**ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA**
**CAMPUS DI CESENA**

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

# Model-Based Contrastive Explanations for XAIP: Towards a General Model and Prototype

Tesi di laurea in
INTELLIGENT SYSTEM ENGINEERING

*Relatore*
**Prof. Giovanni Ciatto**

*Correlatore*
**Prof. Andrea Omicini**

*Candidata*
**Giulia Brugnatti**

Sessione di Laurea Unica
Anno Accademico 2022-2023

# Abstract

Planning is an important sub-field of artificial intelligence (AI) focusing on letting intelligent agents deliberate on the most adequate course of action to attain their goals. Thanks to the recent boost in the number of critical domains and systems which exploit planning for their internal procedures, there is an increasing need for planning systems to become more *transparent* and *trustworthy.* Along this line, planning systems are now required to produce not only plans but also *explanations* about those plans, or the way they were attained. To address this issue, a new research area is emerging in the AI panorama: eXplainable AI (XAI), within which explainable planning (XAIP) is a pivotal sub-field.

As a recent domain, XAIP is far from mature. No consensus has been reached in the literature about what explanations are, how they should be computed, and what they should explain in the first place. Furthermore, existing contributions are mostly theoretical, and software implementations are rarely more than preliminary.

To overcome such issues, in this thesis we design an explainable planning library bridging the gap between theoretical contributions from literature and software implementations. More precisely, taking inspiration from the state of the art, we develop a formal model for XAIP, and the software tool enabling its practical exploitation.

Accordingly, the contribution of this thesis is four-folded. First, we review the state of the art of XAIP, supplying an outline of its most significant contributions from the literature. We then generalise the aforementioned contributions into a unified model for XAIP, aimed at supporting model-based contrastive explanations. Next, we design and implement an algorithm-agnostic library for XAIP based on our model. Finally, we validate our library from a technological perspective, via an extensive testing suite. Furthermore, we assess its performance and usability through a set of benchmarks and end-to-end examples.

**Keywords:** XAIP, Contrastive explanation, model reconciliation.

*To Sofi, you will always be a part of my heart.*

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction

Research efforts on methods to explain the inner logic of learning algorithms and their models began in the early '80s and relevantly boosted over the last decade [45, 51, 40]; because of the prominent use of machine learning (ML) and artificial intelligence (AI) techniques in both academia and industry. Indeed, a large part of these AI and ML-based solutions is affected by a broadly acknowledged issue: their algorithmic opacity [40], which is essentially an unacceptable condition in a world where ML and AI are involved in many (safety-)critical activities [65, 10, 7]. Accordingly, humans' inability to comprehend the result of the behaviour of such techniques can lead to dire consequences; thus in current society, the liability of decisions/actions is still mainly associated with human beings [18, 32]. To complicate the picture, many governments acknowledge citizens' right to receive explanations when AI and ML outcomes may impact their lives [69, 68, 63, 47, 23, 33, 40]. For all those reasons, the issue of the interpretation of ML outcomes is rapidly gaining momentum in recent AI research.

To address that challenge, scholars from different research domains are developing a plethora [67, 66] of scattered techniques and approaches to addressing the interpretation of machine learning algorithms and the generation of human-interpretable explanations, generally known as explainable AI (XAI) [40]. In this framework, explainable planning (XAIP) is an emerging and crucial research area within XAI, focused on explaining AI planning systems to users by leveraging automated planning models and recognizing the role played by humans in the planning loop [28, 67]. XAIP includes topics ranging from epistemic logic to machine learning and techniques from domain analysis to path-finding and goal recognition [38].

To this extent, among the major themes that have emerged within the XAIP playground, we focus on plan explanation.

Plan explanation is amongst the earliest areas of XAIP [39]. It focuses on aiding the users to comprehend the reason that leads a system to suggest a specific plan. In other words, the planning explanation process focuses on presenting the output of some automatic planner into forms which are *intelligible* for humans. These forms may involve the causal [19] and temporal [8] relations' description between plan steps; and the design of interfaces supporting human interaction and understanding. In all such cases, enabling users to question the planning system is fundamental to increasing their understanding of the system's decisions, other than boosting the user's confidence in the system.

Because of its interdisciplinary nature, XAIP attracts the interest of academics with different backgrounds, which develop a number of techniques and approaches to tackle the interpretation of ML and AI methods within the planning domain. As it is natural for an emerging domain, the XAIP landscape is still evolving. Therefore, a definitive systematization of sub-topics and relevant subjects and a conclusive model is still missing. Furthermore, most XAIP contributions proposed so far focus on either theoretical contributions or purpose-specific experiments. Hence, to the best of our knowledge, no efforts have been devoted to engineering a general-purpose software tool for XAIP.

Accordingly, this thesis addresses two crucial challenges on the XAIP scene. Firstly, we tackle the challenge of the hectic conceptual XAIP panorama by providing a concise review of its state of the art, besides supplying an outline of its crucial features and issues. We then generalise the aforementioned contributions into a unified model for XAIP for model-based contrastive explanations through model restriction, meeting the challenge posed by the limited number of XAIP systems software implementations. Additionally, we design and implement a practical solution for our XAIP model. We devise our proposal as a pure-Kotlin library for explaining planning systems, thus endorsing the mission of filling the gap between theoretical and practical contributions in the XAIP landscape.

Along this line, the contribution of this thesis is four-folded. First, we review the state of the art of the XAIP, supplying an outline of its most significant facets and problems. Guided by state-of-the-art research, we develop a proposal for the XAIP model supporting model-based contrastive explanations. Next, we design and implement a pure-Kotlin, algorithm-agnostic library for XAIP based on our model. Finally, we validate our XAIP library's correctness with an extensive testing suite; while we assess its performance and usability with a set of benchmarks and end-to-end examples.

Accordingly, the remainder of this thesis is structured as follows.

Chapter 2 introduces notions and state-of-the-art contributions about the XAIP domain by examining both the planning problem and the explanation background. chapter 2).

In chapter 3, we provide some definitions to support our approach to explanation introducing the formal definition of the system's main entities.

Chapter 4 analyzes requirements and objectives we acknowledged for our proposal.

Chapter 5 begins disclosing details about our framework implementation, consequently, it discusses the validation process describing the metrics used to evaluate the system's performances.

Finally, Chapter 6 concludes this thesis by summarising its main contribution.

# Chapter 2

# State of the Art

This chapter presents state-of-the-art contributions and notions that we will reference in the following chapters of this thesis.

In Section 2.1, we provide a preliminary background about planning, there including its main features, algorithms, and languages.

Section 2.2, in particular, describes well-known planning domains which we extensively exploit in the remainder of this thesis, as running examples.

Finally, Section 2.3 provides an overview of XAIP, starting with a short introduction, followed by a brief road map of its main topics and dimensions.

## 2.1 Planning

This section recalls fundamental notions concerning planning in AI. In particular, Section 2.1.1 gives a broad introduction to the planning problem and a formal explanation of its central components. Next, in Section 2.1.2, we present PPDL, the de-facto standard language for defining planning problems and domains, in practice. Finally, Section 2.1.3 provides an overview of planning algorithms, other than a concise description of well-known planner STRIPS.

### 2.1.1 Fundamental notions

Planning is a term that means different things to different groups of people [46]. In general, we call *classical planning*, or *planning*, the problem of devising a sequence of actions that maps a given initial state to a goal state.

Commonly, planning problems share some common essential elements. In the following paragraphs, we examine those core elements providing a formal definition for each of them, besides some common notions.

**Values:** are symbols representing entities from the domain of the discourse. They can be variables or objects.

More, formally we define the set of values $\mathcal{H}$ as:

$$\mathcal{H} = \mathcal{V} \cup \mathcal{O} \tag{2.1}$$

where:

- $\mathcal{V}$: is a set of variables,
- $\mathcal{O}$: is a set of objects.

**Objects:** represent individual entities which are interesting in the domain of the discourse.

We denote objects as follows:

$$\mathsf{o}_1 \mid \mathsf{o}_2 \mid \mathsf{o}_3 \mid \ldots \tag{2.2}$$

where $\mathsf{o}_i$ are objects' names. We call $\mathcal{O}$ the set of the objects.

**Variables:** represent placeholders for (or references to) unknown entities.

Formally, we denote variables as follows:

$$\mathbf{x}_1 \mid \mathbf{x}_2 \mid \mathbf{x}_3 \mid \ldots \tag{2.3}$$

where $\mathsf{x}_i$ are variables' names. We call $\mathcal{V}$ the set of the variables.

An entity which does not contain any variable is called *ground*.

**Substitution:** is a mapping among variables and objects. Formally, a substitution can either be modelled as a function of the form:

$$\sigma : \mathcal{V} \to \mathcal{H} \tag{2.4}$$

or a set of variables assignments of the form:

$$\sigma = \{\mathbf{x}_1 \mapsto \mathsf{o}_1, \mathbf{x}_2 \mapsto \mathsf{o}_2, \ldots\} \tag{2.5}$$

We call $\Sigma$ the set of substitutions.

**Types:** are names of notable sets of values. All types have a supertype. More formally, we recursively define types as the set $\mathcal{T}$ generated by the following production rule:

$$T := \top \mid \langle \mathsf{t}, T \rangle \tag{2.6}$$

meaning that a type $T$ is either:

- the supertype of all types $\top$,
- or type $\mathsf{t}$ whose super-type is some other $T'$.

**Predicates:** are boolean statements about entities from the domain of the discourse. Let $\mathrm{a} \in \mathbb{N}$ be a natural number, then we define a predicate as:

$$\langle \mathsf{p}, \mathrm{a} \rangle \tag{2.7}$$

where:

- $\mathsf{p}$ is a predicate name,
- a is the number of arguments it takes.

Notably, we call "a" the *arity* of the predicate. We call $\mathcal{P}$ the set of predicates.

**Fluents:** are a ground logic fact which are *true*, or *false* at a given moment. Formally, we denote fluents as follows:

$$\mathsf{p}(\nu_1, \ldots, \nu_{\mathrm{a}}) \tag{2.8}$$

where:

- $\mathsf{p}$ is the predicate name,
- $\nu_1, \ldots, \nu_{\mathrm{a}} \in \mathcal{H}$ are values,
- a is the arity of the fluent.

We call $\mathcal{F}$ the set of all fluents.

**Applying substitutions to fluents.** We say that a fluent is ground if it only contains objects among its values—i.e. if it carries no variable as an argument.

Non-ground fluents may be subject to substitution application. We denote such situation as follows:

$$\begin{gathered} f/\sigma \quad = \quad \mathsf{p}(\nu_1, \ldots, \nu_{\mathrm{a}})/\sigma \quad = \quad \mathsf{p}(\nu_1/\sigma, \ldots, \nu_{\mathrm{a}}/\sigma) \\ \text{where} \\ \nu_i/\sigma = \begin{cases} \nu & \text{if } (\nu_i \equiv \mathsf{x}) \wedge (\mathsf{x} \mapsto \nu) \in \sigma \\ \nu_i & \text{otherwise} \end{cases} \end{gathered} \tag{2.9}$$

In other words, when a substitution $\sigma$ is applied to some fluent $f$, any argument of $f$ which is actually a variable mentioned in $\sigma$ is replaced by the corresponding value mentioned in $\sigma$.

Notably, applying a substitution $\sigma$ to a set of fluents $s = \{f_1, f_2, \ldots\}$ means applying the same substitution to each fluent individually:

$$s/\sigma = \{f_1/\sigma, f_2/\sigma, \ldots\} \tag{2.10}$$

**States:** planning problems usually concern a *state space* that includes all possible situations that could occur. The state space might be both discrete and continuous [46].

Formally, we define a state as a set of fluents. Hence, the set of all possible states is defined as:

$$\mathcal{S} = 2^{\mathcal{F}} \tag{2.11}$$

Finally, we denote individual states by $s, s', s'', \ldots$

**Actions:** represent entities that a planner can exploit to alter the state of the world. An action is specified in terms of its preconditions and post-conditions.

