A PDDL-compliant Kotlin library for automated planning

(v. 0.1.0)

Author
giulia.brugnatti@studio.unibo.it

September 5, 2022

Planning is a leading area of *Artificial Intelligent* that has made relevant advancements in terms of efficiency and sophistication of its algorithms since its emergence in the late 1970s. However, despite numerous planners available, their preponderance is no longer maintained. Whence we decided to create our planning framework based on *PDDL*.

Thus, the project aims to realize a planning framework supporting PDDL and implementing STRIPS as a planning algorithm.

Since PDDL syntax is not often intelligible, the planner is delivered with a DSL to increase its usability for lay users.

1 Goals/requirements

The project's primary objective is the realization of a planning framework that supports PDDL and implements STRIPS as a planning algorithm; the framework should at least provide the features of PDDL [10] proposed in its `1.2` version. The planning framework is provided with a *DSL* to boost its level of usability.

1.1 Scenarios

The framework aims to be a versatile tool to be used both to compute straightforward plans in a simple scenario as a could be a *block world* or to be employed in more complex applications. Thus, the planner's goal is to be a modular and extensible tool that users can utilize in different situations depending on their needs.

1.2 Self-assessment policy

As a self-assessment policy, we will evaluate the code quality by analyzing three main dimensions; the first one concerns its internal organization taking as first level requirements its readability, modularity, and reusability, while the second focuses on being compliant

PLANNER	LAST UPDATE
ENHSP[1]	2020
DiNo[2]	2019
COLIN[3]	2012
MetricFF[4]	2020
POPS[5]	2011
OPTIC[6]	only the executable file available
FF[7]	only the executable file available
TFD[8]	2014
LPG[9]	2004
...	...

Table 1: Planner comparison [12]

with the standards and best practices of the language chosen for the implementation. Finally, the latest is about the tests and their coverage.

On the other hand, we will measure the project's effectiveness by examining its compliance with requirements (functional, non-functional and implementing), whose details we will discuss in the following sections.

2 Background

The planning problem is a relevant area of AI who made significant advancements in terms of its algorithms' performances and elaborateness since its injection in the late 1970s.

In general, the planning problem in AI concerns the decisions undertaken by an intelligent agent (i.e. robots, humans, or software) when seeking to reach some goal. It usually reckons to seek a sequence of actions that will turn the initial state of the world into the target one.

Nevertheless, despite the considerable amount of planners available, a brief survey (whose results are in table 1) revealed that most of them are no longer maintained. Whence we decided to implement our planner based on PDDL 1.2.

2.1 Planning

Planning Problem Planning is a term that means different things to different groups of people [16]; commonly, if the planner is a machine, it is considered a planning algorithm. In general, a planning problem has some common shared ingredients:

State: planning problems usually concern a *state space* that seizes all possible situations that could occur. The state space might be both discrete and continuous [16].

Time: planning problems involve a sequence of decisions that the planner apply over time. We can model time explicitly or implicitly by reflecting the fact that actions must follow in succession and, as in the case of state spaces, time may be either discrete or continuous [16].

Actions: a planner relies on a formal description of the executable actions that is called Domain Theory. The application of the action to a state possibly cause a change within it; thus, it is necessary to specify somewhere in the planning formulation, how the state changes after actions' execution [16]. Usually, that is described within the action, specifying its preconditions and post-conditions.

- **Preconditions:** are the conditions which must hold in the current state to ensure the action's execution.
- **Post-conditions:** are conditions that must be satisfied in the state after the application of the action; thus, they represent the effects of the action on the world.

Initial and goal states: a planning problem usually interests an initial state and from which the planner tries to arrive at a specified goal state.

Plan: is a partially or completely ordered set of actions needed to achieve the goal from the initial state

[9]**ENHSP**: Expressive Numeric Heuristic Search Planner is described at: `https://planning.wiki/ref/planners/enhsp`.

[9]**DiNo**: Discrete Non-linear Heuristic Planner's description can be retrieved at: `https://planning.wiki/ref/planners/dino`.

[9]**MetricFF**: one can find information about the planner at: `https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html`.

[9]**POPS**: Partial Order Planning Forwards' documentation is at: `https://planning.wiki/ref/planners/popf`.

[9]**OPTIC**: Optimising Preferences and Time-Dependent Costs' description can be retrieved at: `https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html`.

[9]**FF**: The Fast Forward Planning System is described at: `https://planning.wiki/ref/planners/ff#metricff`.

[9]**TFD**: one can find more information about *Temporal Fast Downward* at: `http://gki.informatik.uni-freiburg.de/tools/tfd/`

[9]**LPG**: is a a *Planner Based on Local Search for Planning Graphs with Action Costs* whose documentation is at: `https://lpg.unibs.it/lpg/`.

A criterion : encodes the desired outcome of a plan in terms of the state and actions performed. There are generally two different kinds of planning concerns based on the type of criterion:

- **feasibility**: find a plan that causes arrival at a goal state, regardless of its efficiency.
- **optimality**: is about a feasible plan that optimizes performance in some way while reaching the goal state.

Thus, given:

- a state of the world,
- a set of actions authorised,
- a set of objects allowed in the world,
- a goal to achieve,

a planner devises a plan, i.e. a sequence of actions, that leads from the initial state to goal one.

Although planners are not always required, they are relevant in many application scenarios. In particular, we might need planners in those cases where systems need to be observable, accountable, and explainable; thus, whenever reasons behind actions need to be a priori known for any system with some responsibility (i.e. socio-technical systems) [19].

STRIPS: the *Standford Research Institute Problem Solver*(STRIPS) is a well-known algorithm for automatic planning devised in 1971 by *Richard Fikes* and *Nils Nilsson* at *Stanford Research Institute International* (SRI International)[10] [31].

STRIPS is a linear non-hierarchical planner based on backward search exploiting the *Closed World Assumption* (CWA)[11], that operates by running a problem and a domain to achieve a goal. In STRIPS, firstly the world is described by endowing the domain world of all the entities types one can find within: *objects*, *actions*, *preconditions*, and *effects*. Subsequently, one delineates the problem by defining an initial and a goal state. Eventually, when the algorithm is triggered, STRIPS compute the search of the state space, starting from the initial one, performing multiple actions until it achieves the goal, if it is possible.

[10] SRI International is an American nonprofit scientific research institute and organization headquartered in *Menlo Park*, California established in 1946 [30].

[11] The CWA is a presumption employed by formal systems of logic used for knowledge representation, it states that a statement that is true is also known to be true conversely, what is not currently known to be true, is false [27].

