of the validity of the literals.

```
1 (constants a b c table) ; declare constant symbols
2 (predicates on clear) ; declare predicate symbols
3 (compound-tasks move) ; declare compound task symbols
4 (primitive-tasks unstack dostack restack) ; declare primitive task symbols
5 (variables x y z) ; declare variable symbols
6
7 (operator unstack(x y)
8           :pre ((clear x)(on x y))
9           :post ((~on x y)(on x table)(clear y)))
10 (operator dostack (x y)
11           :pre ((clear x)(on x table)(clear y))
12           :post ((~on x table)(on x y)(~clear y)))
13 (operator restack (x y z)
14           :pre ((clear x)(on x y)(clear z))
15           :post ((~on x y)(~clear z)(clear y)(on x z)))
16
17 (declare-method move(x y z)
18                 :expansion ((n restack x y z))
19                 :formula (and (not (veq y table))
20                               (not (veq x table))
21                               (not (veq z table))
22                               (before (clear x) n)
23                               (before (clear z) n)
24                               (before (on x y) n)))
25
26 (declare-method move(x y z)
27                 :expansion ((n dostack x z))
28                 :formula (and (veq y table)
29                               (before (clear x) n)
30                               (before (on x y) n)))
```

Listing 4.8: Example of the syntax used by UCMP.
expliquer plus

4.6.5.2 SHOP2

The next HTN planner is SHOP2 by Nau *et al.* (2003). It remains to this day, one of the reference implementation of an HTN planner. The SHOP2 formalism is quite similar to UCMP's: each method has a signature, a precondition formula and eventually a decomposition description. This decomposition is a set of methods like in UCMP. The methods can also be partially ordered allowing more expressive plans. An example of the syntax of a method is given in ?? 4.9.

```
(:method
  ; head
    (transport-person ?p ?c2)
  ; precondition
    (and
      (at ?p ?c1)
      (aircraft ?a)
      (at ?a ?c3)
      (different ?c1 ?c3))
  ; subtasks
    (:ordered
      (move-aircraft ?a ?c1)
      (board ?p ?a ?c1)
      (move-aircraft ?a ?c2)
      (debark ?p ?a ?c2)))
```

Listing 4.9: Example of method in the SHOP2 language.

expliquer plus

4.6.5.3 HDDL

A more recent example of HTN formalism comes from the PANDA framework by Bercher *et al.* (2014). This framework is considered the current standard of HTN planning and allows for great flexibility in domain description. PANDA takes previous formalism and generalize them into a new language exposed in ?? 4.10. That language was called HDDL.

lesquels ?

S

```
(define (domain transport)
  (:requirements :typing :action-costs)
  (:types
        location target locatable - object
        vehicle package - locatable
        capacity-number - object
  )
  (:predicates
     (road ?l1 ?l2 - location)
     (at ?x - locatable ?v - location)
     (in ?x - package ?v - vehicle)
     (capacity ?v - vehicle ?s1 - capacity-number)
     (capacity-predecessor ?s1 ?s2 - capacity-number)
  )

  (:task deliver :parameters (?p - package ?l - location))
  (:task unload :parameters (?v - vehicle ?l - location ?p - package))

  (:method m-deliver
    :parameters (?p - package ?l1 ?l2 - location ?v - vehicle)
```

```
21      :task (deliver ?p ?l2)
22       :ordered-subtasks (and
23        (get-to ?v ?l1)
24        (load ?v ?l1 ?p)
25        (get-to ?v ?l2)
26        (unload ?v ?l2 ?p))
27    )
28    (:method m-unload
29      :parameters (?v - vehicle ?l - location ?p - package ?s1 ?s2 -
      capacity-number)
30      :task (unload ?v ?l ?p)
31      :subtasks (drop ?v ?l ?p ?s1 ?s2)
32    )
33
34    (:action drop
35      :parameters (?v - vehicle ?l - location ?p - package ?s1 ?s2 -
      capacity-number)
36      :precondition (and
37          (at ?v ?l)
38          (in ?p ?v)
39          (capacity-predecessor ?s1 ?s2)
40          (capacity ?v ?s1)
41        )
42      :effect (and
43          (not (in ?p ?v))
44          (at ?p ?l)
45          (capacity ?v ?s2)
46          (not (capacity ?v ?s1))
47        )
48    )
49 )
```

Listing 4.10: Example of HDDL syntax as used in the PANDA framework.