Formally an **action** is:

$$\langle \mathsf{a}, C, C^+, C^- \rangle \tag{2.12}$$

where:

- $\mathsf{a} \in \mathcal{N}$ is an action name from a set of action names $\mathcal{N}$,
- $C \in 2^{\mathcal{F}}$ is a set of preconditions, i.e. a set of fluents which must be true for the action to be applicable,
- $C^+ \in 2^{\mathcal{F}}$ is a set of positive post-conditions, i.e those conditions that become true after the execution of the action.
- $C^- \in 2^{\mathcal{F}}$ is a set of negative post-conditions, i.e the conditions that turn false after the application of the action at the state.

We say that an action is ground if it only contains ground fluents and we call $\mathcal{A}$ the set of all the actions, $\mathcal{C}$ the set of preconditions, $\mathcal{C}^+$ the set of positive post-conditions and $\mathcal{C}^-$ the set of negative post-conditions.

**Unification:** is the process of checking whether two (sets of) fluents can be made equal by applying the same substitution to both of them. More precisely, unification aims at computing the *most general* substitution $\sigma$ making any two sets of fluents $s_1, s_2 \in 2^{\mathcal{F}}$ equal. Provided that such a substitution exists, we call it most general *unifier*. Accordingly, unification is devoted to the mgu function, of the form:

$$\text{mgu} : 2^{\mathcal{F}} \times 2^{\mathcal{F}} \to \Sigma \tag{2.13}$$

In particular, the mgu function is required to compute the substitution $\sigma$ such that

$$\text{mgu}(s_1, s_2) = \sigma \iff s_1/\sigma \equiv s_2/\sigma \tag{2.14}$$

Unification is fundamental to understand whether an action is applicable to a state or not. Formally, we say that an action $\langle \mathsf{a}, C, C^+, C^- \rangle$ is applicable into a state $s$ if and only if its preconditions $C$ unify with some subset $s'$ of the state, i.e. iff:

$$\exists s' \subseteq s : \mathrm{mgu}(s', C) = \sigma \tag{2.15}$$

**Action application:** we use actions to form plans by chaining ground actions which are consequently applied to some initial state to reach some final state. To do so, we leverage the notion of action *application*.

More formally, actions are applied to states via the following function:

$$\mathit{apply} : \mathcal{S} \times \mathcal{A} \to \mathcal{S} \tag{2.16}$$

which computes destination states from input states. In particular, the function is defined as follows:

$$\mathit{apply}(s, \langle \mathsf{a}, C, C^+, C^- \rangle) = \begin{cases} ((s - C^-) \cup C^+)/\sigma & \text{if } \exists s' \subseteq s : \mathrm{mgu}(s', C) = \sigma \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2.17}$$

It is worth noticing that, for each state, only a subset of ground actions is applicable. Hence, we call *unification* the function responsible for checking the applicability of an action to a state.

**Goals:** represent a concise description of one (unknown) state to be achieved. In particular, a goal consists of a ground set of fluents which should be included in the desired state. We call $\mathcal{G} \subseteq 2^{\mathcal{F}}$ the set of all possible goals.

Notably, we say that a goal $g \in \mathcal{G}$ is satisfied into a state $s \in \mathcal{S}$ iff $g \subseteq s$.

**Plans:** are ordered finite lists of ground actions needed to achieve the goal from the initial state. More formally, we denote a plan as follows:

$$[a_1, \ldots, a_i, \ldots, a_n] \in \mathcal{A}^* \tag{2.18}$$

**Domains:** represent the *universal* aspects of a planning problem, i.e. those aspects that do not alter depending on the scenario considered [34].

Formally, we define a domain as:

$$\langle \mathcal{P}, \mathcal{A}, \mathcal{T}, \mathtt{r} \rangle \tag{2.19}$$

where:

- $\mathcal{P}$ is a finite set of predicates,
- $\mathcal{A}$ is a finite set of actions,
- $\mathcal{T}$ is a finite set of types,

We call $\mathcal{D}$ the set of all possible domains.

**Problems:** express 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 [34].

Formally, we define a problem as:

$$\langle \Delta, \mathcal{O}, s_0, \textsc{g} \rangle \tag{2.20}$$

where:

- $\Delta \in \mathcal{D}$ is a domain,
- $\mathcal{O}$ is a finite set of objects,
- $s_0 \in \mathcal{S}$ is the initial state,
- $\textsc{g} \in \mathcal{G}$ is the goal.

We call $\Pi$ the set of problems.

**Planner:** is an entity able to transform the world by devising a sequence of ground actions that lead from the initial state to the goal state.

Formally, we define a planner $\mathbf{p}$ as a function accepting problems as input and producing plans as output:

$$\mathbf{p} : \Pi \to 2^{\mathcal{A}^*} \tag{2.21}$$

It is worth noticing that each planner may output a multitude of plans—as well as none. This is because, for each input problem and its initial state, there may be multiple sequences of actions which lead to the goal. Sometimes, there may be none, because there exists no sequence of action which may lead to the achievement of a given goal, from a given initial state. When this is the case, a planner is assumed to output the empty set—denoting the impossibility to solve the planning problem provided as input.

## 2.1.2 Planning in Practice: The PDDL Language

Planning problems need languages to describe their domain and how they change when actions are applied to reach a goal. The *Planning Domain Definition Language* (PDDL) is a formal knowledge representation language designed to express planning models [36].

It was first developed in 1998 by *Drew McDermott* as a means of facilitating systems comparison; then it become the planning language of the *International*

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

Table 2.1: PDDL versions with the major features introduced in each of them [15, 35].

*Planning Competition* (ICAPS[1]) and the de-facto standard for problem specification of many planning systems.

It must keep in mind that PPDL is not the only modelling language for planning, thus the field of classical planning has seen many representations [24, 57, 4] before PDDL standardized the notations. Accordingly, PDDL was born as an attempt to standardise planning languages to let all competing planners of the International Planning Competition (IPC) use the same input language. To this extent, PDDL provides an abstraction layer to leverage STanford Research Institute Problem Solver [24](STRIPS), Action Description Language [57](ADL) and many other representational languages to support different levels of expressivity. Within this framework, several variants of PDDL emerged in the last years to capture different aspects of the planning problem at the increasing level of complexities, with a particular focus on deterministic problems [36]. More precisely, PDDL characteristics spread from the most essential as type specification for *objects*, *predicates* and *actions*, which can be found in its first version, to the more advanced features as the *numeric variables* and *durative actions* as shown in table 2.1.

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

Listing 2.1: Snippet showing a minimal example of a domain definition in PPDL.

```
(define (problem block_world_problem)

    (:domain block_world)

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

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

    (:goal at(X, arm))
)
```

A running example of PDDL usage to describe a problem definition for a block world domain is proposed in listing 2.1.

### 2.1.3 Planners and Planning Algorithms

Commonly, we call *planner*, or *problem solver* the entity that devises the plan; if a planner is a machine is considered a planning algorithm. Planning algorithms should encapsulate the machinery of planning independently of the domain of application [28].

Planning algorithms may rely on several different approaches to solve the planning problem. In classical planning, two of the most relevant planning categories are linear and non-linear planners and hierarchical and non-hierarchical planners.

**Linear and non-linear planner:** are also called total-order planners and partial-order planners, respectively. Linear planners maintain a partial solution as an ordered list of actions found, while non-linear planners only represent partial-order temporal constraints on actions.

**Hierarchical and non-hierarchical planner:** for non-hierarchical planners, there is no such concept of the importance of a goal; everyone has the same significance for the resolution of the problem, whereas hierarchical planners distinguish between the degrees of relevance of goals and actions, attempting to solve the most relevant first.

In classical planning, users share the same planning model and reason capabilities as the planner, however, this assumption is not always valid. Indeed, users' understanding may differ from that of the planner, in these cases, the user and the planner may aim to solve the same problem by leveraging different models of the problem. We call the problems under the above-described scenario *Multi-Model Planning* (MMP) problems.

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) [56].

### The STRIPS Algorithm

The *Standford Research Institute Problem Solver* is a well-known First-Order Logic [48] language with an associated automated solver devised in 1971 by *Richard Fikes* and *Nils Nilsson* at *Stanford Research Institute International* (SRI International[2]) [75].

More precisely, the STRIPS solver is a linear non-hierarchical planner based on a backward search whose central purpose is to find a sequence of actions to transform a given initial state of the world into a new state that is compliant with the goal proposed. We define the problem space for STRIPS by three entities:

- an initial state of the world,
- a set of actions,
- a goal.

A problem is solved when the solver devises a final state that satisfies the given goal.

**Challenges:** among the most relevant characteristics of STRIPS that determined its popularity, we here focus on the way it addresses two crucial challenges, namely non-determinism and efficiency.

[2]SRI International is an American nonprofit scientific research institute and organization headquartered in *Menlo Park*, California established in 1946 [74].

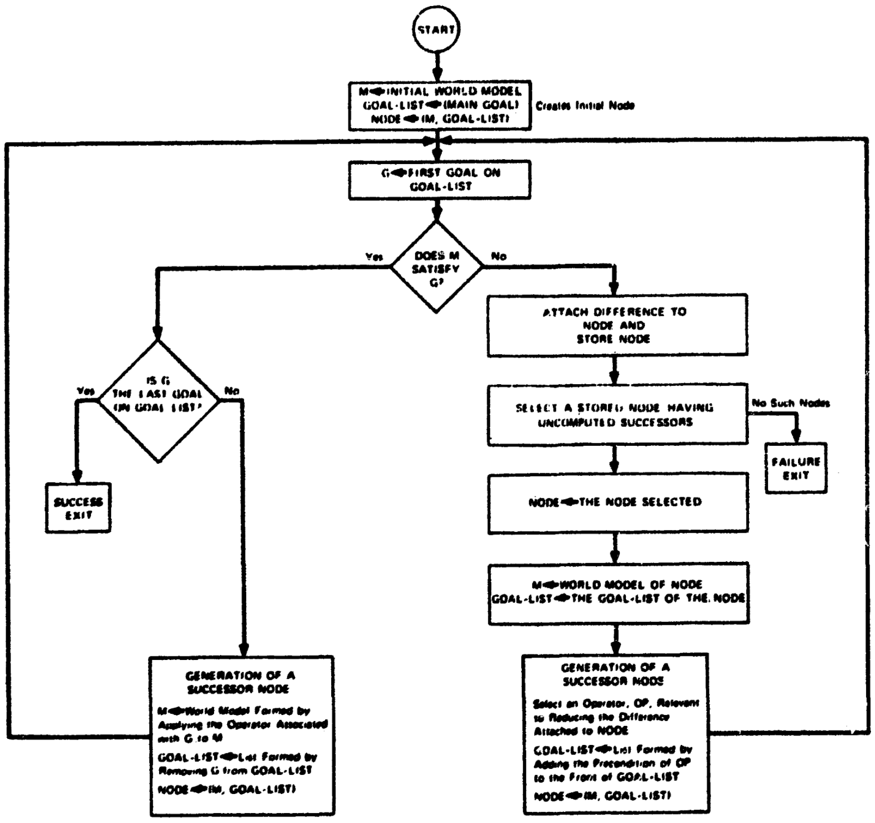
START
M ◀▶ INITIAL WORLD MODEL
GOAL-LIST ◀▶ (MAIN GOAL)
NODE ◀▶ (M, GOAL-LIST)
Creates Initial Node
G ◀▶ FIRST GOAL ON GOAL-LIST
DOES M SATISFY G?
Yes
No
ATTACH DIFFERENCE TO NODE AND STORE NODE
IS G THE LAST GOAL ON GOAL LIST?
Yes
No
SELECT A STORED NODE HAVING UNCOMPUTED SUCCESSORS
No Such Nodes
FAILURE EXIT
SUCCESS EXIT
NODE ◀▶ THE NODE SELECTED
M ◀▶ WORLD MODEL OF NODE
GOAL-LIST ◀▶ THE GOAL-LIST OF THE NODE
GENERATION OF A SUCCESSOR NODE
M ◀▶ World Model Formed by Applying the Operator Associated with G to M
GOAL-LIST ◀▶ List Formed by Removing G from GOAL-LIST
NODE ◀▶ (M, GOAL-LIST)
GENERATION OF A SUCCESSOR NODE
Select an Operator, OP, Relevant to Reducing the Difference Attached to NODE
GOAL-LIST ◀▶ List Formed by Adding the Precondition of OP to the Front of GOAL-LIST
NODE ◀▶ (M, GOAL-LIST)

Figure 2.1: Flowchart of the STRIPS solver from the original paper [24].