Algorithm: many formalizations of the STRIPS algorithm are available, in this document we use the one propose in [6]. Before examining the STRIPS algorithm some preliminary descriptions are required. Thus let:

- $[Head \mid Tail]$ denote a stack whose head is $Head$ and where other elements are represented by $Tail$;
- $Actions$ be a set of action names, ranged by N;
- $State$, $State_0$ states, i.e. sets of fluents, ranged by F;
- $pre(N) \mapsto G_1 \wedge \ldots \wedge G_n$ the function mapping each action name N to the conjunction of preconditions of N;
- $post_+(N) \mapsto \{G_1, \ldots, G_n\}$ the function mapping each action name N to the set of positive effects of N; item $apply(State, N) \mapsto State'$ be the function mapping a state-action-name couple to the state attained by applying that action to that state;

The pseudo-code implementation of the algorithm is in algorithm 1.

noend 1 Pseudo-code of the STRIPS algorithm.

function STRIPS($State_0$, $Actions$, $Goal \equiv G_1 \wedge \ldots \wedge G_n$)
  $State \leftarrow State_0$; $Stack \leftarrow [G_1 \wedge \ldots \wedge G_n]$; $Plan \leftarrow []$

  while $Stack \equiv [H \mid Tail]$ **do** ▷ while $Stack$ is not empty
    if $\exists F \in State : mgu(H, F) = \theta$ **then**
      $Stack \leftarrow Tail/\theta$ ▷ Pop
    else if $\exists N \in Actions : \exists E \in post_+(N) : mgu(H, E) = \theta$ **then** ▷ CP
      $Stack \leftarrow [pre(N), N \mid Tail]/\theta$ ▷ Pop; Push(N); Push($pre(N)$)
    else if $H \equiv G_1 \wedge \ldots \wedge G_m$ **then**
      $Stack \leftarrow [G_1, \ldots, G_m \mid Tail]$ ▷ Pop, Push(G_1), ..., Push(G_m), CP
    else if $H \in Actions$ **then**
      $State \leftarrow apply(State, H)$
      $Plan \leftarrow [H \mid Plan]$
    else
      return ▷ Failure

  return reverse($Plan$) ▷ Success

Because of its non-deterministic choice in the ordering and the possibility of applying multiple actions to reduce a goal, STRIPS can often result in inefficiency. For that reason, the search strategy applied plays a crucial role, as we will examine in section 5.1.

We choose to support STRIPS as a planner algorithm for many reasons: the first one is because this algorithm has a long history in the planning domain; thus, even if its design date back to the late 1970s, it is still used in many lectures as the first example

of planner for simplicity. Indeed, STRIPS is computationally simple because it rests on extensional knowledge where the problem of testing the satisfiability of a formula is reduced to control if a fluent is in a set of fluents that made up the goal (as will be later analyzed in section 4.1.1).

PDDL: the *Planning Domain Definition Language* (PDDL) is a formal knowledge representation language designed to express planning models [14]. It was first developed in 1998 by *Drew McDermott* as a means of facilitating systems comparison; it has become the language of the *International Planning Competition* (ICAPS[12]) and a de-facto standard input language of many planning systems but it is not the only modelling language for planning. Several variants of PDDL have emerged to capture different aspects of the planning problem at the increasing level of complexities, with a particular focus on deterministic problems [14].

Thus, PDDL is a domain-independent[13] planner who includes a different set of features depending on its version. The characteristics spreads from the most simple as type specification for *objects*, *negated preconditions*, *conditional add/del effects*, that can been found in its first version, to the more advanced as the *numeric variables* and *durative actions*. Furthermore, it includes STanford Research Institute Problem Solver (STRIPS) as a notable case of planner available.

As planning has evolved, so has the language used to describe it; thus, as shown in table 2, now many versions of PDDL are available with different levels of expressivity [13].

We choose to support PDDL instead of other planning languages because, since its adoption by ICAPS, it has become a de-facto standard in the planning system domain.

A running example of PDDL usage to describe a domain for a simple gripper problem is proposed in 1 [1].

2.2 2P-kt

2P-kt is a multi-platform, Kotlin-based reboot of *tuProlog*. It is an open ecosystem for Symbolic Artificial Intelligence (AI). 2P-kt has a multi-module architecture with several incrementally inter-dependent modules which aim to support symbolic manipulation and reasoning in an extensible and flexible way.

We choose 2P-kt because it is far more than yet another Prolog interpreter; indeed, it is an ecosystem for logic programming(LP) and symbolic AI; where LP functionalities are made available as re-usable and extensible libraries. Thus, a Prolog interpreter is available, but not only [5].

[12]Further information about ICAPS are at `https://www.icaps-conference.org/competitions/`

[13]The core philosophy built into PDDL is that the techniques applied to solving a problem should know nothing about the problem they are solving.

Listing 1: Snippet showing a minimal example of a domain definition in PPDL.

```
(define (domain letseat)

  (:requirements :typing)

  (:types
    location locatable - object
    bot cupcake - locatable
    robot - bot
  )

  (:predicates
    (on ?obj - locatable ?loc - location)
    (holding ?arm - locatable ?cupcake - locatable)
    (arm-empty)
    (path ?location1 - location ?location2 - location)
  )


  (:action pick-up
    :parameters
     (?arm - bot
      ?cupcake - locatable
      ?loc - location)
    :precondition
     (and
        (on ?arm ?loc)
        (on ?cupcake ?loc)
        (arm-empty)
      )
    :effect
     (and
        (not (on ?cupcake ?loc))
        (holding ?arm ?cupcake)
        (not (arm-empty))
     )
  )

  (:action drop
    :parameters
     (?arm - bot
      ?cupcake - locatable
      ?loc - location)
    :precondition
     (and
        (on ?arm ?loc)
        (holding ?arm ?cupcake)
      )
    :effect
     (and
        (on ?cupcake ?loc)
        (arm-empty)
        (not (holding ?arm ?cupcake))
     )
  )

  (:action move
    :parameters
     (?arm - bot
      ?from - location
      ?to - location)
    :precondition
     (and
      (on ?arm ?from)
      (path ?from ?to)
     )
    :effect
     (and
      (not (on ?arm ?from))
      (on ?arm ?to)
     )
  )
)
```

Version	Features
PDDL 1.2	- predicate centric (i.e. classical representation) - object types; - ADL features (e.g. conditional effects, equality);
PDDL 2.1	- numeric fluents; - durative actions;
PDDL 2.2	- timed-initial literals; - derived predicates;
PDDL 3.0	- state-trajectory constraints (hard constraints for the planning process); - preferences (soft constraints for the planning process);
PDDL 3.1	- object fluents;
PDDL+	- continuous processes; - exogenous events;
PPDDL	- probabilistic action effects; - reward fluents;
MA-PDDL	- multi-agent planning;

Table 2: PDDL versions with the major features introduced in each of them [3].

3 Requirements Analysis

3.1 Functional requirements

Given an initial state of the world and a goal position within it, the planner must be able to synthesize all the possible sequences of actions (plans) that lead from the initial state to the target state (goal), if they exist.

It should let users describe problems through the use of predicates and actions defining:

- a set of objects existing in the world which have predicate proprieties[14] to specify what is true about them;
- a set of actions to perform, conditions that must hold;
- a set of states for the system;
- an end goal to be reached at the end of the computation.

3.2 Non functional requirements

For the planner's implementation, the software engineering's principles of modularity and decomposition are pursued, along with the ones of extensibility and inspectability. Furthermore, other negligible requirements are the code's maintainability and the level of usability of that software produced.

[14]Predicate properties: properties that are either false or true.

3.3 Implementation requirements

The implementation requirements are summarized as follows:

- **Programming language**: the target language for the planner and its DSL implementation is *Kotlin*; thus, we will follow the language best practices during the implementation;
- **Test**: given the choice of Kotlin as the target platform for the implementation, the framework chosen to implement the tests is *Kotest*[15];
- **Version Control**: we will use *GitHub* as code hosting platform for version control and collaboration;
- **Continous Integration**: a *Github action* will check the correctness of the code each time the code will be uploaded on a develop branch (and its sub-branches) and on a pull request on the master. Furthermore, the *Renovate* bot will automatically handle the dependency update[16].
- **Code examination**: to avoid bikeshedding on formatting, elude code smells, while implementing the Kotlin best practice standards we will use code analyzer tools: *Detekt*[17] and *Ktlint*[18].
- **Coverage**: we will check the unit tests' coverage analysis with the *IntelliJ* coverage tool, and it should be above the 85% for both classes and methods;
- **Documentation**: the code documentation will be automatically generated using a Kotlin documentation engine named *Dokka*[19].
- **Logic-engine**: as advanced in section 2, to avoid to re-invent the wheel, the *2P-kt* framework will be used behind the scenes to implement the core logic operations.