4.6.5.4 HPDDL

pourquoi en majuscule?

A very recent language proposition was done by RAMOUL (2018). He proposes HPDDL with a simple syntax similar to the one of UCMP. In **??** 4.11 we give an example of HPDDL method. Its expressive power seems similar to that of UCMP and SHOP.

```
1 (:method do_navigate
2   :parameters(?x - rover ?from ?to - waypoint)
3   :expansion((tag t1 (navigate ?x ?from ?mid))
4              (tag t2 (visit ?mid))
5              (tag t3 (do_navigate ?x ?mid ?to))
6              (tag t4 (unvisited ?mid)))
7   :constraints((before (and (not (can_traverse ?x ?from ?to)) (not
      (visited ?mid))
8                             (can_traverse ?x ?from ?mid)) t1)))
```

Listing 4.11: Example of HPDDL syntax as described by Ramoul.

4.6.6 Ontological

Another idea is to merge automated planning and other artificial intelligence fields with knowledge representation and more specifically ontologies. Indeed, since the "semantic web" is already widespread for service description, why not make planning compatible with it to ease service composition ?

TODO: cite Eva

4.6.6.1 WebPDDL

This question finds it first answer in 2002 with WebPDDL. This language, explicited in **??** 4.12, is meant to be compatible with RDF by using URI identifiers for domains (McDermott and Dou 2002). The syntax is inspired by PDDL, but axioms are added as constraints on the knowledge domain. Actions also have a return value and can have variables that aren't dependant on their parameters. This allows for greater expressivity than regular PDDL, but can be partially emulated using PDDL 3.1 constraints and object fluents.

```
(define (domain www-agents)
  (:extends (uri "http://www.yale.edu/domains/knowing")
            (uri "http://www.yale.edu/domains/regression-planning")
            (uri "http://www.yale.edu/domains/commerce"))
  (:requirements :existential-preconditions :conditional-effects)
  (:types Message - Obj Message-id - String)
  (:functions  (price-quote ?m - Money)
               (query-in-stock ?pid - Product-id)
               (reply-in-stock ?b - Boolean) - Message)
  (:predicates (web-agent ?x - Agent)
               (reply-pending a - Agent id - Message-id msg - Message)
               (message-exchange ?interlocutor - Agent
                                 ?sent ?received - Message
                                 ?eff - Prop)
               (expected-reply a - Agent sent expect-back - Message))
  (:axiom
      :vars (?agt - Agent ?msg-id - Message-id ?sent ?reply - Message)
      :implies (normal-step-value (receive ?agt ?msg-id) ?reply)
      :context (and (web-agent ?agt)
                    (reply-pending ?agt ?msg-id ?sent)
                    (expected-reply ?agt ?sent ?reply)))
  (:action send
      :parameters (?agt - Agent ?sent - Message)
      :value (?sid - Message-id)
      :precondition (web-agent ?agt)
      :effect (reply-pending ?agt ?sid ?sent))
  (:action receive
    :parameters (?agt - Agent ?sid - Message-id)
    :vars (?sent - Message ?eff - Prop)
    :precondition (and (web-agent ?agt) (reply-pending ?agt ?sid ?sent))
    :value (?received - Message)
    :effect (when (message-exchange ?agt ?sent ?received ?eff) ?eff)))
```

Listing 4.12: Example of WebPDDL syntax by Mc Dermott.