**Non-determinism:** non-determinism is an essential 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. STRIPS have to tackle non-determinism each time that: in a state, the solver may apply multiple operators. As shown in fig. 2.1, the planner addresses the problem by devising a hierarchy of goals, sub-goals and states generated by the search process as a *search tree.*

More precisely, the STRIPS planner divides the problem into multiple sub-problems; whenever the solver finds a situation where different actions could be applied. Consequently, it attempts to solve the sub-problem by producing a world model to which the action is applicable. If the solver finds such a model, then it applies the operator and reconsiders the original goal in the resulting state; otherwise, it tries to wig different applicable actions or steps back to the previous decision point if it gets stuck[24].

In the following lines, we leverage an example of non-determinism in a planning context. Let: $\mathrm{i}, \mathrm{j} \in \mathbb{Z}_0^+$. Let:

$$\Delta = \langle \mathcal{P}', \mathcal{A}', \mathcal{T}', \mathtt{r} \rangle \tag{2.22}$$

where:

- $\mathcal{P}'$ is a finite set of predicates,
- $\mathcal{A}' = \{a_0, \ldots, a_i\}$ is a finite set of actions,
- $\mathcal{T}'$ is a finite set of types,

be the domain for the problem $\Pi$, defined as follows: $\Pi = \langle \Delta, \mathcal{O}, s, \textsc{g} \rangle$, where:

- $\Delta \in \mathcal{C}$ is a domain,
- $\mathcal{O}' = \{\mathtt{o}_0, \ldots, \mathtt{o}_j\} \in \mathcal{O}$ is a finite object set,
- $s \in \mathcal{S}$ is a a initial state,
- $\textsc{g} = a_1(x) \in \mathcal{G}$ is a goal.

The problem $\Pi$ has a nonground goal $\textsc{g}$ that leads the solver to a nondeterministic scenario.

Thus, let

$$\mathcal{S} = \{\mathbf{x} \mapsto \mathtt{o}_1, \mathbf{x} \mapsto \mathtt{o}_2, \ldots \mathbf{x} \mapsto \mathtt{o}_j\} \tag{2.23}$$

Once the solver pushes the action $a_1$ on the stack, it could apply any substitution in $\mathcal{S}$ to the variable $\mathbf{x}$. Each of them is a valid variable assignment for the variable $\mathbf{x}$.

**Efficiency:** a primary design problem for the solver is how to satisfy the storage requirements of a search tree in which each node may contain a different state of the world. Thus, given that we may define each state as a large set of fluent and that STRIPS exploit a search tree process, it will be infeasible to recopy the entire state context of the computation each time we generate a new node. In STRIPS, we solve the problem exploiting the *Closed World Assumption*, thus stating that whatever we do not declare as *true* is *false* by default.

**Algorithm:** scholars proposed many implementations of STRIPS in the last 50 years, for this thesis, we design our version of the algorithm.

To this extent, our STRIPS planner addresses the nondeterministic challenge by leveraging a data structure that saves the execution context at the time of choice. In this way, the algorithm can explore all the possible branches and devise all the solutions for a given problem.

Before examining the STRIPS algorithm, we require some preliminary descriptions. Let, $\Pi = \langle \Delta, \mathcal{O}, s, \textsc{g} \rangle$ be a tuple where:

- $\Delta \in \mathcal{C}$ is a domain,
- $\mathcal{O}' \in \mathcal{O}$ is a finite object set,
- $s \in \mathcal{S}$ is a a initial state,
- $\textsc{g} \in \mathcal{G}$ is a goal.

Furthermore, our solver leverage three data structures:

- one that represents the currently simulated state,
- a stack responsible for orderly storing the currently unsatisfied goals and un-applied actions,
- one storing the names of the actions simulated so far.

Before the solver execution, the system is in the initial state ($i$) provided in $\Pi$.

When we trigger the execution, the planner pushes the fluents in the goal, $\textsc{g}$, on the stack. Next, it repeats the following operations until the stack is empty:

- if the top of the stack is fluent:
  - if it is possible to find any substitution that allows the fluent to *unify* with any fluent of the current state, the fluent is removed, and the substitution found is applied to the remaining elements of the stack.
  - if no unification is possible, then we look for an action in $\mathcal{A}'$ that has at least one positive effect that matches with the fluent.
    - if we find it then we push the action and its preconditions on the stack,
    - otherwise, we interrupt the computation and explore the next choice point if it exists, or we interrupt the execution if it does not.
- if the top of the stack is an action, $a$, whose name matches an action's name in $\mathcal{A}'$, then:
  - we remove $a$ from the stack,
  - we apply it the current state:
    - if the application of the action to the state produces at least a new state; we set it as the new current state, we add $a$ to the plan and saves eventually other states into the choice point data structure for later explorations,

∗ if the action does not apply to the current state of the system, that is to say, that the application function cannot produce any states with a current couple of parameters, then the solver explores the next choice point if it exists, or interrupt the computation if it does not.

**Search Strategy:** a crucial decision in the design of the algorithm is the choice of search strategy to apply. We choose the *Iterative Deepening Depth-First Search* (IDDFS): a state space search strategy in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found [73].

We prefer IDDFS because of the advantages it presents compared to other techniques such as the *Depth-first search* (DFS)[3] or to the *Breadth-first search* (BFS)[4] because of its characteristics. Thus, IDDFS manage to combine the advantages of both techniques. On one hand, it overcomes the limitation in terms of space and time required to compute the solution presented in BFS, and on the other, it guarantees the optimality[5] and completeness[6]. IDDFS combines depth-first search space efficiency and breadth-first search fast search (for nodes closer to the root).

A comparison among the three search strategies analyzed is proposed in table 2.2, where:

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

## 2.2 Well-known Planning Domains

Generally, planning domains are formal specifications that describe a relational structure among the elements of the problem.

[3]**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 [72].

[4]**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 [70].

[5]**Optimality**: find a feasible plan that optimizes performance in some carefully specified manner, in addition to arriving in a goal state [60].

[6]**Completeness**: a problem is complete if it always finds a solution if one exists [61].

| | **BFS** | **DFS** | **IDDFS** |
|---|---|---|---|
| **Complete** | Yes (if $b$ is finite). | No: fails in infinite-depth spaces, spaces with loops. | Yes. |
| **Time** | $O(b^{\mathrm{d}+1})$ (keeps every node in memory). | $O(b^{\mathrm{m}})$: terrible if $m$ is much larger than $d$ but if solutions are dense, may be much faster than BFS. | $O(b^{\mathrm{d}})$ |
| **Space** | $\mathrm{O}(b^{\mathrm{d}+1})$ (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 2.2: Search strategy comparison [61].

During the last three decades, scholars develop a substantial benchmark of planning domains to test and evaluate the performance of their algorithm. Among the most relevant examples of planning domains, one can find the well-known problem of the Hanoi Tower [37], the Block World [20], or some more pragmatic domains as the logistics ones [50].

## 2.2.1 Block World

The Blocks World (BW) is one of the most relevant domains for demonstrating planning systems. *Terry Winograd* designed it 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 finite number of cubical blocks of equal [7] size and a surface large enough to hold all of them. The domain also includes an agent (usually in form of a robotic arm) that can pick a single block and move it to another position (i.e. on top of some other blocks, or on the floor) [20].

Many reasons lead planning researchers extensively study blocks-world planning; firstly the simplicity of its concepts makes it intuitive to understand even to non-expert users, secondly because despite the minimalism of its components

[7]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 modelled; therefore, the user can assume the blocks to be cubic and equal in size.

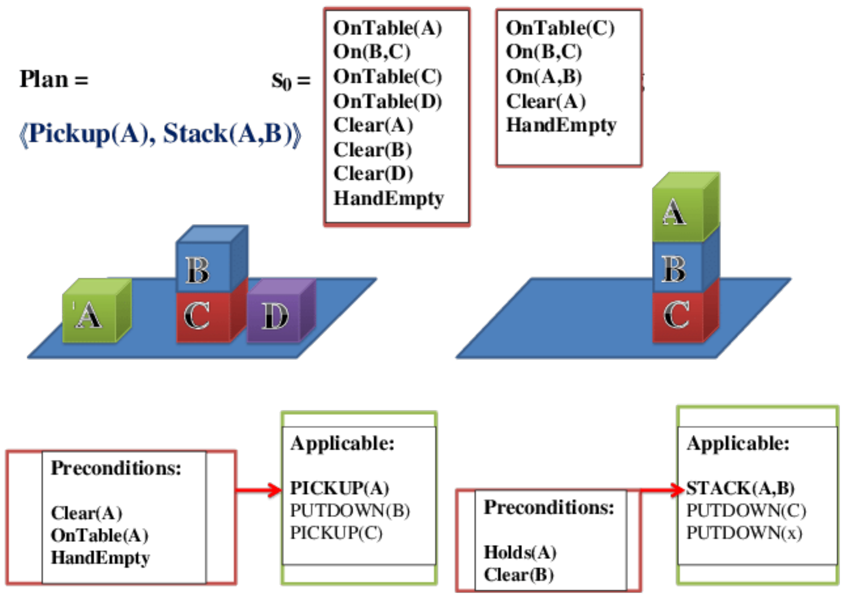

Figure 2.2: Example of a block world scenario [58].

it captures several of the significant challenges posed in planning systems [58]. A visual example of the BW domain is fig. 2.2.

### Formal representation

In this paragraph, we provide an example of a formal representation of the Block World domain according to our model. Let:

- $\mathbf{x} \in \mathcal{V}$ be a variable,
- $\mathbf{y} \in \mathcal{V}$ be a variable,
- $\mathcal{O}' = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\} \in \mathcal{O}$ be a set of objects,
- $s = \{\text{at(a, floor)}, \text{at(b, floor)}, \text{at(c, floor)}, \text{at(d, floor)}, \text{clear(a)}, \text{clear(b)}, \text{clear(c)}, \text{clear(d)}, \text{armEmpty()}\} \in \mathcal{S}$ be the initial state,
- $\textsc{G} = \{\text{on(x)}\} \in \mathcal{G}$ be the goal,
- $\mathcal{P}' = \{\text{on(blocks, blocks)}, \text{at(blocks, locations)}, \text{clear(blocks)}, \text{armEmpty()}\} \in \mathcal{P}$ be a set of predicate,
- $\mathcal{A}' = \{\mathit{pick}(x), \mathit{putdown}(x), \mathit{clearArm}(), \mathit{stack}(x, y), \mathit{unstack}(x, y)\} \in \mathcal{A}$ be set of actions,

- $\mathcal{T}' = \{\langle \textit{blocks, strings}\rangle, \langle \textit{locations, strings}\rangle, \langle \textit{string, anything}\rangle, \langle \textit{numbers, anything}\rangle, \textit{anything}\} \in \mathcal{T}$ be a set of types,
- $\Delta = \langle \mathsf{d}, \mathcal{P}', \mathcal{A}', \mathbf{T}'\rangle \in \mathcal{D}$ be a domain,
- $\Pi = \langle \Delta, \mathcal{O}', s, \mathrm{G}\rangle \in \mathcal{P}$ be a problem.