4 Design

4.1 Structure / Domain Entities

The project's structure is described by taking into account two dimensions: the planner, which contains the core entities of the domain problem, and the DLS designed to use it.

[15] Kotest official documentation is at `https://kotest.io/`.
[16] Renovate: an open-source tool which automatically creates *pull requests* for all types of dependency updates. Crowdsourced tests and package adoption data are used to flag potentially risky updates and enable auto-merging for those that meet user-defined conditions [23].
[17] One can find Detekt documentation at: `https://detekt.dev/`.
[18] Further details about Ktlint are at: `https://ktlint.github.io/`.
[19] Dokka official documentation is at `https://github.com/Kotlin/dokka`.

4.1.1 Planner

The central domain entities identified follow closely the ones proposed in PDDL 1.2; however, to improve their comprehension and avoid confusion when calling the `2P-Kt` methods to implement the core logic operations, we rename some of them in the final version.

PDDL specifies a set of necessary concepts to define a search problem and its domain: the initial state, the actions available in a given state, the goal, etcetera.

Those core elements are analyzed in the following paragraphs and summarised in fig. 1.

Value: is the representation of an element that can be exploited by a program; it is wrapper for two kind of entities: *variable* or *object*, who will be discussed in the following paragraphs.

- **Variable:** is an entity that wraps a logic variable; it is responsible for the parameterisation of an action. Variables that stand for terms of the problem instance; are instantiated to object from a specific problem instance when an action is grounded for application.
- **Objects:** are what is interesting in the world. Objects are *constants*; each object name must be unique and should be typed. If not typed, then they will typically take on the properties of the base type object [9].

Type: is an entity that defines an object's type, it allows to define parent to child relationship by allowing a *supertype* definition.

Fluent: is a ground logic fact which is *true*, or *false* at a given moment.

Predicates: are signatures for the fluents allowed in the problem the user wants to model.

Precondition and effect: the preconditions and effects of an action are conjunctions of literals. The **precondition** defines the circumstances in which one can execute an action, and the **effect**, on the other hand, establishes the result of the action's execution.

In general, we say that action a is applicable in state s if the preconditions are satisfied by s. When an action schema a contains variables, it may have multiple applicable instantiations [22].

Goal: represents the state of the world we want to end at, i.e. things that we want to be true eventually. The goal is similar to a precondition: conjunction of literals (positive or negated atomic sentences) that may contain variables. Given that goals are conjunction of goals, one can see them as predicates on a state; thus, a condition one can test as it returns a boolean result. Consequently, the satisfiability of the goal concerns the computation of the sub-set relationship among states, and the goal as the conjunction of fluents that compose the goal must be in a state to consider the goal as fulfilled. The problem is solved when we find a sequence of actions that finish in a state s that entails the goal [22].

Operand: represents a logic operand (or, and, not) that one could apply to fluents to compose an expression.

Expression: is an entity which wraps fluents, binary and unary expressions.

BinaryExpression: is a particular sort of Expression which considers expressions composed of two expressions and an operand.

UnaryExpression: is a sub-type of Expression which concerns an operand that one must apply to only one Expression.

Axiom: is a derived predicate which essentially takes its value as the result of evaluating a logical expression over other predicates [12]. Thus, axioms are analogous to a set of `if-then-else` rules used to manipulate the fluents among states; they are not dissimilar to *Prolog* rules.

Variable Assignment: is a general type for logic substitutions; a Variable Assignment is a map used to assign values.

State: planning problems involve a state space that captures all possible situations that could arise. Each state is a conjunction of *ground fluents* which one can manipulate by logical inference.

Initial state: the state of the world that we start in, i.e. things that are true before the computation commences. The initial state is a conjunction of ground atoms [22].

Actions: are ways of changing the state of the world, i.e. things that happen that change the facts. A plan generates actions that manipulate the state. In the planning formulation, one must specify how the state changes when actions are applied.

As with any system for action description, our framework needs to deal with the *frame problem*[20] defining what changes and what stays the same as the result of the action. Thus, analogously to PDDL, we decide to define the result of an action in terms of what changes; everything not altered is unmentioned [22].

Domain: represents the *universal* aspects of a problem, i.e. those aspects that do not alter depending on the scenario considered [12].

Problem: is one of the two major parts that form a planning problem apart from the planner. It expresses the global *worldly* aspects of a problem planned as which actions one can perform executed, along with the types of objects acceptable, the property which holds on them and the final goal of the computation [12].

[20] Frame problem: it is a logic issue which concerns the definition of a formulæ able to describe the effects of actions without having to write a large number of accompanying formulæ that explain the mundane, obvious non-effects of those actions [18].

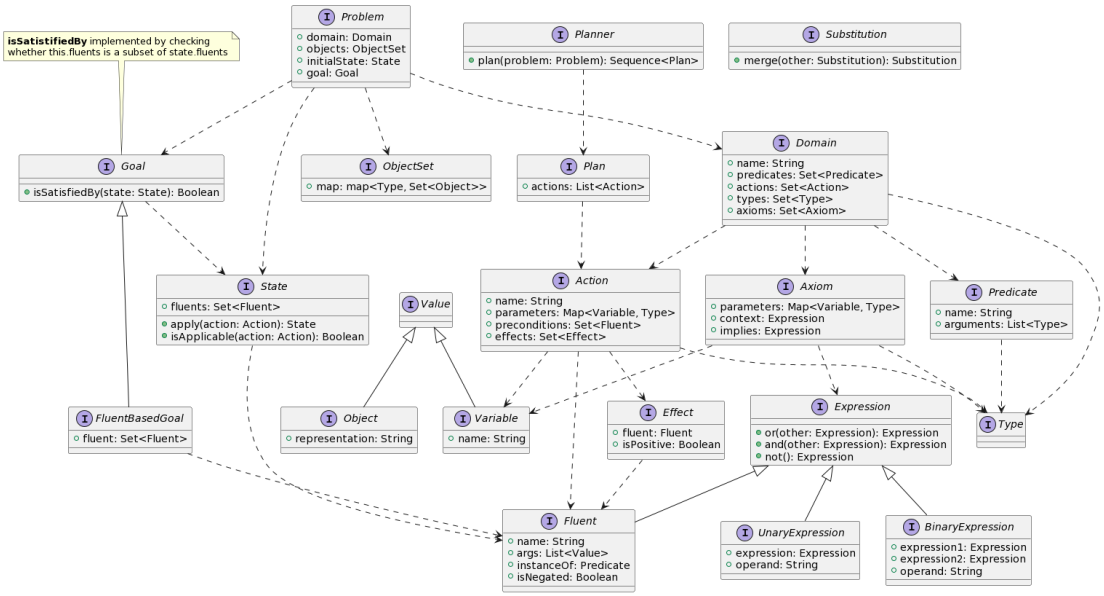

Figure 1: UML diagram including the core entities of the problem.

Plan: is a type for the result of the planning computation; if a planner can find at least one possible way to solve the problem it can elaborate a plan which includes the steps needed to solve the proposed problem. The resolution's stages involve the choice of the actions taken to move across the states of the scenario to reach the goal.

