$$\mathbb{\Pi}_{\mathcal{C}} = \mathbb{\Pi}^* (r_*, \{\text{min} : \mathbb{m} \in \mathcal{C} \wedge \mathbb{m}(pre(\omega)) \vDash eff(\omega)\}, \mathbb{s}(pre(\omega)) \neq \varnothing)$$

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).

4.5.4 Probabilistic planning

Probabilistic planning tries to deal with uncertainty by working on 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 states as search space with the policy as the iterator.

$$\mathbb{\Pi}_{\mathbb{p}} = \mathbb{\Pi}^* (\mathbb{pol}, pre(\omega), \mathbb{s} \vDash eff(\omega))$$

At each iteration a state is chosen from the frontier. The frontier is updated with the application of the most likely to succeed action given by the policy. The search stops when the frontier satisfies the goal.

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 $\mathbb{\Pi}$. We note r_+ the set of classical refinements in a plan along with any action decomposition (see chapter 6).

$$\mathbb{\Pi}_{\omega} = \mathbb{\Pi}^* (r_+, \mathbb{\Pi}(\omega), \bigotimes(\mathbb{s}) = \varnothing \wedge \forall a \in A_{\mathbb{m} \in \mathbb{s}} \mathbb{\Pi}(a) = \varnothing)$$

The instance of the general planner for HTN planning is similar to the PSP one: it fixes flaws in plans. The idea is to add a *decomposition flaw* to the classical flaws of PSP. This technique is more detailed in chapter 6.

4.6 Discussion on General Planning

In this chapter, we presented a way to formalize all planning paradigms under a unifying notation. This is quite interesting in the fact that it is now easier to explain planners that uses one or more paradigms at once: the so called "hybrid planners". That last notion is far from new as demonstrated by Gerevini *et al.* (2008) and in the field of HTN with task insertion (HTN-TI) which reuses PSP like techniques.

In the following two chapters, we will show how using SELF with this formalism allows for a general planning framework and how it can be used to make hybrid planners.

5 COLOR Framework

Using the formalism for a general planner, it becomes possible to define a general *action language*. Action languages are language used to encode planning domains and problems. Among the first to emerge, we can find the popular STRIPS. It is derived from its eponymous planner Stanford Research Institute Problem Solver (Fikes and Nilsson 1971).

After STRIPS, one of the first languages to be introduced to express planning domains is like Action Description Language (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 by changING in effects and to change both preconditions and effects of actions into logic formula instead of simple states.

manque intro du chapitre: ce que tu vas présenter !

5.1 PDDL

The most popular standard action language in automated planning is the Planning Domain Description Languaghe (PDDL). It was created for the first major automated planning competition hosted by AIPS in 1998 (Ghallab *et al.* 1998). this language came This along with syntax and solution checker written in Lisp. The goal was 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 listing 5.1.

```
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 5.1: 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 42. For example, consider the classic block world domain expressed in listing 5.2. 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.

```
 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 5.2: Classical PDDL 3.0 definition of the domain Block world

aN

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 5.1. Most requirements are parts of PDDL that either increase the complexity of planning significantly or that require extra implementation effort to meet. For example, the `quantified-precondition` adds quantifiers into the logical formula of precondition forcing a check on all fluents of the state to check the validity