### 2.2.2 Logistics domain

Logistics domains concern the coordination of the activity of resources that spread from equipment and information to people. Standard logistics planning domains usually involve a set of objects (either package, box or terrain samples) and a simple conveyance (i.e. trucks, planes, or rovers) that must deliver them to different locations.

The primary example of this kind of domain is *Logistics* [50] proposed by *Drew McDermott* and used during the International Planning Competition of 1998 and 2000. *Logistics* is a classical planning domain where vehicles such as trucks and aeroplanes transport parcels within and between cities.

We implement a basic version of the logistics domain proposed by *McDermott* where an agent must transport boxes between different locations. The agent must therefore be able to move in the space, loading objects from their location and unloading them in another.

#### Formal representation

In this paragraph, we provide an example of a formal representation of the Logistics domain according to our model. Let:

- $\mathbf{x} \in \mathcal{V}$ be a variable,
- $\mathbf{y} \in \mathcal{V}$ be a variable,
- $\mathbf{z} \in \mathcal{V}$ be a variable,
- $\mathcal{O}' = \{\texttt{robots(r)}, \texttt{locations(l1, l2, l3, l4, l5, l6, l7)}, \texttt{containers(c1, c2)}\} \in \mathcal{O}$ be a set of objects,
- $s = \{$at(r, l1), inContainer(c1, l2), inContainer(c2, l3), connected(l1, l2), connected(l1, l3), connected(l2, l4), connected(l3, l4), connected(l4, l5), connected(l1, l6), connected(l5, l6), connected(l5, l6), connected(l5, l7), connected(l1, l5), connected(l2, l1), connected(l3, l1), connected(l4, l2), connected(l5, l4), connected(l6, l2), connected(l6, l5), connected(l7, l5), connected(l5, l1)$\} \in \mathcal{S}$ be the initial state,

- $\textsc{g} = \{\text{move(x, y, z)}\} \in \mathcal{G}$ be the goal,
- $\mathcal{P}' = \{\text{connected(locations, locations)}, \text{atLocation(robots, locations)}, \text{loaded(robots, containers)}, \text{inContainerLocation(containers, robots)}, \text{unloaded(robots)}\} \in \mathcal{P}$ be a set of predicate,
- $\mathcal{A}' = \{move(x, y, z), load(z, y, x), unload(z, y, x), stack(x, y), unstack(x, y)\}$ $\in \mathcal{A}$ be set of actions,
- $\mathcal{T}' = \{\langle \textsf{\textit{robots, strings}} \rangle, \langle \textsf{\textit{containers, strings}} \rangle, \langle \textsf{\textit{locations, strings}} \rangle, \langle \textsf{\textit{strings, anything}} \rangle, \textsf{\textit{anything}}\} \in \mathcal{T}$ be a set of types,
- $\Delta = \langle \mathsf{d}, \mathcal{P}', \mathcal{A}', \mathbf{T}' \rangle \in \mathcal{D}$ be a domain,
- $\Pi = \langle \Delta, \mathcal{O}', s, \textsc{g} \rangle \in \mathcal{P}$ be a problem.

## 2.3 Explainable Planning

The demand for explanation within the AI domain is not recent. Since the earliest days of AI, scholars claimed that intelligent systems should be able to describe their results, justifying the reason behind their decisions [45].

In the past decade, advancements in AI and ML techniques (i.e. improvements in accuracy and prediction capabilities) led to a surge in their adoption in a growing number of real-world applications (i.e.traffic control [65], robotics [10], and healthcare [7], etc.) to develop automated or semi-automated systems.

Yet, ML is not the panacea. A significant part of the progress that caused its wide adoption into high-stakes domains was often achieved to the detriment of human interpretability. More specifically, despite the increased predictive power, ML techniques present some disadvantages that make them perform inadequately in some application scenarios. One blatant example is the opacity of the machine learning algorithms, and, more precisely, the impossibility for users to understand the system's decisions that lead to a given outcome. To this extent, the opacity of the algorithms is a remarkable issue in all the contexts where: humans are either accountable for their decisions or expected to provide some sort of explanation for them, even if the decision has been supported by some AI system [32].

Consequently, in the last decade, there has been a soar of research under the umbrella of eXplainable AI (XAI) which addresses that issue. Along this line, eXplainable AI Planning (XAIP) is the sub-field of XAI which addresses the design of trustworthy planners that can interact with humans while their decision-making processes remain transparent.

Because of its interdisciplinary nature, XAIP arouses the interest of academics with widely different backgrounds, which develop a plethora of scattered techniques and approaches to tackle the interpretation of ML and AI methods. As is natural for an emerging domain, the XAIP landscape is still evolving; thus, it is premature to present a definitive systematization into sub-topics and relevant subjects. However, in the following sections, we provide a concise synthesis of its most relevant topics besides a brief taxonomy of the crucial research areas.

### 2.3.1 The Call for Explanations

As advanced in Chapter 1, the need for explanation date back to forty years ago in the era of expert systems [45]. However, in recent years become more prominent because of ML and AI's momentum. Indeed the pervasive usage of such techniques in real-world applications leads to conjunct efforts of the European Union, United States of America, and China governments to impose to design those systems to be able to explain their behaviour in a human-understandable way [63, 47, 23]. One eloquent example in this context is the GDPR [68] regulation which states citizens' *right to explanation* [33] that implicitly requires the AI system to become understandable eventually. Indeed, understanding the system is a requirement to identify potential biases or issues and to guarantee algorithmic fairness, and that system performs as conceived and expected [18].

The field of XAIP addresses this challenge by considering the need for trust, transparency in the decision process, and interaction between humans and machines to comprehend the reasoning behind an AI algorithm decision. To this extent, automated planning solutions are particularly well suited for explanation generation due to their use of symbolic models [8].

### 2.3.2 Taxonomy of XAIP Approaches

As advanced in Section 2.3, to devise a complete taxonomy for the XAIP approaches is premature; nevertheless, in the following paragraphs, we discuss some crucial features and lines of research.

Within the scope of XAIP, explanations can be categorized based on their scope, outcome, and approach to generating them.

**Scope:** the primary distinction among the explanations approach we consider is their scope. In this respect, we distinguish two categories *local* and *global explanations*:

**Local:** local explanations aim to explain a particular decision (local) taken in a planning problem;

- **Global:** global explanations, on the contrary, are geared towards the entire (global) problem model.

**Approaches:** the second category of approaches considered are algorithm and model-based explanations.

- **Algorithm-based:** methods seek to explain the underneath planning algorithm. The major challenge when tailoring explanations for specific algorithms is properly leveraging the algorithm's properties to design an efficacious explanation, thus general concepts within the algorithm domain, such as heuristics values, are hardly understandable for lay users [49].
- **Model-based:** explanations exploit algorithm-agnostic approaches for generating explanations. In this case, one can evaluate the properties of a solution independently of the planner used to devise it. Within this framework, we find two different lines of research:
  - **Domain tailored explanations:** is a research area focused on developing custom explanations based on a specific model. The central issue of this type of explanation is that they require thorough endeavour, besides tailoring explanations for each domain individually leading to effective explanation [49].
  - **General approaches:** field of research which develops explanations based on general planning concepts. In this case, the crucial issue is to design effective explanations considering only general planning concepts.

  Furthermore, recent studies demonstrate that model-based approaches are better suited for an explanation, thus for AI Planning decision-making mechanisms [42, 12]. Indeed, a model-based representation for AI plans is agnostic to the methods used to produce the plan. Model-based approaches rely only on the user-provided solution, domain, and model of the problem; thus, they only examine the properties of a solution independently of the method used to produce it [1].
- **Problem-based:** explanations related to the metric, agents' priorities, set of constraints, edge costs and obstacle/target locations, number of agents, et cetera.

**Output:** the latter category focuses on making the outputs of the planning process more palatable to human decision-makers, to this extent, there are two primary approaches: text-based and visual-based explanations.

**Text-based:** are the most of the relevant solutions proposed on the XAIP panorama [22, 21, 19]. They focus on providing a textual explanation for the planning problem. Text explanations also include every method generating symbols that represent the functioning of the model. These symbols may portray the rationale of the algorithm using a semantic mapping from model to symbols [3].

**Visual-based:** explanations based on visualization [43, 11] leverage on multimedia learning principle, which states that humans learn better from words and pictures than from words alone.

Aware of this taxonomy, scholars recently proposed different lines of research to address XAIP challenges. To the best of our knowledge, the most prominent proposals to generate explanation leverage on plan-property dependencies, or are either model or argumentation-based.

**Through plan-property dependencies:** proposal on this line of research [22, 21] usually works at the level of plan properties which are generally boolean functions on plans, that capture aspect of the plan the user cares. Thus the proposals in this line work with a two stages process; they usually assume that a set of relevant properties is part of the input, and next, they examine the plan space looking for dependencies across plan properties.

**Argumentation-based:** central to these approaches [19] is the use of causality within the planning model. Indeed, they usually leverage some sort of engine to extract causalities which then form a knowledge base of causal links and form arguments with defeasible rules that will be later utilised in the explanations.

These approaches allow for a definition more than a casual representation of plans, they permit multiple types of causality to be distinguished and different causal chunks to be created and combined to generate explanations.

**Model-based:** as advanced in Section 2.3.2 are approaches that do not consider the planner used to devise the solution. Central to this line of research is the proposal of [9]. In this works scholars develop a model-based approach to explanation *as a service*, thus presenting a method to build *contrastive explanations* around an existing planning system. More precisely, they develop a strategy to address the challenge of multi-model planning, by devising a system that enables users to query a planner about a solution that the planner proposes. Initially, users suggest a set of constraints that the system injects into the former model. Consequently, the system produces a new plan and provides a comparison, in terms of actions added and discarded, between the former plan and the new one.

### 2.3.3 The many faces of XAIP

Besides the previous road map about the most relevant approaches, when designing an explanation system, the primary dimensions one has to consider are:

- who is the explainer?
- who is the explainee?
- what is an explanation?
- what is the nature of explanations?

Conventional answers to the above questions in the XAIP domain are:

**Explainer:** is an automatic planner or, in general, a software agent.

**Explainee:** is the end user, thus a person who interacts with the system, asking questions about the proposed solutions. Commonly in automated planning, the explainee can either be a user collaborating with a planner in a decision support setting, a human teammate in a human-robot team, or a direct stakeholder in the robots' plans [12].

**Explanation definition:** one of the challenges of XAIP is to understand what constitutes an explanation. However, because of its interdisciplinary nature, there is a lack of agreement on what it consists of and which features it should have to be effective. Nevertheless, we can affirm that a plan explanation to be considered efficacious and helpful must be understandable to humans. We give a formal definition of plan explanation in Section 3.1.2 for now, we can affirm that it concerns the translation of the planner solutions in a form human-comprehensible.

**Explanation nature:** explanations may come in various forms and have different models. Nevertheless, in general, they either express differences between the domain-action models of the software agent and the lay user or concern discrepancy between the initial and, or the goal state assumptions of the planning problems of the agent and the user [43]. Scholars from cognitive sciences agree that one of the most relevant explanation types is the contrastive explanation.

**Contrastive explanation:** Miller surveyed over 250 research papers in philosophy, psychology, and cognitive science on how people expound on each other. According to that survey, when people request an explication of an event, they usually demand an explanation relative to some contrast case [41, 51].

To this extent, we can interpret explanations as answers to "what-if-things-had-been-different" questions; as we desire to understand how the changes introduced to make the difference omitting the factors that do not [76].

Thus, we can see that contrastive explanations address the problem where a user implicitly or explicitly asks for a comparison among multiple solutions.

# Chapter 3

# Formal model

In this chapter, we provide the formal definitions and methods that support our approach to explanation.

In general, the need for explanation arises when the plan proposed by an automatic planner does not conform with the user's expectation. This work seeks to fill the gap between these mismatched positions by allowing the user to question the system and its decision.