Planner: is the entity that encompasses the resolution strategy. The planner is the *another half of the proverbial planning equation* [12], it uses the elements examined in the previous paragraphs to wrap the PDDL entities defining the *Planning problem*, the planner seeks to solve it. If it does, it can therefore produce a plan.

4.1.2 DSL

Programming languages can be general-purpose (GPLs) or domain-specific (DSLs), depending on the distinctiveness of their constructs.

DSLs are languages specialized for a specific application or business context exploited to improve their understandability and reuse. To achieve the prior requirements, DSLs usually change how code is utilized and written, making it easier to learn and use [8].

We decide to design a DSL for our framework to enhance its usability and readability. To meet that goal, we implement a new entity for each of the core elements of the domain and the problem, enabling the user to use them more conveniently. The UML diagram of the DSL is shown in fig. 2

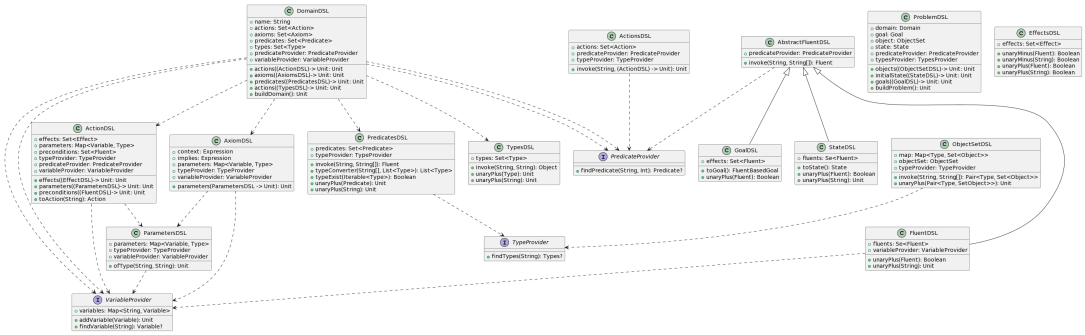

Figure 2: UML diagram including the core entities of the DSL.

4.2 Behaviour

A *Nondeterministic state Automaton* can describe the application's behaviour.

Nondeterminism is a pivotal concept in the theory of computing; it concerns the possibility of having several options for what can occur at numerous stages of the elaboration process. Next, we then examine the available results that those options can have, commonly concentrating on whether or not there exists a succession of choices that result in a particular outcome (i.e. endorsement of the NFA) [24].

Indeed, during the computation of the outcome plan, the strips planner finds multiple choice points that cause a nondeterministic scenario. We choose to envisage the challenge of dealing with nondeterministic planning problems by implementing a backtracking algorithm. Thus, for every choice point where the execution may nondeterministically produce several different outcomes, the planner firstly saves all the possibilities in the stack and then examines them one at a time. In this way, the algorithm can explore all possible different execution paths and elaborate the relative plans.

Choice points: the choice points are closely linked to the PDDL structure; indeed, they not only concern the possible actions executed from a state, as perhaps one would expect, but also the effects and the substitutions to apply at a given moment of the computation.

4.3 Interaction

The project consists of a framework that has no active entity; therefore, there is no active interaction among the components. Once the user defines the domain and the problem, the planner passively utilizes these latter, and the elements therein contained to compute the plans to solve the problem, if they exist.

However, from an interaction perspective, one can find three(or four in case one is using the DSL) entities: the user, who wants to utilize the framework to define the domain for a problem and later the problem itself, the framework who is responsible for the generations of the two elements required by the user and the planner who is asked by

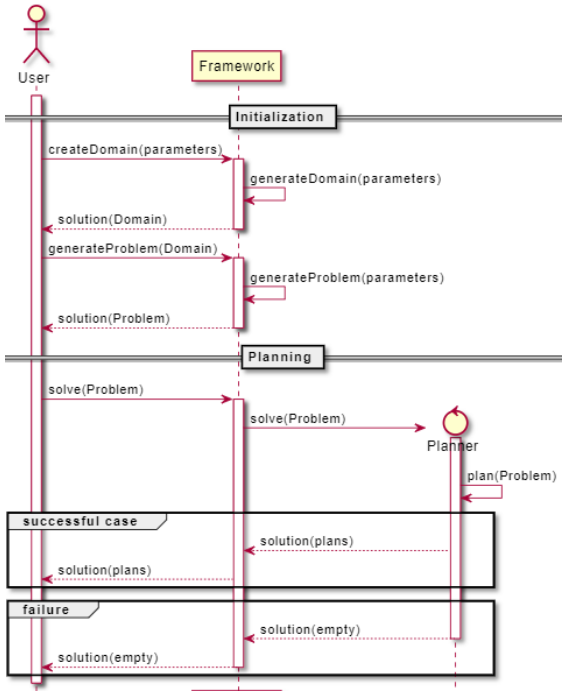

Figure 3: Sequence diagram for the Framework.

the framework to generate the plans to solve a given problem. Furthermore, if the user decides to utilize the framework exploiting the DSL, we have one additional element: the DSL itself, responsible for converting the call made by the user behind the scenes into terms understandable for the framework and calling it itself to solve the query required by the user.

In fig. 3 is shown the sequence diagram for the framework, whereas the fig. 4 illustrates the sequence diagram for the DSL.

5 Implementation Details

5.1 Strips planner

The strips planner is the crux of the matter in the planner model as it is the entity able to trigger the computation of the plan. A pivotal decision in the design of this entity has been the choice of search strategy to be performed. Indeed, the node's order expansion significantly impacts the system's performance.

Thus, the *StripsPlanner* follows the STRIPS algorithm while performing an Iterative deepening depth-first search (IDDFS).

Search Strategy : when it comes to the choice of search strategy, we evaluate several techniques. The approach chosen was the IDDFS: a state space/graph search strategy

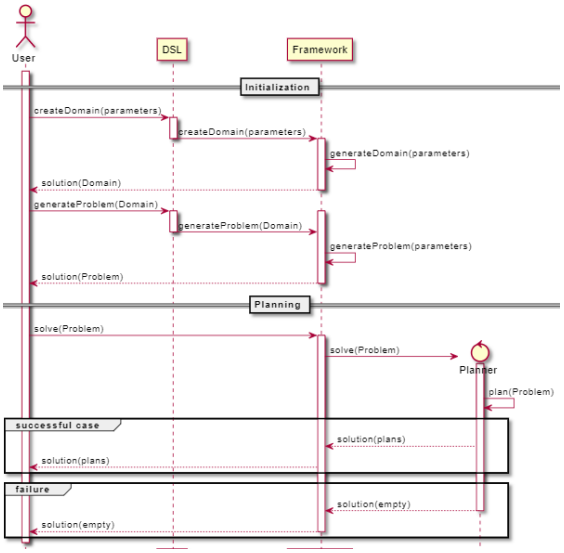

Figure 4: Sequence diagram for the DSL.

in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found [29].

We choose that search strategy because of the advantages it presents compared to other techniques such as the Depth-first search (DFS)[21] or to the Breadth-first search (BFS)[22] because of its characteristics. Thus, the IDDFS manage to combine the advantages of both techniques and overcomes the limitation in terms of space and time required to compute the solution presented in BFS, guaranteeing optimality[23] and completeness[24]. IDDFS combines depth-first search's space efficiency and breadth-first search's fast search (for nodes closer to root).