manque un point

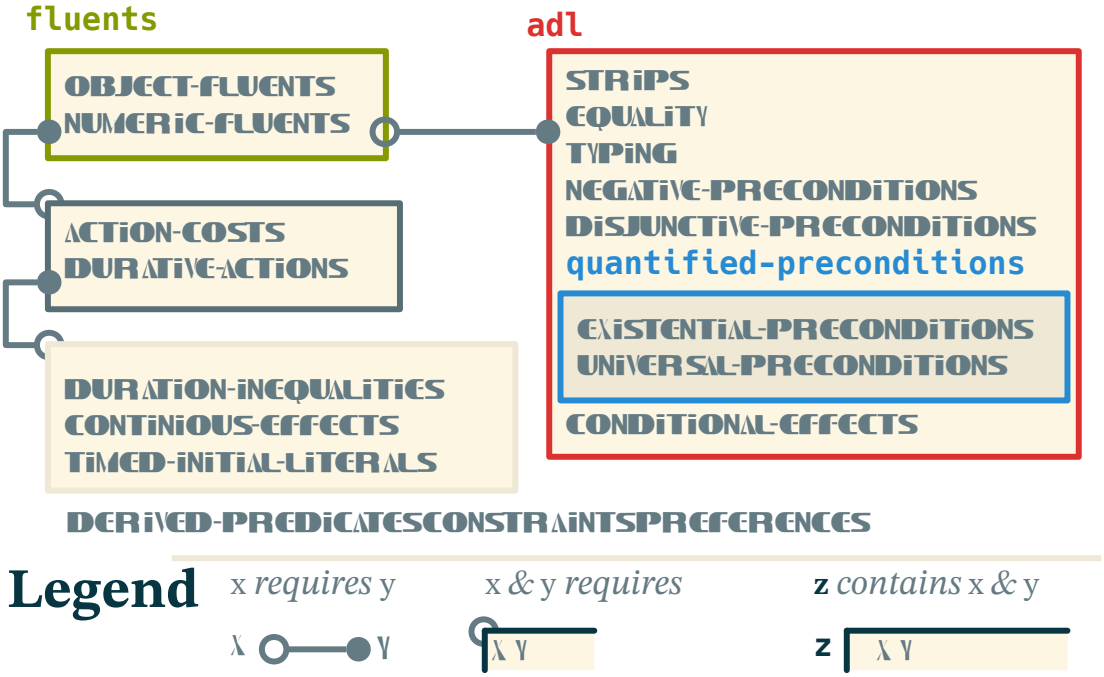

Figure 5.1: Dependencies and grouping of PDDL requirements.

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 **??**.

Table 5.1: Planners participating in the Classic track of the 2018 International Planning Competition (IPC). Most of the planners uses the FastDownward framework. {#tbl:ipc}

Planner	Translated	Language	Framework	Rank
Delfi	Yes	C++	FD	1
Complementary	Yes	C++	FD	2
Planning-PDBs	Yes	C++	FD	3
Scorpion	Yes	C++	FD	4
FDMS	Yes	C++	FD	5
DecStar	Yes	C++	LAMA	6
Metis	Yes	C++	FD	7
MSP	Yes	Lisp	FD	8
Symple	Yes	C++	FD	9
Ma-plan	No	C	None	10

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

5.2 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.
dentLY

5.2.1 PDDL+

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 listing 5.3, 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+.

```
(:durative-action downlink
    :parameters (?r - recorder ?g - groundStation)
    :duration (> ?duration 0)
    :condition (and (at start (inView ?g))
                    (over all (inView ?g))
                    (over all (> (data ?r) 0)))
    :effect (and (increase (downlinked)
                      (* #t (transmissionRate ?g)))
                 (decrease (data ?r)
                      (* #t (transmissionRate ?g)))))
```

Listing 5.3: Example of PDDL+ durative action from Fox's paper.

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.

5.3 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.

5.3.1 PPDDL

PPDDL is such a language. It was used during the 4^{th} and 5^{th} IPC for its probabilistic track (Younes and Littman 2004). It allows for probabilistic effects as demonstrated in listing 5.4. 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 5.4: Example of PPDDL use of probabilistic effects from Younes's paper.

5.3.2 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 listing 5.5 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;
```

```
43  discount = 0.9;
44 }
```

Listing 5.5: Example of RDDL syntax by Sanner.

5.4 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.

ariseS
concurRently

5.4.1 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. Listing 5.6 is an example of the syntax of MAPL domains. PDDL 3.0 seems to share a similar syntax.

S
comMunication

```
1 (:state-variables
2   (pos ?a - agent) - location
3   (connection ?p1 ?p2 - place) - road
4   (clear ?r - road) - boolean)
5 (:durative-action Move
6   :parameters (?a - agent ?dst - place)
7   :duration (:= ?duration (interval 2 4))
8   :condition
9     (at start (clear (connection (pos ?a) ?dst)))
10  :effect (and
11    (at start (:= (pos ?a) (connection (pos ?a) ?dst)))
12    (at end (:= (pos ?a) ?dst))))
```

Listing 5.6: Example of MAPL syntax by Brenner.

5.4.2 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 listing 5.7, 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

détailler ce n'est pas au lecteur de devoir trouver tout seul les info

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

Listing 5.7: Example of MA-PDDL syntax by Kovacs.

5.5 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 an executable plan. generateS

5.5.1 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 listing 5.8. 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.

```
(constants a b c table) ; declare constant symbols
(predicates on clear) ; declare predicate symbols
(compound-tasks move) ; declare compound task symbols
(primitive-tasks unstack dostack restack) ; declare primitive task symbols
(variables x y z) ; declare variable symbols

(operator unstack(x y)
          :pre ((clear x)(on x y))
          :post ((~on x y)(on x table)(clear y)))
(operator dostack (x y)
          :pre ((clear x)(on x table)(clear y))
          :post ((~on x table)(on x y)(~clear y)))
(operator restack (x y z)
          :pre ((clear x)(on x y)(clear z))
          :post ((~on x y)(~clear z)(clear y)(on x z)))

(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 5.8: Example of the syntax used by UCMP.

5.5.2 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 listing 5.9.

```
1 (:method
2   ; head
3     (transport-person ?p ?c2)
4   ; precondition
5     (and
6       (at ?p ?c1)
7       (aircraft ?a)
8       (at ?a ?c3)
9       (different ?c1 ?c3))
10  ; subtasks
11    (:ordered
12      (move-aircraft ?a ?c1)
13      (board ?p ?a ?c1)
14      (move-aircraft ?a ?c2)
15      (debark ?p ?a ?c2)))
```

Listing 5.9: Example of method in the SHOP2 language.