Taking inspiration from [9], we develop our formal model to address the explanation generation enabling the user to add additional constraints to the original model to fit its expectation.

## 3.1 Definitions

A need for explanation in automated planning arises when an automatic solver proposes a plan that does not align with the expectations of the explainee. We devised a proposal to address this issue by allowing an explainee to question a planner about the decisions made. Along these lines, taking a page from [9], in the next paragraphs we define a set of questions an explainee can ask a solver to bridge the gap between their mismatched positions. To answer the questions asked, our system may devise a novel model for the domain, the problem, or both with additional constraints to satisfy the explainee's expectations and provide a proper explanation.

In the following paragraphs, we formally define the main entities for our explanation system model, examining the admissible questions and the process required to generate explanations for them.

| | **Question** |
|---|---|
| $\Theta_1$ | Why is the ground action $a$ used in the plan $\rho$ rather not being used? |
| $\Theta_2$ | Why is the ground action $a$ not used in the plan $\rho$ rather than being used? |
| $\Theta_3$ | Why is the ground action $a$ used in state $s$ rather than the ground action $b$? |
| $\Theta_4$ | Why plan $\rho$ rather than plan $\rho'$? |
| $\Theta_5$ | Why is plan $\rho$ an appropriate solution? |

Table 3.1: Taxonomy of the questions we provide an explanation to.

### 3.1.1 Question

A question is a query formulated by a user when the plan proposed by a planner agent does not match the user's expectations.

Users may ask different questions according to the scenario considered. Taking a page from [9], we identify five types of questions that we summarize in table 3.1.

Given the above considerations, we call $\mathcal{Q}$ the set of all types of questions considered within our model. Formally we define:

$$\mathcal{Q} \triangleq \bigcup_{i=0}^{5} \mathcal{Q}_i \tag{3.1}$$

Furthermore, we call $\mathcal{Q}^C$ the set of explicitly contrastive questions and we formally definite it as follows:

$$\mathcal{Q}^C \triangleq \bigcup_{i=0}^{4} \mathcal{Q}_i \tag{3.2}$$

In the following paragraphs, we proposed a definition for each of the questions identified.

**Question 1:** given a plan $\rho$, a formal question of the type $\mathcal{Q}_1$ is asked of the form:

**Definition 1.** *Why is the ground action $a$ used in the plan $\rho$ rather than not being used?*

Formally a question of the first type is:

$$\langle \Pi, \rho, a \rangle \tag{3.3}$$

where:

- $\Pi \in \mathcal{P}$: is the problem,
- $\rho$: is the plan the user wants to ask questions about,

- $a$: is the ground action the user wants not to appear in the plan.

**Question 2:** given a plan $\rho$ a formal question of the type $\mathcal{Q}_2$ is asked of the form:

**Definition 2.** *Why is the ground action a not used in the plan $\rho$ rather than being used?*

Formally a question of the second type is:

$$\langle \Pi, \rho, a, \mathrm{i} \rangle \tag{3.4}$$

where:

- $\Pi \in \mathcal{P}$: is the problem,
- $\rho$: is the plan the user wants to ask questions about,
- $a$: is the action the use wants to add to the plan $\rho$,
- i: is the position within $\rho$ in which the user wants to insert $a$.

**Question 3:** given a plan $\rho$, a formal question of the type $\mathcal{Q}_3$ is asked of the form:

**Definition 3.** *Why is the ground action $a_i$ used in state s rather than ground action b?*

Formally, a question of the third type is:

$$\langle \Pi, \rho, a, \mathrm{i}, b, s \rangle \tag{3.5}$$

where:

- $\Pi \in \mathcal{P}$: is the problem,
- $\rho$: is the plan the user wants to ask questions about,
- $a$: is the ground action the user wants to remove from the plan $\rho$,
- i: is the potion within $\rho$ in which the user wants to insert $b$ at the place of the former element that occupied that position,
- $b$: is the ground action the user wants to add to the plan $\rho$, in place of $a$,
- $s$ is the state in which the replacement must be applied.

**Question 4:** Given two plans $\rho$ and $\rho'$ a formal question of the type $\mathcal{Q}_4$ is asked of the form:

**Definition 4.** *Why plan $\rho$ rather than plan $\rho$' ?*

Formally a question of the fourth type is:

$$\langle \Pi, \rho, \rho' \rangle \tag{3.6}$$

where:

- $\Pi \in \mathcal{P}$ is the problem,
- $\rho$ is the plan former plan,
- $\rho'$ is the new plan proposed by the user.

**Question 5:** Given a plan $\rho$ a formal question of the type $\mathcal{Q}_5$ is asked of the form:

**Definition 5.** *Why is plan $\rho$ an appropriate solution?*

Formally, a question of the fifth type is:

$$\langle \Pi, \rho \rangle \tag{3.7}$$

is a tuple where:

- $\Pi \in \mathcal{P}$ is the problem,
- $\rho$ is the plan the user wants to have insight about.

### 3.1.2 Explanation

As previously advanced, commonly, the need for explanation arises when a solver proposes a plan that does not match the user's expectation. It may happen because the user expects a specific solution different from the one offered by the planner, or because the solution proposed lacks certain qualities. These expectations commonly arise from a model held by the user that differs from the model used by the planner (MMP). In this context, an explanation is an entity that should reconcile the user's mental model with the one held by the planner. To this extent, we need a model reconciliation whenever different models attempt to describe the same phenomenon and their responses are different. A general solution to achieve common ground is to align these responses altering one or both of the models and trying to bring them closer. For planning models, these modifications may involve changes in the structure of the actions, in the collection of the objects identified in a state, or their properties, as well as a redefinition of the goal of the problem, or the model's constraints [9].

We describe these situations later in this chapter in Section 3.2, but for now, we say that the need for explanation arises when a planner proposes a solution that does not match the user's expectations. In this situation, the system must devise an explanation to reconcile the user's and planner's models. Thus, an explanation should give some insight into a plan so that a user can use it to better comprehend the decision process of the planner. An explanation is in the form:

$$\mathcal{E} \triangleq \bigcup_{i=0}^{1} \mathcal{E}_i \tag{3.8}$$

### General Explanation

A general explanation analyses a single plan at the time, providing some common insight about it. Nevertheless, even if a general explanation does not perform any explicit comparison, it internally uses as evaluation metrics for the plan's features the optimal solution for the problem. To this end, we consider as an optimal solution the plan containing a minor number of grounded actions that accomplish the goal. A general explanation is in form of:

$$\langle \Pi, \rho, \mu \rangle \tag{3.9}$$

is a tuple where:

- $\Pi \in \mathcal{P}$ is an instance of the problem model,
- $\rho$ is the new plan, either proposed by the user or by the framework following the user's suggestions,
- $\mu$ is the minimal solution for $\Pi$.

### Contrastive Explanation

A contrastive explanation is an explanation that focuses on the differences between two proposed solutions. Within our model, a user requires a contrastive explanation when the user's expectation does not conform with the solution proposed by the solver. In this scenario, a contrastive explanation can help to reconcile the user's mental model with the planner's model providing an insight into what would have happened if the planner had executed the solution proposed by the user. Specifically, we design these explanations to highlight the difference between the solution proposed by the solver and the one suggested by the user or obtained by adding the constraints the user suggested. Let: $\mathcal{W} = \{1, \dots, 5\}$ and $i \in \mathcal{W}$, thus we define a contrastive explanation as:

$$\mathcal{E}_2 = \langle \rho, \rho', \Delta, \alpha, \gamma, \xi \rangle \tag{3.10}$$

is a tuple where:

- $\rho$ is the plan the user wants to ask questions about,
- $\rho'$ is the new plan, either created by the user or by the framework following the user's suggestions,
- $\Theta \in \mathcal{Q}^C$ is the user's question,
- $\alpha$ is the list of the actions that are present in the $\rho'$ but not in the $\rho$,
- $\gamma$ are the operators present in the $\rho$ but not in the $\rho'$, are operators present in both plans.
- $\xi$ are the operators present in both $\rho$ and $\rho'$.

### 3.1.3 Explainer

The explainer is the element entitled to act as a bridge between the user and the explanation system, allowing the user to ask an answer to a given question. Formally the explainer is:

$$\mathrm{e} : (\mathcal{Q}) \mapsto \mathcal{E} \tag{3.11}$$

An example of the explainer function is:

$$\mathrm{e}(\Theta) \mapsto \varepsilon \tag{3.12}$$

where:

- e is a function mapping each question to the respective explanation,
- $\Theta \in \mathcal{Q}$ is the question, the user wants an answer to,
- $\varepsilon \in \mathcal{E}$ is the explanation for $\Theta$.

### 3.1.4 Simulator

As mentioned above, to reconcile the mental model of the user and the model exploited by the planner it is often necessary to alter one or both models. Within this framework, we modify the planner model according to the user's suggestions to fit the user's expectations. More specifically, we call *constraints* the suggestions of the users to reconcile the model of the user and the one of the planner. We analyze how to inject the user constraints into the model in Section 3.2; for now, we can say that when a user adds constraints to a model or asks to test the satisfiability of a plan proposal, the explanation system requires a way to check if the user's suggestion results in a valid solution for the proposed problem, or if otherwise, they prevent to find a plan that satisfies the goal. The simulator is the component of the system that: given a plan $\rho$ and a state $\mathrm{s}$, can simulate its execution to control if it leads to an acceptable solution for the problem.

Let $\mathcal{S}' \subseteq \mathcal{S}$ and $\mathcal{A}' \subseteq \mathcal{A}$, formally, the simulator is:

$$\mathrm{s} : (\mathcal{S}, \mathcal{A}') \mapsto \mathcal{S}' \tag{3.13}$$

An example of the simulation function is:

$$\mathrm{s}(\mathrm{s}, \rho) \mapsto \mathcal{S}'' \tag{3.14}$$

where:

- s: is the simulator function that map a pair of state ($s$), plan ($\rho$) into the set of states ($\mathcal{S}''$),
- $s$: is the initial state for the plan execution,
- $\rho$: is the proposed plan whose satisfiability is to check,
- $\mathcal{S}''$: is a set of states reachable for simulating the execution of the plan $\rho$ using $\mathrm{s}$ as the initial state.

## 3.2 Compilation

We call *compilation* the process performed by the system to extract the user's suggestions from the question and inject them into the model. Thus, our explanatory system performs a compilation process of the user constraints into the former problem by creating a new model, called *hypothetical domain*, and a new problem named *hypothetical problem*. Then the system uses these two elements to generate the new plan (*hypothetical plan*) complaint with the user suggestions. We depict the main idea of the compilation process in fig. 3.1.

In the following sections, we give a formal description of the compilation process of each question proposed in table 3.1.

### 3.2.1 Removal of an Operator from a Plan

Let:

- $\Delta \in \mathcal{D}$ be a domain,
- $\mathcal{O}' \in \mathcal{O}$ be a set of objects,
- $s \in \mathcal{S}$ be the initial state,

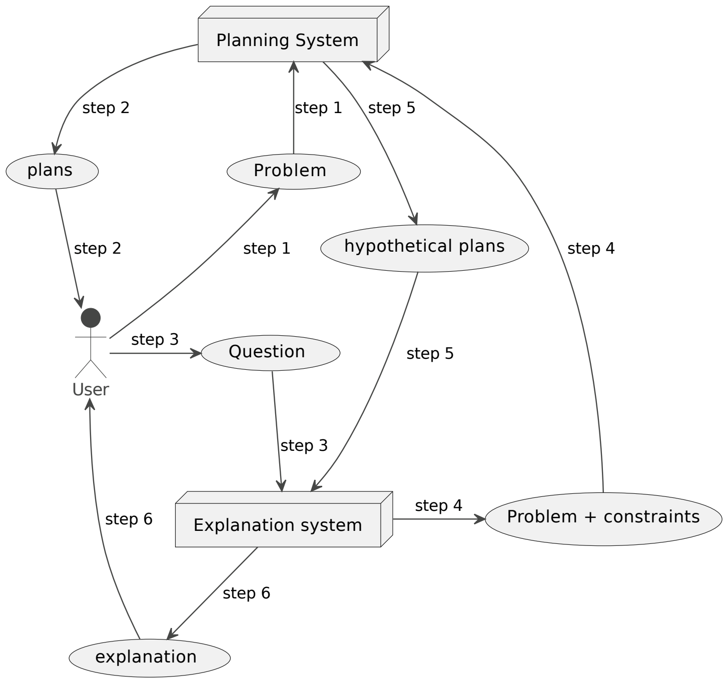

Figure 3.1: Interaction process that leads to the generation of an explanation.