A comparison among the three search strategies analyzed is proposed in table 3, where:

- b: maximum branching factor of the search tree;
- d: depth of the least-cost solution;
- m: maximum depth of the state space;

[21] **DFS**: it is an algorithm for searching or traversing graph or tree data structures. DFS starts at the root node and explores as far as possible along each branch before backtracking [28].

[22] **BFS**: it is a procedure for searching a tree data structure for a node that satisfies a designated trait. It commences at the tree root and explores all nodes at the current depth ahead of moving on to the nodes at the next depth level. BFS utilizes extra memory to maintain a list of the child nodes that met but not yet examined [26].

[23] **Optimality**: find a feasible plan that optimizes performance in some carefully specified manner, in addition to arriving in a goal state [20].

[24] **Completeness**: a problem is said to be complete if it always finds a solution if one exists [21].

	BFS	**DFS**	**IDDFS**
Complete	Yes (if b is finite).	No: fails in infinite-depth spaces, spaces with loops.	Yes.
Time	$O(b^{d+1})$ (keeps every node in memory).	$O(b^m)$: terrible if m is much larger than d but if solutions are dense, may be much faster than BFS.	$O(b^d)$
Space	$O(b^{d+1})$ (keeps every node in memory).	$O(bm)$	$O(bd)$
Optimal	Yes, if cost is nondecreasing function of node depth.	No.	Yes, if step cost is a nondecreasing function of node depth.

Table 3: Search strategy comparison [21].

Data structure : the Kotlin *Sequences* has been chosen as a data structure to provide the plans computed. The crucial reasons for the *Sequence* type's choice are its lazy evaluation, top-level functions for instantiating sequences and extension functions. Indeed, as the planner may compute numerous plans for each goal, having a data structure eagerly evaluated could be inefficient.

5.2 Logic Paradigm

As advanced in section 3.2 the logic paradigm is used to implement some core processes. Notably, the `2P-Kt` framework is used behind the scenes to implement the following procedures: refreshing the variables, implementing the logic substitution in the `mgu()` and `merge()` methods, and handling the representation of some variables (i.e. Object representation).

Entities representation : in general, to avoid reinventing the wheel, most of the core logic operations, as some entities, are not implemented ex-novo but mapped on `2P-Kt`.

Variable refreshing The "refresh of a variable" allows a formula to be re-used in different contexts, avoiding undesired variable assignments. In practice, a formula is refreshed by consistently replacing each variable contained with some bare new variable of a similar name never used before [4].

That aspect is a central issue to be considered for the project, as without a proper variable refreshing, *spurious substitution* could happen. To avoid the problem, all the entities able to perform a logic substitution provide a refresh method which takes the context as a parameter to perform the refreshing operation. The latter is essential to avoid renaming variables that belong to the same context.

Listing 2: Snippet showing the predicates definition using the framework.

```
object Predicates {
    val at = Predicate.of("at", Types.blocks, Types.locations)
    val on = Predicate.of("on", Types.blocks, Types.blocks)
    val armEmpty = Predicate.of("arm_empty")
    val clear = Predicate.of("clear", Types.blocks)
}
```

Logic substitution A logical substitution application consists in rewriting a formula by assigning variables with another term, as prescribed by a given substitution.

In general, the framework relies on `2P-Kt` to perform the actual logic substitution; for that reason, when a substitution is required, the term to substitute is converted in the correspondent `2P-Kt` entity to call the substitution method, and then re-converted into an element compliant with the current framework.

Unification is an essential aspect of logic, thus consisting of the computation of variable assignments to let a given formula be equal to another one hence fitting a particular context. This operation implement in the framework using the `merge()` method. That latter exploits the `+` operator defined into `2P-Kt` after having converted the terms in `2P-Kt` elements and then re-converts them after having performed the unification operation.

However, in general, several unifiers may exist for any two formulæ the Most General Unifier is an operation which computes a substitution making two formulæ equal.

5.3 DSL

Kotlin provides several mechanisms to implement DSL conveniently: type aliases, context control, operators overloading, extension functions, infix function, lambda with receiver, etcetera.

We use many of them to develop our DSL to enhance the conciseness and readability of the framework while avoiding boilerplate code. Namely, we made great use of the lambda with receiver, combined with the operators overloading, the extension functions, and the infix functions allowing us to implement a DSL which significantly simplifies the framework usage.

The snippets in listing 11 and listing 3 illustrate an example of the simplification introduced by the DSL. Indeed, both snippets of code have the same purpose: the definition of the predicates for a given domain; however one can see that using the DSL the code is more concise and readable compared to the one written using only the framework.

6 Validation

As advanced in section 1.2 we perform the project's evaluation by analyzing some primary dimensions: its internal organization and compliance with the requirements defined

Listing 3: Snippet showing the effects definition using the DSL.

```
predicates {
    +"at"("blocks", "locations")
    +"on"("blocks", "blocks")
    +"arm_empty"
    +"clear"("blocks")
}
```

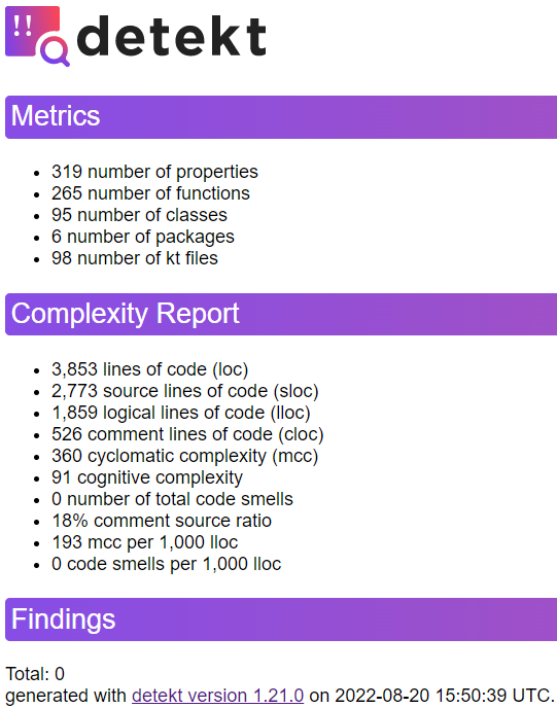

Figure 5: Result of the Detekt examination.

along with the test evaluation.

The code's internal organization follows the Kotlin conventions while avoiding code smells (as shown by the Detekt report in fig. 5; we tried our utmost to respect all requirements examined in section 3.

Furthermore, since we decide to adopt test-driven development, tests play a relevant role in this assessment. We design a test for each entity implemented to check for the preponderance of their functionalities. All the tests created are executed without errors, as shown by the results of the Github action in fig. 6.

In addition, their respect for the coverage requirement expresses in section 3.3 as shown by the results of IntelliJ's coverage tool in fig. 7.

Test Summary

97	0	0	1.750s
tests	failures	ignored	duration

100% successful

Packages Classes

Package	Tests	Failures	Ignored	Duration
default-package	73	0	0	1.639s
dsl	24	0	0	0.111s

Generated by Gradle 7.5 at Aug 20, 2022, 4:03:27 PM

Figure 6: Result of the tests' analysis.

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	95.1% (78/82)	91.6% (208/227)	91% (437/480)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
<empty package name>	92% (23/25)	84.1% (37/44)	81.6% (40/49)
dsl	100% (16/16)	98.3% (57/58)	97% (128/132)
dsl.provider	100% (11/11)	100% (21/21)	100% (22/22)
impl	93.1% (27/29)	89.2% (83/93)	91.6% (230/251)
impl.res	100% (1/1)	90.9% (10/11)	65.4% (17/26)

Figure 7: IntelliJ coverage report.