5.5.3 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 listing 5.10. That language was called HDDL.

```
(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)
    :task (deliver ?p ?l2)
     :ordered-subtasks (and
      (get-to ?v ?l1)
      (load ?v ?l1 ?p)
      (get-to ?v ?l2)
      (unload ?v ?l2 ?p))
  )
  (:method m-unload
    :parameters (?v - vehicle ?l - location ?p - package ?s1 ?s2 -
    capacity-number)
    :task (unload ?v ?l ?p)
    :subtasks (drop ?v ?l ?p ?s1 ?s2)
  )

  (:action drop
    :parameters (?v - vehicle ?l - location ?p - package ?s1 ?s2 -
    capacity-number)
    :precondition (and
        (at ?v ?l)
        (in ?p ?v)
        (capacity-predecessor ?s1 ?s2)
        (capacity ?v ?s1)
      )
    :effect (and
        (not (in ?p ?v))
        (at ?p ?l)
        (capacity ?v ?s2)
        (not (capacity ?v ?s1))
      )
  )
)
```

Listing 5.10: Example of HDDL syntax as used in the PANDA framework.

5.5.4 HPDDL

A very recent language proposition was done by Ramoul (2018). He proposes HPDDL with a simple syntax similar to the one of UCMP. In listing 5.11 we give an example of HPDDL method. Its expressive power seems similar to that of UCMP and SHOP.

```
(:method do_navigate
  :parameters(?x - rover ?from ?to - waypoint)
  :expansion((tag t1 (navigate ?x ?from ?mid))
             (tag t2 (visit ?mid))
             (tag t3 (do_navigate ?x ?mid ?to))
             (tag t4 (unvisited ?mid)))
  :constraints((before (and (not (can_traverse ?x ?from ?to)) (not (visited
    ?mid))
                            (can_traverse ?x ?from ?mid)) t1)))
```

Listing 5.11: Example of HPDDL syntax as described by Ramoul.

5.6 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 ?

5.6.1 WebPDDL

This question finds it first answer in 2002 with WebPDDL. This language, explicited in listing 5.12, 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 5.12: Example of WebPDDL syntax by Mc Dermott.

5.6.2 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 listing 5.13, 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.

Où? donner numéro de lignes

```
(define (domain www-agents)
  (:extends knowing regression-planning commerce)
  (:requirements :existential-preconditions :conditional-effects)
  (:types Message - Obj Message-id - String )
  (:type-fun (Key t) (Feature-type (keytype t)))
  (:type-fun (Key-pair t) (Tup (Key t) t))
  (: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))
  (:facts
    (freevars (?agt - Agent ?msg-id - Message-id
               ?sent ?reply - Message)
      (<- (and (web-agent ?agt)<!-- -->
               (reply-pending ?agt ?msg-id ?sent)
               (expected-reply ?agt ?sent ?reply))
               (normal-value (receive ?agt ?msg-id) ?reply))))
  (:action (send ?agt - Agent ?sent - Message) - (?sid - Message-id)
    :precondition (web-agent ?agt)
    :effect (reply-pending ?agt ?sid ?sent))
  (:action (receive ?agt - Agent ?sid - Message-id) - (?received - Message)
    :vars (?sent - Message ?eff - Prop)
    :precondition (and (web-agent ?agt)
                       (reply-pending ?agt ?sid ?sent))
```

```
30     :effect (when (message-exchange ?agt ?sent ?received ?eff) ?eff)))
```

Listing 5.13: Example of the updated OPT syntax as described by Mc Dermott.

5.7 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.

5.7.1 Framework

encore des BEFORE ?????????

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 knOW as constraints. We use parameterized types as specified **BEFORE**.

```
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 5.14: Content of the file "planning.w"

The file presented in listing 5.14, 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.

Then at line 5, 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. Line 6 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.

5.7.2 Example domain

Using the classical example domain used earlier, we can write the following file in listing 5.15.

```
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 5.15: Blockworld written in SELF to work with Color

we
arity

At ~~line~~ line 1, We need to include the file defined in listing 5.14. After that line 3 defines the allowed ar~~r~~ity of each relation/function used by fluents. This restricts eventually the cardinality between parameters (one to many, many to one, etc).

Line 5 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.

5.7.3 Differences with PDDL

Nommer autrement

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
? 5.15 line 3 in listing 5.2 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.

suMMarized 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.

Also another difference between SELF and PDDL is the underlying planning framework. We presented the one of SELF (listing 5.14) 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.

supposeS

5.8 Conclusion

In this chapter we explain how a general planning framework can be designed to interpret any planning paradigm. We explained how classical action languages encode their domain representation and specific features. After illustrating each language with an example, we propose our framework based on SELF and compare it to the standard currently in use.