- $\text{G} \in \mathcal{G}$ be the goal,
- $\mathcal{P}' \in \mathcal{P}$ be a set of predicate,
- $\mathcal{A}' \in \mathcal{A}$ be set of actions,
- $\mathcal{T}' \in \mathcal{T}$ be a set of types,
- $\Theta \in \mathcal{Q}_1$ be a question of a user.

Given a plan $\rho$, a problem $\Pi = \langle \Delta, \mathcal{O}', s, \text{G} \rangle$ where the domain is: $\Delta = \langle \mathsf{d}, \mathcal{P}', \mathcal{A}', \mathbf{T}' \rangle$ a question of the first type $\Theta$ is asked of the form:

**Question 1.** *Why is ground action a used in the plan $\rho$ rather than not being used?*

To coerce the planning system to remove the ground action $a$ from the plan $\rho$; it is necessary to make each plan that contains that $a$ unacceptable. To do so, we create a new fluent and, consequently, a new predicate: not_executed_action that represents the ground action the system has not yet performed. Thereby, we add among the fluent of the goal a fully instantiated version (ground) of not_executed_action, to state which operator the planner must not execute for the produced plan to be considered acceptable. Furthermore, we create a new action, $a'$, which is equal to $a$, except for its negative effects which also include not_execute_action. The execution of $a'$ will invalidate the plan as it removes from the goal the new ground fluent. Formally, the compilation process creates a hypothetical domain as follows:

$$\Delta' = \langle \mathsf{d}, \mathcal{P}', \mathcal{A}', \mathcal{T} \rangle \tag{3.15}$$

where:

- $\mathsf{d}$: is the former domain name,
- $\mathcal{P}'$: new set of predicates, defined as follows:

$$\mathcal{P}' = \mathcal{P} \cup \{\text{not\_executed\_action}\} \tag{3.16}$$

- $\mathcal{A}'$: new set of actions, defined as follows:

$$\mathcal{A}' = \mathcal{A} \cup \{a\} \tag{3.17}$$

- $\mathcal{T}$: is the former set of types.

and a hypothetical problem defined as:

$$\Pi' = \langle \Delta', \mathcal{O}', \textsc{s}, \textsc{g} \rangle \tag{3.18}$$

where:

- $\Delta$: is the hypothetical domain,
- $\mathcal{O}'$: is the former set of objects,
- $\textsc{s}$: is the former initial state,
- $\textsc{g}$: is the former goal.

### Example

Let:

- $\mathbf{x} \in \mathcal{V}$ be a variable,
- $\mathbf{y} \in \mathcal{V}$ be a variable,
- $\mathcal{O}' = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\} \in \mathcal{O}$ be a set of objects,

- $s = \{\text{at(a, floor)}, \text{at(b, floor)}, \text{at(c, floor)}, \text{at(d, floor)}, \text{clear(a)}, \text{clear(b)}, \text{clear(c)}, \text{clear(d)}, \text{armEmpty}\} \in \mathcal{S}$ be the initial state,
- $\text{G} = \{\text{on(a,b)}\} \in \mathcal{G}$ be the goal,
- $\mathcal{P}' = \{\text{on(blocks, blocks)}, \text{at(blocks, locations)}, \text{clear(blocks)}, \text{armEmpty}\} \in \mathcal{P}$ be a set of predicate,
- $\mathcal{A}' = \{\mathit{pick}(x), \mathit{putdown}(x), \mathit{clearArm}(), \mathit{stack}(x, y), \mathit{unstack}(x, y)\} \in \mathcal{A}$ be set of actions,
- $\mathcal{T}' = \{\mathsf{\textit{blocks}}, \mathsf{\textit{locations}}, \mathsf{\textit{string}}, \mathsf{numbers}, \mathsf{\textit{anything}}\} \in \mathcal{T}$ be a set of types,
- $\Delta = \langle \mathsf{d}, \mathcal{P}', \mathcal{A}', \mathbf{T}' \rangle \in \mathcal{D}$ be a domain,
- $\Pi = \langle \Delta, \mathcal{O}', s, \text{G} \rangle \in \mathcal{P}$ be a problem,
- $\rho = [\mathit{pick}(a), \mathit{stack}(a, b), \mathit{pick}(c)]$ the user wants to ask question about,
- $\mathit{pick}(c)$ the action the user wants to remove from the plan. This may happen because the user realizes that the plan $\rho$ which the user has in mind may contain a ground action that is not useful to accomplish the goal.

To compel the solver to not execute the ground action $\mathit{pick}(c)$ the compilation process must modify both $\Pi$ and $\Delta$. Thus starting from the domain, the procedure creates a new predicate not_executed_action, adds it to $\mathcal{P}'$ and consequently generates the respective fluent: not_executed_action. At this point, the process devises a new action using as a template the action the user wants to remove from the plan. We call such new action $\mathit{pick}'(x)$. $\mathit{pick}'(x)$ is equal to $\mathit{pick}(x)$ except for its negative post-conditions which also contain the new fluent not_executed_action. At this point, the compilation alters the set of action $\mathcal{A}'$ removing $\mathit{pick}(x)$ and adding $\mathit{pick}'(x)$ to $\mathcal{A}'$. Additionally, the compilation considers the problem $\Pi$ adding the ground version of the fluent not_executed_action among the goal of $\Pi$. The final plan according to the user suggestion is: $\rho' = [\mathit{pick}(a), \mathit{stack}(a, b)]$

More formally, the new problem created in the compilation process is $\Pi' = \langle \Delta', \mathcal{O}', s, \text{G}' \rangle$ where:

- $g' = \{\text{on(a,b)}, \mathit{not\_executed\_pick}(c)\} \in \mathcal{G}$ is the new goal,
- $\Delta = \langle \mathsf{d}, \mathcal{P}'', \mathcal{A}'', \mathbf{T}' \rangle \in \mathcal{D}$ is new a domain; where:
- $\mathcal{P}'' = \{\text{on(blocks, blocks)}, \text{at(blocks, locations)}, \text{clear(blocks)}, \text{armEmpty}, \text{not\_executed\_action}\} \in \mathcal{P}$ is the new set of predicate,

- $\mathcal{A}' = \{pick'(x), putdown(x), clearArm(), stack(x,y), unstack(x,y)\} \in \mathcal{A}$ is new set of actions.

In this last formulation of the problem $\Pi'$ we explicitly represent only the elements the compilation process alters.

### 3.2.2 Insertion of an Operator to a Plan

Let:

- $\Delta \in \mathcal{D}$ be a domain,
- $\mathcal{O}' \in \mathcal{O}$ be a set of objects,
- $s \in \mathcal{S}$ be the initial state,
- $\textsc{g} \in \mathcal{G}$ be the goal,
- $\mathcal{P}' \in \mathcal{P}$ be a set of predicate,
- $\mathcal{A}' \in \mathcal{A}$ be set of actions,
- $\mathcal{T}' \in \mathcal{T}$ be a set of types,
- $\Theta \in \mathcal{Q}_2$ be a question of a user.

Given a plan $\rho$, a problem $\Pi = \langle \Delta, \mathcal{O}', s, \textsc{g} \rangle$ where the domain is: $\Delta = \langle \mathsf{d}, \mathcal{P}', \mathcal{A}', \mathbf{T}' \rangle$ a question of the second type $\Theta$ is asked of the form:

**Question 2.** *Why is the ground action a not used in the plan $\rho$ rather than being used?*

To compel the planner to perform $a$, it is necessary to create a new fluent and consequently a new predicate, exectuted_action, that signals to the system the execution of an operator. We must expand the goal $\textsc{g}$ with executed_action to express which ground action the planner must perform for the final plan to be admissible. Likewise, the positive effects of the action associated with the ground action $a$ must include executed_action, we call that new action $a'$. Accordingly, only the execution of the ground action chosen by the user will produce an acceptable plan.

Formally, the compilation process creates a hypothetical domain as follows:

$$\Delta' = \langle \mathsf{d}, \mathcal{P}', \mathcal{A}', \mathcal{T}' \rangle \tag{3.19}$$

where:

- $\mathsf{d}$: is the former domain name,

- $\mathcal{P}'$: new set of predicates, defined as follows:

$$\mathcal{P}' = \mathcal{P} \cup \{executed\_action\} \tag{3.20}$$

- $\mathcal{A}'$: new set of actions, defined as follows:

$$\mathcal{A}' = \mathcal{A} \cup \{a\} \tag{3.21}$$

- $\mathcal{T}'$: is the former set of types.

and a hypothetical problem defined as:

$$\Pi' = \langle \Delta', \mathcal{O}', s, \text{G} \rangle \tag{3.22}$$

where:

- $\Delta'$: is the hypothetical domain,
- $\mathcal{O}' \in \wr$: is the former set of objects,
- $s$: is the former initial state,
- G: is the former goal.

**Example**

Let:

- $\mathbf{x} \in \mathcal{V}$ be a variable,
- $\mathbf{y} \in \mathcal{V}$ be a variable,
- $\mathcal{O}' = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\} \in \mathcal{O}$ be a set of objects,
- $s = \{\text{at(a, floor)}, \text{at(b, floor)}, \text{at(c, floor)}, \text{at(d, floor)}, \text{clear(a)}, \text{clear(b)}, \text{clear(c)}, \text{clear(d)}, \text{armEmpty}\} \in \mathcal{S}$ be the initial state,
- $\text{G} = \{\text{on(a,b)}\} \in \mathcal{G}$ be the goal,
- $\mathcal{P}' = \{\text{on(blocks, blocks)}, \text{at(blocks, locations)}, \text{clear(blocks)}, \text{armEmpty}\} \in \mathcal{P}$ be a set of predicate,
- $\mathcal{A}' = \{pick(x), putdown(x), clearArm(), stack(x, y), unstack(x, y)\} \in \mathcal{A}$ be set of actions,
- $\mathcal{T}' = \{\mathit{blocks}, \mathit{locations}, \mathit{string}, \mathit{numbers}, \mathit{anything}\} \in \mathcal{T}$ be a set of types,

- $\Delta = \langle \mathsf{d}, \mathcal{P}', \mathcal{A}', \mathbf{T}' \rangle \in \mathcal{D}$ be a domain.
- $\Pi = \langle \Delta, \mathcal{O}', s, \textsc{g} \rangle \in \mathcal{P}$ be a problem.
- $\rho = [\mathit{pick}(a), \mathit{stack}(a, b)]$ the user wants to ask question about,
- $\mathit{pick}(c)$ the action the user wants to add to the plan. This may happen because the user wants o to investigate how the solver would handle the adjunct of useless action to the plan.