7 Deployment Instructions

The framework is implemented and distributed as a Gradle project, so the only requirement is the presence of the tool installed on the machine. To utilize the project it is sufficient to:

- download the project from the repository;
- open it with an IDE;
- define a domain and problem, or using the existing ones;
- call the Strips planner, or defining a new planner, passing the problem chosen as an argument;

8 Usage Examples

This section presents some usage examples for the project; however, before getting to the crux of the matter, we believe some preliminary knowledge is needed. One can exploit STRIPS and PDDL to address numerous problems. Indeed, provided that we can depict the problem and the world domain with a finite set of preconditions, effects and actions, it is possible to formulate a PDDL domain and problem to resolve it.

For example, problems that can be represented by employing PDDL and STRIPS are Rubik's cube, stacking blocks, navigating a robot in Shakey's World, and several others.

As a running example for the project, we choose one of the most popular model domains in the history of AI: Block World (BW). *Terry Winograd* designed BW in the 1970s; firstly, its usage did not involve Computer Science fields as its conceiver utilized it for his natural language understanding program, and only after 1975 for studies in computer visions.

The version BW chosen for the project is *Elementary BW* that involves a set of blocks of equal[25] size that a robot can pick up to move in another position (i.e. on top of some other blocks) [7].

We choose the BW domain not only for its popularity in planning studies but also for its versatility and simplicity. A visual example of BW domain is: fig. 8.

As mentioned at the beginning of this section, this part is about the project's running examples. Namely, this section shows how to solve a problem using PDDL, the framework, and the DSL implemented to underline the project's compliance with PDDL and show the main difference between the framework and DSL usage. The three examples have the same purpose: create all the entities needed to generate a plan to solve a problem; we choose that latter in the BW domain; in particular, the problem chosen is about to plan which actions are needed to stack the block a on the block b.

[25] The actual formalization of the Elementary BW domain would require the blocks to be cubic; however the in artefact delivered the blocks physical characteristics are not explicitly model; therefore, the user can assume the blocks to be cubic and equal size.

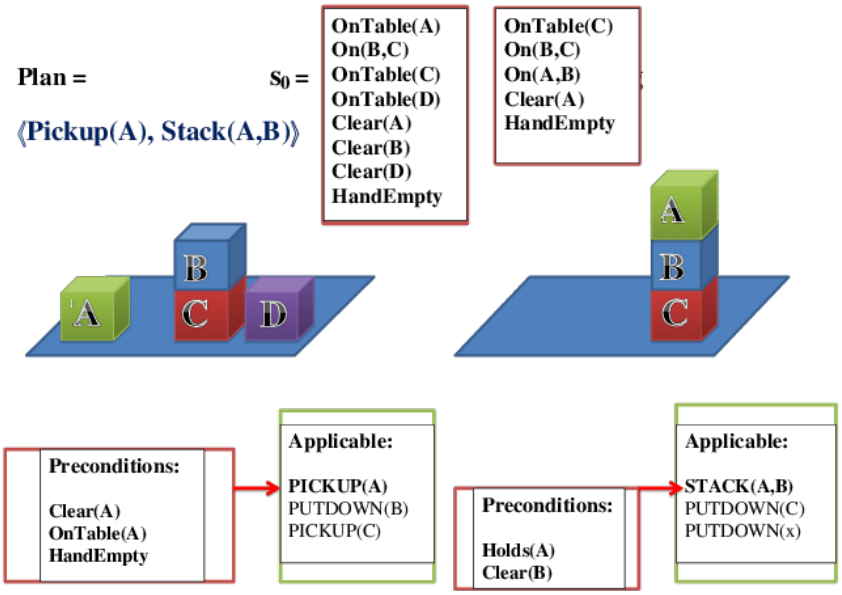

Figure 8: Example of a block world scenario [15]

8.1 PDDL

The first example proposed is the one implemented in PDDL. The listing 4 provides a concise description of the domain required to solve the problem. Whereas listing 5 shows the problem definition.

The main difference between the domain's definition using PDDL compared to the framework and the DSL, presented later in this section, is that PDDL requires a field `Requirements` that is similar to import/include statements in programming languages. This field is needed because PDDL is a remarkably general language, and most planners support only a subset of that. Thus one must declare the subset of the language supported in the `requirements`.

In this example, the Block World domain requires the `: strips` and the `:typing` requirements: the first allows the usage of add and delete effects as specified in STRIPS, while the second allows the use of typing for objects [11] [17].

8.2 Framework

Firstly, we present the example using the framework; since the implementation of the domain requires many elements and defining them within the domain instance itself can result inefficient and quite complex, we decide to use some singleton objects to provide these core elements. Thus, priorly we show those elements and subsequently how to implement the domain and the problem.

Listing 4: Snippet showing the Domain definition using PDDL.

```
(define (domain block_world_domain)

    (:requirements :strips :typing)

    (:types
        anything
        strings - anything
        locations - strings
        blocks - strings
    )

    (:constants
        floor - locations
        arm - locations
    )

    (:predicates
        (at ?X - blocks ?Y - locations)
        (on ?X - blocks ?Y - blocks)
        (arm_empty)
        (clear ?X - blocks)
    )

    (:action pick
        :parameters (?X - blocks)
        :precondition (and (clear ?X) (arm_empty))
        :effect
            (and
                (at ?X ?arm)
                (not (arm_empty))
                (not (clear ?X))
                (not (at ?X floor))
            )

    (:action stack
        :parameters (?X - blocks ?Y -locations)
        :precondition (and (at ?X ?arm) (clear ?Y))
        :effect
            (and
                (on ?X ?Y)
                (clear ?X)
                (arm_empty)
                (not (at ?X ?arm))
                (not (clear ?Y))
            )
    )

    (:action unstack
        :parameters (?X - blocks ?Y - locations)
        :precondition (and (on ?X ?Y) (clear ?X) (arm_empty))
        :effect
            (and
                (at ?X ?arm)
                (clear ?Y)
                (not(arm_empty))
                (not(clear ?X))
                (not(on ?X ?Y))
            )
    )

    (:action putdown
        :parameters (X? - blocks)
        :precondition (and (at ?X arm))
        :effect
            (not (at ?X ?arm))
            (clear ?X)
            (arm_empty)
            (at ?X floor)
        }
    }
)
```

Listing 5: Snippet showing the Domain definition using PDDL.

```
(define (problem block_world_problem)

    (:domain block_world\domain)

    (:objects
        a - blocks
        b - blocks
        c - blocks
    )

    (:init
        (on a floor)
        (on b floor)
        (on c floor)
        (clear a)
        (clear b)
        (clear c)
    )

    (:goal (and (on a b) (at b floor)))
)
```

Listing 6: Snippet showing the values definition for the stack a b problem using the framework

```
object Values {
    val a = Object.of("a")
    val b = Object.of("b")
    val c = Object.of("c")

    val floor = Object.of("floor")
    val arm = Object.of("arm")

    val X = Variable.of("X")
    val Y = Variable.of("Y")
}
```

8.2.1 Core elements

We decide to present the core elements in order of complexity; starting from the most basic: values (listing 6) and types (listing 7), followed by the object set (listing 8), the fluents (listing 9), the predicates (listing 11) and the effects (listing 10), concluding with the most complex the actions (listing 12).

8.2.2 Domain

Using the singleton objects the definition of the domain result simplified as shown in: listing 13.