4.6.6.2 OPT

This previous work was updated by McDermott (2005). The new version is called OPT and allows for some further expressivity. It can express hierarchical domains with links between actions and even advanced data structure. The syntax is mostly an update of WebPDDL. In ?? 4.13, we can see that the URI was replaced by simpler names, the action notation was simplified to make the parameter and return value more natural. Axioms were replaced by facts with a different notation.

préciser numéro de ligne

```
1 (define (domain www-agents)
2   (:extends knowing regression-planning commerce)
3   (:requirements :existential-preconditions :conditional-effects)
4   (:types Message - Obj Message-id - String )
5   (:type-fun (Key t) (Feature-type (keytype t)))
6   (:type-fun (Key-pair t) (Tup (Key t) t))
7   (:functions (price-quote ?m - Money)
8               (query-in-stock ?pid - Product-id)
9               (reply-in-stock ?b - Boolean) - Message)
10  (:predicates (web-agent ?x - Agent)
11               (reply-pending a - Agent id - Message-id msg - Message)
12               (message-exchange ?interlocutor - Agent
13                                 ?sent ?received - Message
14                                 ?eff - Prop)
15               (expected-reply a - Agent sent expect-back - Message))
16  (:facts
17    (freevars (?agt - Agent ?msg-id - Message-id
18               ?sent ?reply - Message)
19      (<- (and (web-agent ?agt)<!-- -->
20               (reply-pending ?agt ?msg-id ?sent)
21               (expected-reply ?agt ?sent ?reply))
22               (normal-value (receive ?agt ?msg-id) ?reply))))
23  (:action (send ?agt - Agent ?sent - Message) - (?sid - Message-id)
24    :precondition (web-agent ?agt)
25    :effect (reply-pending ?agt ?sid ?sent))
26  (:action (receive ?agt - Agent ?sid - Message-id) - (?received -
      Message)
27    :vars (?sent - Message ?eff - Prop)
28    :precondition (and (web-agent ?agt)
29                       (reply-pending ?agt ?sid ?sent))
30    :effect (when (message-exchange ?agt ?sent ?received ?eff) ?eff)))
```

Listing 4.13: Example of the updated OPT syntax as described by Mc Dermott.

4.7 Color and general planning representation

From the general formalism of planning proposed earlier, it is possible to create an instanciation of the SELF language for expressing planning domains. This extension was the primary goal of creating SELF and uses almost all features of the language.

4.7.1 Framework

In order to describe this planning framework into SELF, we simply put all fields of the actions into properties. Entities are used as fluents, and the entire knwoledge domain as constraints. We use parameterized types as specified **BEFORE**.mettre ref

```
1 "lang.w" = ? ; //include default language file.
2 Fluent = Entity;
3 State = (Group(Fluent), Statement);
4 BooleanOperator = (&,|);
5 (pre,eff, constr)::Property(Action,State);
6 (costs,lasts,probability) ::Property(Action,Float);
7 Plan = Group(Statement);
8 -> ::Property(Action,Action); //Causal links
9 methods ::Property(Action,Plan);
```

Listing 4.14: Content of the file "planning.w"

The file presented in **??** 4.14, gives the definition of the syntax of fluents and actions in SELF. The first line includes the default syntax file using the first statement syntax. The fluents are simply typed as entities. This allows them to be either parameterized entities or statements. States are either a set of fluents or a logical statement between states or fluents. When a state is represented as a set, it represents the conjunction of all fluents in the set.

on est dans la section 4.7.1

Then at section 4.7.1, we define the preconditions, effects and constraint formalism. They are represented as simple properties between actions and states. This allows for the simple expression of commonly expressed formalism like the ones found in PDDL. section 4.7.1 expresses the other attributes of actions like its cost, duration and prior probability of success.

Plans are needed to be represented in the files, especially for case based and hierarchical paradigms. They are expressed using statements for causal link representation. The property -> is used in these statements and the causes are either given explicitly as parameters of the property or they can be inferred by the planner. We add a last property to express methods relative to their actions.