To enforce the solver to execute $\mathit{pick}(c)$ the compilation process must modify both $\Pi$ and $\Delta$. Thus starting from the domain, the procedure creates a new predicate executed_action, adds it to $\mathcal{P}'$ and consequently generates the respective fluent: executed_action. At this point, the process devises a new action using as a template the action the user wants to remove from the plan, we call such new action $\mathit{pick}'(x)$. $\mathit{pick}'(x)$ is equal to $\mathit{pick}(x)$ except for its positive post-conditions that also contain the new fluent executed_action. At this point, the compilation alters the set of action $\mathcal{A}'$ removing $\mathit{pick}(x)$ and adding $\mathit{pick}'(x)$ to $\mathcal{A}'$. Finally, the compilation considers the problem $\Pi$ adding the ground version of the fluent executed_action among the goal of the plan. The final plan according to the user suggestion is: $\rho' = [\mathit{pick}(a), \mathit{stack}(a, b), \mathit{pick}(c)]$

More formally, the new problem created in the compilation process is $\Pi' = \langle \Delta', \mathcal{O}', s, \textsc{g}' \rangle$ where:

- $g' = \{\text{on(a,b)}, \mathit{executed\_pick}(c)\} \in \mathcal{G}$ is the new goal,
- $\Delta = \langle \mathsf{d}, \mathcal{P}'', \mathcal{A}'', \mathbf{T}' \rangle \in \mathcal{D}$ is new a domain; where:
- $\mathcal{P}'' = \{\text{on(blocks, blocks)}, \text{at(blocks, locations)}, \text{clear(blocks)}, \text{armEmpty}, \text{executed\_action}\} \in \mathcal{P}$ is the new set of predicate,
- $\mathcal{A}' = \{\mathit{pick}'(x), \mathit{putdown}(x), \mathit{clearArm}(), \mathit{stack}(x, y), \mathit{unstack}(x, y)\} \in \mathcal{A}$ is new set of actions.

In this last formulation of the problem $\Pi'$ we explicitly represent only the elements the compilation process alters.

### 3.2.3 Replacement of an Operator in a Plan

Let:

- $\Delta \in \mathcal{D}$ be a domain,
- $\mathcal{O}' \in \mathcal{O}$ be a set of objects,

- $s \in \mathcal{S}$ be the initial state,
- $\text{G} \in \mathcal{G}$ be the goal,
- $\mathcal{P}' \in \mathcal{P}$ be a set of predicate,
- $\mathcal{A}' \in \mathcal{A}$ be set of actions,
- $\mathcal{T}' \in \mathcal{T}$ be a set of types,
- $\Theta \in \mathcal{Q}_3$ be a question of a user.

Given a plan $\rho = [a_1, a_2, \ldots, a_n]$, a problem $\Pi = \langle \Delta, \mathcal{O}', s, \text{G} \rangle$ where the domain is: $\Delta = \langle \mathsf{d}, \mathcal{P}', \mathcal{A}', \mathbf{T}' \rangle$ a question of the third type $\Theta$ is asked of the form:

**Question 3.** *Why is ground action $a_i$ used in state s' rather than operator $b$?*

To replace $a$ in the final plan there is no need to include additional elements to the system; indeed, it is only necessary to force it to execute $b$ in $s$ obtaining a new state $s''$. Then $s'$ becomes a new initial state for the problem (hypothetical problem):

$$\Pi' = \langle \Delta, \mathcal{O}', s'', \text{G} \rangle \tag{3.23}$$

where:

- $\Delta$: is the former domain,
- $\mathcal{O}'$: is the former set of objects,
- $s'$: is the new initial state defined as follows:

$$\text{a}(s, b) \mapsto \mathsf{s}' \tag{3.24}$$

- G: is the former goal.

Next, we generate the hypothetical plan ($\Pi' = \langle a_{i+1}, a_{i+2}, \ldots, a_n \rangle$) from the hypothetical problem. Consequently, the final plan is obtained by concatenating the operators of $\rho$ until $a$, with $b$ and operators within the $\Pi'$. Hence, the resulting plan is: $\langle a_1, a_2, \ldots, a_i - 1, b, a_{i+1}, a_{i+2}, \ldots, a_n \rangle$

**Example**

Let:

- $\mathbf{x} \in \mathcal{V}$ be a variable,
- $\mathbf{y} \in \mathcal{V}$ be a variable,
- $\mathcal{O}' = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\} \in \mathcal{O}$ be a set of objects,
- $s = \{\text{at(a, floor)}, \text{at(b, floor)}, \text{at(c, floor)}, \text{at(d, floor)}, \text{clear(a)}, \text{clear(b)}, \text{clear(c)}, \text{clear(d)}, \text{armEmpty}\} \in \mathcal{S}$ be the initial state,
- $\text{G} = \{\text{on(a,x)}\} \in \mathcal{G}$ be the goal,
- $\mathcal{P}' = \{\text{on(blocks, blocks)}, \text{at(blocks, locations)}, \text{clear(blocks)}, \text{armEmpty}\} \in \mathcal{P}$ be a set of predicate,
- $\mathcal{A}' = \{pick(x), putdown(x), clearArm(), stack(x, y), unstack(x, y)\} \in \mathcal{A}$ be set of actions,
- $\mathcal{T}' = \{\textit{blocks}, \textit{locations}, \textit{string}, \textit{numbers}, \textit{anything}\} \in \mathcal{T}$ be a set of typesan,
- $\Delta = \langle \mathsf{d}, \mathcal{P}', \mathcal{A}', \mathbf{T}' \rangle \in \mathcal{D}$ be a domain,
- $\Pi = \langle \Delta, \mathcal{O}', s, \text{G} \rangle \in \mathcal{P}$ be a problem,
- $\rho = [pick(a), stack(a, b)]$ the user wants to ask question about,
- $stack(a, c)$ the action the user wants to take $pick(a)$ place in the plan,
- $s'$ the in which the action $stack(a, c)$ should be executed in place of $stack(a, b)$.

In this latter case, there is no need to alter $\Delta$, we only need the compilation to replace $s$ with $s$' in $\Pi$. More formally the compilation process creates a new problem $\Pi'$ defined as follows: $\Pi' = \langle \Delta, \mathcal{O}', s', \text{G} \rangle \in \mathcal{P}$. At this point, we can calculate the new plan without further modification to our model. The nontrivial step is to handle the creation of the final plan; thus, it shall also contain the ground actions in the original plan before $stack(a, b)$. The final plan according to the user suggestion is: $\rho' = [pick(a), stack(a, c)]$

### 3.2.4 Comparison of plans

Let $\Theta \in \mathcal{Q}_4$. Given two plans $\rho$ and $\rho'$ a formal question $\Theta$ is asked of the form:

**Question 4.** *Why plan $\rho$ rather than plan $\rho$' ?*

In this case, there is no need for compilation as the user only asks for a comparison between two plans, not to change the problem to fit its expectations.

### 3.2.5 Request for plan properties

Let $\Theta \in \mathcal{Q}_5$. Given a plan $\rho$ a formal question $\Theta$ is asked of the form:

**Question 5.** *Why is plan $\rho$ an appropriate solution?*

As for the former question, the compilation process is unnecessary because there are no constraints to add to the plan; the user only requires insight into $\rho$.

# Chapter 4

# Design

Armed with the formal description of our XAIP model, in this chapter, we examine the design and the architecture of our software implementation proposal along with possible means of interaction.

Namely, this chapter leverages the formal models provided in Chapter 2 and Chapter 3 to envisage a software solution to reduce the gap between concepts and implementation. In Section 4.1, we present the design of the architecture of our solution, examining its core entities and some significant design choice. Section 4.2 discusses how to interact with the different components of the solution.

## 4.1 Architectural Design

We divide our project into multiple micro-modules, each representing a contribution to our proposal. Figure 4.1 illustrates the dependency graph depicting the project's overall architecture. More specifically, within the figure, we find the four modules that form part of our proposal and a set of arrows that connect them. Each arrow indicates a directed dependency from one module to another. As one can notice from the figure, we design each module to be as self-contained as possible; however, some dependencies are inevitable for the coherence of the system; thus, all the modules depend on the planning module as it provides the core entities to use within the system.

More specifically, the planning module is a completely self-contained module that includes our software proposal for the planning formal model. The planning module is modular and extensible to better comply with the users' needs and application scenarios. To this extent, we exploit the APIs of the planning module within the different modules of the project to build other abstractions remaining consistent.

The explanation module is the centre of the project, as it contains the entity

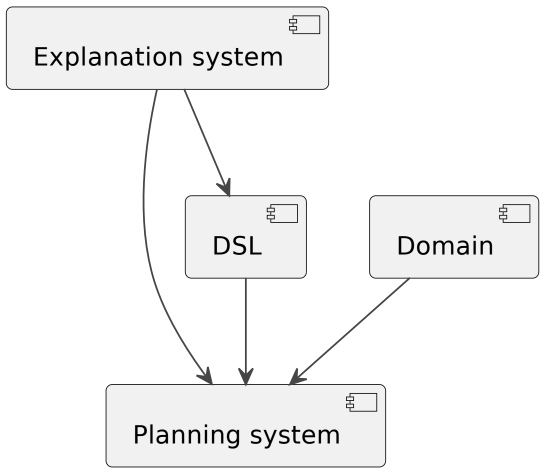

Figure 4.1: Modules and Architectural Dependencies of the Project.

needed to address the challenges proposed by the XAIP domain. Coherently, as advanced in the previous paragraph, it depends on the abstractions of the planning module that we exploit to build the explanations.

The DSL module is a utility module to enhance the usage of the planning module. More specifically, this module provides a handy way to exploit the planning module's entities: thus, it allows users to abstract many implementation details, writing more readable and concise problems.

Finally, the domain module provides our implementation of the Block World domain and Logistics domain as defined respectively in Section 2.2.1 and Section 2.2.2. Namely, we propose implementations compliant with the abstraction defined in the planning module.

In the following sections, we propose a reification into Object Oriented software of the entities delineated in the previous chapters, along with the presentation of new interfaces and methods to handle some core software aspects.

### 4.1.1 Planning module

This section presents our proposal for the reification of the planning's formal model defined in Section 2.1.1 examining its core blocks with an insight into some design choices.

#### Building blocks

In the following lines, we provide a concise description of the central elements of the module exhibited in fig. 4.2.

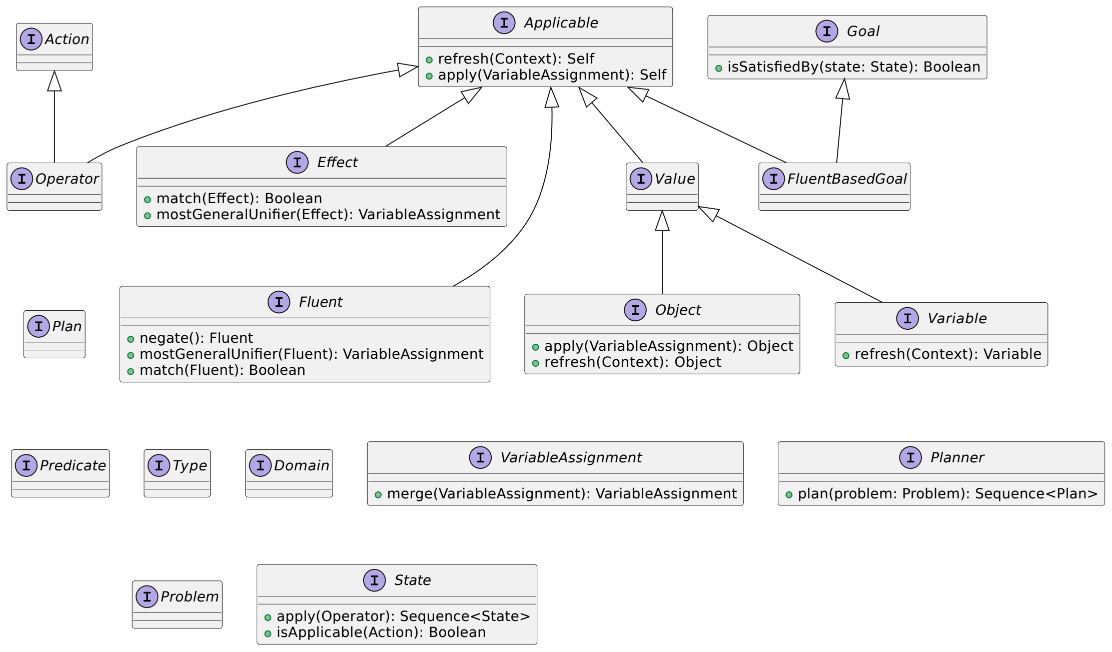
Action
Applicable
+ refresh(Context): Self
+ apply(VariableAssignment): Self
Goal
+ isSatisfiedBy(state: State): Boolean
Operator
Effect
+ match(Effect): Boolean
+ mostGeneralUnifier(Effect): VariableAssignment
Value
FluentBasedGoal
Plan
Fluent
+ negate(): Fluent
+ mostGeneralUnifier(Fluent): VariableAssignment
+ match(Fluent): Boolean
Object
+ apply(VariableAssignment): Object
+ refresh(Context): Object
Variable
+ refresh(Context): Variable
Predicate
Type
Domain
VariableAssignment
+ merge(VariableAssignment): VariableAssignment
Planner
+ plan(problem: Problem): Sequence<Plan>
Problem
State
+ apply(Operator): Sequence<State>
+ isApplicable(Action): Boolean

Figure 4.2: UML diagram including the core entities of the Planning module.