Listing 7: Snippet showing the types definition for the stack *a b* problem using the framework

```
object Types {
    val anything = Type.of("anything")
    val strings = Type.of("strings", anything)
    val blocks = Type.of("blocks", strings)
    val locations = Type.of("locations", strings)
}
```

Listing 8: Snippet showing the object set definition for the stack *a b* problem using the framework

```
object ObjectSets {
    val all = ObjectSet.of(
        Types.blocks to setOf(Values.a, Values.b, Values.c),
        Types.locations to setOf(Values.floor, Values.arm))
}
```

Listing 9: Snippet showing the fluents definition for the stack *a b* problem using the framework

```
object Fluents {
        val atAFloor = Fluent.positive(Predicates.at, Values.a, Values.floor)
        val atBFloor = Fluent.positive(Predicates.at, Values.b, Values.floor)
        val atCFloor = Fluent.positive(Predicates.at, Values.c, Values.floor)

        val atAArm = Fluent.positive(Predicates.at, Values.a, Values.arm)
        val atBArm = Fluent.positive(Predicates.at, Values.b, Values.arm)
        val atCArm = Fluent.positive(Predicates.at, Values.c, Values.arm)

        val atXFloor = Fluent.positive(Predicates.at, Values.X, Values.floor)
        val atXArm = Fluent.positive(Predicates.at, Values.X, Values.arm)

        val atYFloor = Fluent.positive(Predicates.at, Values.Y, Values.floor)
        val atYArm = Fluent.positive(Predicates.at, Values.Y, Values.arm)

        val armEmpty = Fluent.positive(Predicates.armEmpty)

        val onAB = Fluent.positive(Predicates.on, Values.a, Values.b)

        val clearA = Fluent.positive(Predicates.clear, Values.a)
        val clearB = Fluent.positive(Predicates.clear, Values.b)
        val clearC = Fluent.positive(Predicates.clear, Values.c)
        val clearX = Fluent.positive(Predicates.clear, Values.X)
        val clearY = Fluent.positive(Predicates.clear, Values.Y)
    }
```

Listing 10: Snippet showing the effects definition for the stack a b problem using the framework

```
object Effects {
    val atXFloor = Effect.of(Fluents.atXFloor, true)
    val armEmpty = Effect.of(Fluents.armEmpty, true)
    val onXY = Effect.of(Fluents.onXY)
}
```

Listing 11: Snippet showing the predicates definition for the stack a b problem using the framework

```
object Predicates {
    val at = Predicate.of("at", Types.blocks, Types.locations)
    val on = Predicate.of("on", Types.blocks, Types.blocks)
    val armEmpty = Predicate.of("arm_empty")
    val clear = Predicate.of("clear", Types.blocks)
}
```

8.2.3 Problem

Finally, having instantiated the priors objects and variable, the problem is instantiated as shown in listing 14.

8.2.4 Planner

Defined all the elements needed for the planning problem one can call the Strips planner passing the problem instantiated as argument as demonstrate in listing 18.

As can be seen, despite the previous elements, the planner is not instantiated and utilized using the DSL. Indeed, we decided not to design a DSL entity for the planner because we evaluated that its usage would not have improved by that.

8.3 DSL

This section shows how to obtain the same result using the DSL.

8.3.1 Domain

As in the prior case, firstly, it is necessary to define the domain for the problem. Specifying its name and all the elements used within it: the types for the objects and the variables, followed by the predicates and the actions allowed as shown in listing 16

8.3.2 Problem

The domain definition is followed by problem one. We present that latter in listing 17; where we show how the objects used in the problem are defined, followed by a definition of the initial state and the goal.

Listing 12: Snippet showing the actions definition for the stack a b problem using the framework

```
object Actions {
    val pick = Action.of(
        name = "pick",
        parameters = mapOf(
            Values.X to Types.blocks
        ),
        preconditions = setOf(Fluents.armEmpty, Fluents.clearX),
        effects = setOf(
            Effect.of(Fluents.atXArm),
            Effect.negative(Fluents.armEmpty),
            Effect.negative(Fluents.clearX),
            Effect.negative(Fluents.atXFloor)
        )
    )

    val stack = Action.of(
        name = "stack",
        parameters = mapOf(
            Values.X to Types.blocks,
            Values.Y to Types.locations
        ),
        preconditions = setOf(Fluents.atXArm, Fluents.clearY),
        effects = setOf(
            Effect.of(Fluents.onXY),
            Effect.of(Fluents.clearX),
            Effect.of(Fluents.armEmpty),
            Effect.negative(Fluents.atXArm),
            Effect.negative(Fluents.clearY)
        )
    )

    val unstack = Action.of(
        name = "unstack",
        parameters = mapOf(
            Values.X to Types.blocks,
            Values.Y to Types.locations
        ),
        preconditions = setOf(
            Fluents.onXY,
            Fluents.clearX,
            Fluents.armEmpty
        ),
        effects = setOf(
            Effect.negative(Fluents.clearX),
            Effect.negative(Fluents.onXY),
            Effect.negative(Fluents.armEmpty),
            Effect.of(Fluents.atXArm),
            Effect.of(Fluents.clearY)
        )
    )

    val putdown = Action.of(
        name = "putdown",
        parameters = mapOf(
            Values.X to Types.blocks
        ),
        preconditions = setOf(Fluents.atXArm),
        effects = setOf(
            Effect.negative(Fluents.atXArm),
            Effect.of(Fluents.clearX),
            Effect.of(Fluents.armEmpty),
            Effect.of(Fluents.atXFloor)
        )
    )
}
```

Listing 13: Snippet showing the domain definition for the stack a b problem using the framework

```
val domain = Domain.of(
             name = "block_world",
             predicates = setOf(
                 Predicates.at,
                 Predicates.on,
                 Predicates.armEmpty,
                 Predicates.clear
             ),
             actions = setOf(Actions.pick, Actions.stack, Actions.unStack),
             types = setOf(
                 Types.blocks,
                 Types.locations,
                 Types.anything,
                 Types.strings
             )
         )
```

Listing 14: Snippet showing the problem definition for the stack a b problem using the framework

```
val problem = Problem.of(
             domain = domain,
             objects = ObjectSets.all,
             initialState = State.of(
                 Fluents.atAFloor,
                 Fluents.atBFloor,
                 Fluents.atCFloor,
                 Fluents.armEmpty,
                 Fluents.clearA,
                 Fluents.clearB,
                 Fluents.clearC
             ),
             goal = FluentBasedGoal.of(Fluents.atBFloor, Fluents.onAB)
         )
```

Listing 15: Snippet showing how to call the Strips planner to solve a problem.

```
StripsPlanner().plan(problem)
```

Listing 16: Snippet showing the Domain definition for the stack a b problem using the DSL

```
val domainDSL = domain {
    name = "block_world"
    types {
        +"anything"
        +"strings"("anything")
        +"blocks"("strings")
        +"locations"("strings")
    }
    predicates {
        +"at"("blocks", "locations")
        +"on"("blocks", "blocks")
        +"arm_empty"
        +"clear"("blocks")
    }
    actions {
        "pick" {
            parameters {
                "X" ofType "blocks"
            }
            preconditions {
                +"arm_empty"()
                +"clear"("X")
            }
            effects {
                +"at"("X", "arm")
                -"arm_empty"
                -"at"("X", "floor")
                -"clear"("X")
            }
        }
        "stack" {
            parameters {
                "X" ofType "blocks"
                "Y" ofType "locations"
            }
            preconditions {
                +"at"("X", "arm")
                +"clear"("Y")
            }
            effects {
                +"on"("X", "Y")
                +"clear"("X")
                +"arm_empty"
                -"at"("X", "arm")
                -"clear"("Y")
            }
        }
        "unStack" {
            parameters {
                "X" ofType "blocks"
                "Y" ofType "locations"
            }
            preconditions {
                +"on"("X", "Y")
                +"clear"("X")
                +"arm_empty"
            }
            effects {
                +"at"("X", "arm")
                +"clear("Y")
                -"arm_empty"
                -"clear"("X")
                -"on"("X", "Y")
            }
        }
    }
}
```