4.7.2 Example domain

repréciser où, le earlier c'est 26 pages de lecture ce n'est pas au lecteur de rechercher de quoi tu parles….

Using the classical example domain used earlier, we can write the following file in **??** 4.15.

```
1 "planning.w" = ? ; //include base terminology
2
3 (! on !, held(!), down(_)) :: Fluent;
4
5 pickUp(x) pre (~ on x, down(x), held(~));
6 pickUp(x) eff (~(down(x)), held(~));
7
8 putDown(x) pre (held(x));
9 putDown(x) eff (held(~), down(x));
10
11 stack(x, y) pre (held(x), ~ on y);
12 stack(x, y) eff (held(~), x on y);
```

```
13
14 unstack(x, y) pre (held(~), x on y);
15 unstack(x, y) eff (held(x), ~ on y);
```

Listing 4.15: Blockworld written in SELF to work with Color

At line section 4.7.2, We need to include the file defined in ?? 4.14. After that section 4.7.2 defines the allowed arrity of each relation/function used by fluents. This restricts eventually the cardinality between parameters (one to many, many to one, etc).

section 4.7.2 encodes the action $pickup$ defined earlier. It is interesting to note that instead of using a constant to denote the absence of block, we can use an anonymous exclusive quantifier to make sure no block is held. This is quite useful to make concise domains that stay expressive and intuitive.

c'est bien de mettre en avant l'intérêt de COLOR vs PDDL, mais l'illustrer avec un exemple serait mieux (remarque aussi de samir).

4.7.3 Differences with PDDL

SELF+Color is more consice than PDDL. It will infere most types and declaration. Variables are also inferred if they are used more than once in a statement and also part of parameters.

While PDDL uses a fixed set of extensions to specify the capabilities of the domain, SELF uses inclusion of other files to allow for greater flexibility. In PDDL, everything must be declared while in SELF, type inference allows for usage without definition. It is interesting to note that the use of variables names `x` and `y` are arbitrary and can be changed for each statement and the domain will still be functionally the same. The line 3 in ?? 4.2 is a specific feature of SELF that is absent in PDDL. It is possible to specify constraints on the cardinality of properties. This limits the number of different combinations of values that can be true at once. This is typically done in PDDL using several predicate or constraints.

Most of the differences can be sumarized saying that 'SELF do it once, PDDL needs it twice'. This doesn't only mean that SELF is more compact but also that the expressivity allows for a drastic reduction of the search space if taken into account. Thiébaux *et al.* (2005) advocate for the recognition of the fact that expressivity isn't just a convenience but is crucial for some problems and that treating it like an obstacle by trying to compile it away only makes the problem worse. If a planner is agnostic to the domain and problem, it cannot take advantages of clues that the instanciation of an action or even its name can hold (Babli *et al.* 2015).

Whatever the time and work that an expert spends on a planning domain it will always be incomplete and fixed. SELF allows for dynamical extension and even adresses the use of reified actions as parameters. Such a framework can be useful in multi-agent systems where agents can communicate composite actions to instruct another agent. It can also be useful for macro-action learning that allows to improve hierarchical domains from repeating observations. It can also be used in online planning to repair a plan that failed. And at last this framework can be used for explanation or inference by making easy to map two similar domains together. (lots of **CITATION**).

Also another difference between SELF and PDDL is the underlying planning framework. We presented the one of SELF (?? 4.14) but PDDL seems to suppose a more classical state based formalism. For example the fluents are of two kinds depending if they are used as preconditions or effects. In the first case, the fluent is a formula that is evaluated like a predicate to know if the action can be executed in any given state. Effects are formula enforcing the values of existing fluent in the state. SELF just suppose that the new knowledge is enforcing and that the fluents are of the same kind since verification about the coherence of the actions are made prior to its application in planning.

ajouter exemple puis conclusion générale du chapitre