Listing 17: Snippet showing the problem definition for the stack a b problem using the DSL.

```
val problemDSL = problem(domainDSL) {
    objects {
        +"blocks"("a", "b", "c")
        +"locations"("floor", "arm")
    }
    initialState {
        +"at"("a", "floor")
        +"at"("b", "floor")
        +"at"("c", "floor")
        +"arm_empty"
        +"clear"("a")
        +"clear"("b")
        +"clear"("c")
    }
    goals {
        +"at"("b", "floor")
        +"on"("a", "b")
    }
}
```

Listing 18: Snippet showing how to call the Strips planner to solve a problem.

```
StripsPlanner().plan(problem)
```

8.3.3 Planner

Finally, after the domain and the problem definition, we can call the Strips planner to compute a plan to solve the defined problem as shown in listing 18. As advanced in the previous paragraph, there is no DSL for the planner; therefore, the code utilized to call it is the same as the one used in the framework example.

9 Conclusions

Driven by the absence of suitably updated planners, we design our planning framework. For this purpose, we base our planning framework on a de-facto standard for planning languages: PDDL. Many versions of PDDL exist to better adapt to different domain problems; since we do not want to address any specific problem, we decided to use the former one, taking into consideration only the core aspects of the languages.

Thus, after a preliminary study of PDDL, we design the core interface for the domain and its tests, and then we realize the problem entities.

To avoid spending time implementing the core logic operations needed to deal with the *logic paradigm*, we decide to use the *2P-kt* framework behind the scene, delegating to that one the management of the variables substitutions, merging operations, etcetera.

Finally, after the planner's completion, we create a DSL to enhance the user's interaction with the framework.

9.1 Future Works

The planner proposed it is in its first implementation; thus, the following versions could focus on many aspects such as the refinement of some of the features implemented, its extension, etcetera.

Firstly, one preliminary improvement could be to change the search strategy from the current one exploiting IDDFS to another utilizing heuristics to select both the goals and the actions. Another central aspect one can improve in the future is the axiom usage as the framework in its current version only supports in definition but does not utilization in a problem definition.

The proposed project follows the features exposed in PDDL 1.2 thus, given its modularity, one could effortlessly extend it to support more advanced characteristics.

Next, the current implementation of the planner only supports the STRIPS one; however, it would be interesting to add more algorithms to solve the planning problem.

9.2 What did we learn

It would be superficial and oversimplified to reduce what we have learned to enhance the fluency in the usage of Kotlin, boost the awareness of the planning problem, or other rather technical skills that I developed during the implementation of this project.

Thus, apart from the expertise we have acquired in technologies we were using, we think considerable improvements concern the development of the right mindset to face the issues raised during the implementation as a growth in the awareness of several thorny matters.

References

[1] Fares Alaboud. Getting started with pddl. `https://fareskalaboud.github.io/LearnPDDL/`. Accessed: 2022-09-03.

[2] R.S Aylett, G.J Petley, P.W.H Chung, B Chen, and D.W Edwards. Ai planning: solutions for real world problems. *Knowledge-Based Systems*, 13(2):61–69, 2000.

[3] Lukáš Chrpa. Modelling languages, knowledge engineering tools, domain reformulation.

[4] Giovanni Ciatto. Automatic inference on horn clauses, February 2022.

[5] Giovanni Ciatto. Knowledge representation with horn clauses, February 2022.

[6] Giovanni Ciatto. Planning in strips with prolog, March 2022.

[7] Stephen Cook and Yongmei Liu. A complete axiomatization for blocks world. *Journal of Logic and Computation*, 13, 12 2001.

[8] M. Fowler and R. Parsons. *Domain-specific Languages*. Addison-Wesley signature series. Addison-Wesley, 2011.

[9] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 12 2003.

[10] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. Pddl - the planning domain definition language. *AIPS-98 Planning Competition Committee*, 08 1998.

[11] Adam Green, Benjamin Jacob Reji, Christian Muise ChrisE2018, Enrico Scala, Francisco Martin Rico Felipe Meneguzzi, Henry Stairs, Mau Magnaguagno Jan Dolejsi, and Jonathan Mounty. Requirements (pddl 1.2). `https://planning.wiki/ref/pddl/requirements/`. Accessed: 2022-09-03.

[12] Adam Green, Benjamin Jacob Reji, Christian Muise, Felipe Meneguzzi Enrico Scala, Francisco Martin Rico, Henry Stairs, Jan Dolejsi, Mau Magnaguagno, and Jonathan Mounty. Planning.wiki - the ai planning & pddl wiki. `https://planning.wiki/`. Accessed: 2022-07-16.

[13] Adam Green, Benjamin Jacob Reji, Christian Muise, Felipe Meneguzzi Enrico Scala, Francisco Martin Rico, Henry Stairs, Jan Dolejsi, Mau Magnaguagno, and Jonathan Mounty. What is planning domain definition language (pddl)? `https://planning.wiki/guide/whatis/pddl`. Accessed: 2022-07-13.

[14] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise, 2019.

[15] Krystian Jobczyk and Antoni Ligeza. Strips in some temporal-preferential extension. In -, 06 2017.

[16] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[17] Paola Mello. Writing planning domains and problems in pddl. `http://lia.deis.unibo.it/Courses/AI/applicationsAI2009-2010/materiale/pddl.html/`. Accessed: 2022-09-03.

[18] Stanford Encyclopedia of Philosophy. The frame problem. `https://plato.stanford.edu/entries/frame-problem/`. Accessed: 2022-07-30.

[19] Andrea Omicini. Planning for intelligent agents, March 2022.

[20] Andrea Roli. Planning, March 2022.

[21] Andrea Roli. Solving problems by searching- uninformed search, February 2022.

[22] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Pearson, 3 edition, 2009.

[23] Mend staff. Renovate docs. `https://docs.renovatebot.com/`. Accessed: 2022-07-30.

[24] John Watrous. Nondeterministic finite automata, October 2020.

[25] Wikipedia. A* search algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm. Accessed: 2022-07-24.

[26] Wikipedia. Breadth-first search. https://en.wikipedia.org/wiki/Breadth-first_search. Accessed: 2022-07-23.

[27] Wikipedia. Closed-world assumption. https://en.wikipedia.org/wiki/Closed-world_assumption. Accessed: 2022-08-04.

[28] Wikipedia. Depth-first search. https://en.wikipedia.org/wiki/Depth-first_search. Accessed: 2022-07-23.

[29] Wikipedia. Iterative deepening depth-first search. https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search. Accessed: 2022-07-23.

[30] Wikipedia. Sri international. https://en.wikipedia.org/wiki/SRI_International. Accessed: 2022-07-16.

[31] Wikipedia. Stanford research institute problem solver. https://en.wikipedia.org/wiki/Stanford_Research_Institute_Problem_Solver. Accessed: 2022-07-16.