**Value:** is a formal reification for entity *Value* presented in Section 2.1.1. It is the top for hierarchy illustrated in fig. 4.3 composed by: *variable* or *object*, which we examine in the following paragraphs.

**Variable:** is a software implementation for the abstraction *Variable* shown in Section 2.1.1; thus, an entity that wraps a logic variable. A variable 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. We can instantiate a variable leveraging the method

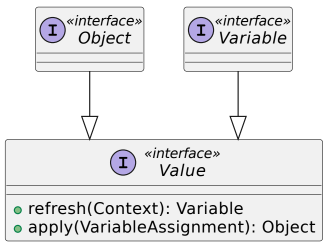
«interface» Object
«interface» Variable
«interface» Value
+ refresh(Context): Variable
+ apply(VariableAssignment): Object

Figure 4.3: UML diagram showing the hierarchy of *Value*.

`apply()`, which performs a logic substitution of the variable.

**Object:** is a reification for the entity *Object* presented in Section 2.1.1. Objects are *constants*; each object name must be unique and should be typed. If not typed, they will typically take on the properties of the base type object [27].

**ObjectSet:** is an entity not present in the formal model; it aims to map a set of objects to their type.

**Type:** is the software representation of the abstraction *Type* presented in Section 2.1.1. Types are values that an object can assume, they determine the set of allowed operations on each object. Types entities let users define parent-to-child relationships by allowing a *supertype* definition.

**Fluent:** is the reification of formal entity *Fluent* shown in Section 2.1.1. It is a ground logic fact which is *true*, or *false* at a given moment.

**Predicate:** is the software implementation of the abstraction *Predicate* presented in Section 2.1.1. Predicates are signatures for the fluents allowed in the problem the user wants to model.

**Goal:** is the reification of the abstraction Goal presented in Section 2.1.1; we devise it as an interface aimed at stating if the state satisfies a goal.

**FluentBasedGoal:** this interface is a specialized version of Goal. To this extent, a FluentBasedGoal is a conjunction of fluents(ground, or not). Given that FluentBasedGoals are a conjunction of fluents, we can see them as predicates on a state; thus, a condition one can test as it returns a boolean result. Consequently, the satisfiability of a FluentBasedGoal concerns the computation of the sub-set relationship among states, and the FluentBasedGoal as the conjunction of fluents that compose the FluentBasedGoals must all be in a state to consider the goal as fulfilled.

**Variable Assignment:** it the software implementation of the formal entity *Substitution* presented in Section 2.1.1 is a general type for logic substitutions; a Variable Assignment is a map used to assign values.

**State:** is the reification of the abstraction *State* shown in Section 2.1.1. A state is a conjunction of *ground fluents* which one can manipulate by logical inference.

**Effect:** is an abstraction that reify the *Post-conditions* presented Section 2.1.1. The purpose of this concept is to bundle the operations these elements must perform, notably as advanced in Section 2.1.1, an action must be able to provide a way for a user to check if it is applicable in a given state of the computation. We can prove that by leveraging the unification methods:

- `match()` that states if any unification among two fluents is possible,
- `mostGeneralUnifier()` that returns the most general unifier between any two fluents.

We show the structure of this entity in fig. 4.4.

**Action:** is the software implementation for the formal abstraction of *Action* presented in Section 2.1.1. Actions are ways of changing the state of the world. An action description consists of two main parts: a description of the effects of the action and the condition under which it is applicable.

As with any system for action description, our proposal needs to deal with the *frame problem*[1] 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 [62]. As illustrated by figure fig. 4.4, an action is a significant set of parameters; thus, an action is represented by:

- `name`: a consistent name to capture the purpose of the action within the domain;
- `effects`: a list of effects which can be either positive or negative; more specifically:
  - the `negativeEffects` represent all the conditions that became false after the application of the action to the state;
  - the `positiveEffects` on the contrary, represent all the conditions that turn true after the application of the action to the state;

**Precondition:** the preconditions of action are conjunctions of literals that determine its applicability at the current state of the world.

[1]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 [55].

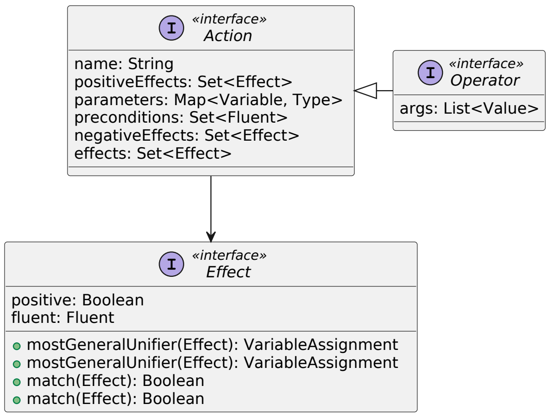
«interface»
Action
name: String
positiveEffects: Set<Effect>
parameters: Map<Variable, Type>
preconditions: Set<Fluent>
negativeEffects: Set<Effect>
effects: Set<Effect>
«interface»
Operator
args: List<Value>
«interface»
Effect
positive: Boolean
fluent: Fluent
mostGeneralUnifier(Effect): VariableAssignment
mostGeneralUnifier(Effect): VariableAssignment
match(Effect): Boolean
match(Effect): Boolean

Figure 4.4: UML diagram showing the hierarchy of *Action*.

**Post-conditions:** the post-conditions of action are conjunctions of literals that define the action's effects on the world. The effects can be either positive or negative.

**Operator:** is a ground action, or in other words, an action that is fully instantiated and has no variables among its terms.

**Applicable:** is one of the abstractions which is not formally represented in the formal model presented in Section 2.1.1. This element provides a signature for all the entities that must perform a logic substitution. The applicable hierarchy is illustrated in fig. 4.5. From the figure we can notice that the interface provides two methods:

- `apply()`,
- `refresh()`

Both of them are required within the logic paradigm to handle correctly a logic substitution. The first, `apply()`, is responsible for the implementation of the logic substitution, whereas `refresh()` is a utility method leveraged to handle correctly the variable names within the logic paradigm. The details of this latter method will be discussed in Chapter 5.

**Domain:** is the software implementation of the entity *Domain* shown in Section 2.1.1. A Domain synthesises the pivotal aspects of a specific domain, i.e. predicates and actions allowed as shown in fig. 4.6.

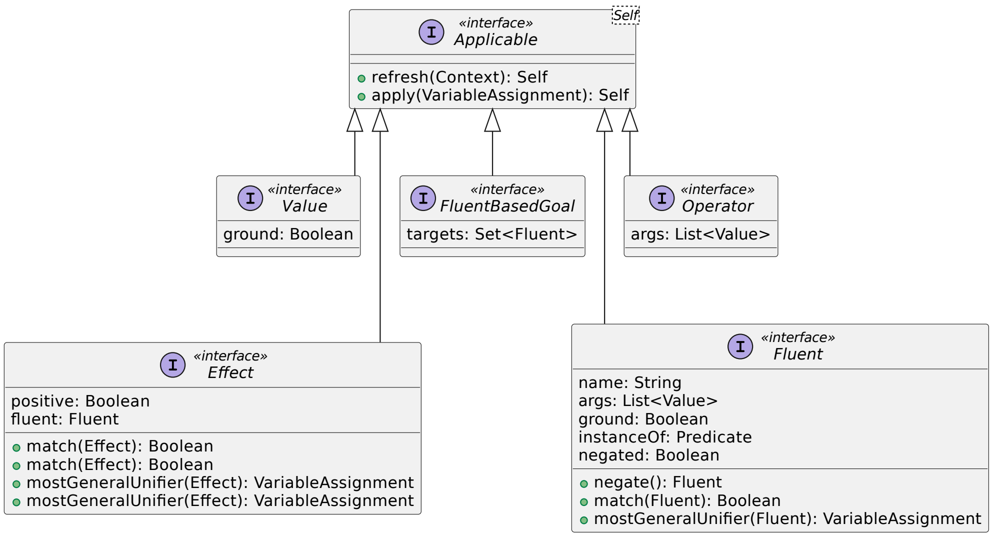

Figure 4.5: UML diagram showing the hierarchy of *Applicable*.

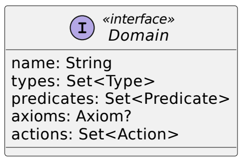

Figure 4.6: UML diagram showing the *Domain* entity.

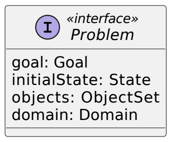

Figure 4.7: UML diagram showing the *Problem* entity.

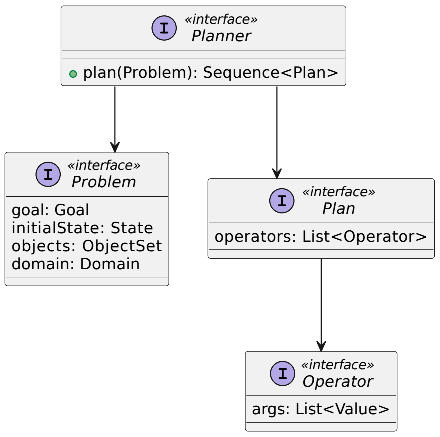

Figure 4.8: UML diagram showing the dependencies within the *Planner* entity.

**Problem:** is the reification of the *Problem* presented in Section 2.1.1. A problem wraps the critical constraints (i.e. goal, initial state, object allowed, etc.), about the underlying problem for a plan to a generated plan to be effective for execution as shown in fig. 4.7.

**Plan:** is a software implementation for the *Plan* abstraction presented in Section 2.1.1. A 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 devise a sequence of operators that leads from the initial state to goal one.

**Planner:** is the reification for entity *Planner* shown in Section 2.1.1 is the entity that encompasses the resolution strategy that given a problem produces a plan to accomplish its goal. A representation of this entity is in fig. 4.8, from this we can understand that a planner is an entity which exploits a `Problem` to devise a `Plan`. Accordingly, this concept only needs one method: `plan()` which encapsulates the resolution strategy for the problem. Thus, within the scope of the project, a planner is essentially any entity that given a problem can devise a plan.

#### DSL

Given the structural complexity of the components of the planning system, we decide to design a DSL to enhance its usage.

Accordingly, we envisage the DSL as a software layer that introduces a level of abstraction aimed at hiding behind the scenes the complexity of the planning system. More specifically, we devise this component to provide a high-level language to define a planning problem, and consequently all its components, to this extent our DSL must be fully interoperable and integrated with the planning system meaning that all the features of the planning system can be exploited from within our DSL.

Essentially, this domain-specific language is a straightforward way to write the planning entities avoiding repetition and improving their understandability, readability and reuse. The DSL is a pivotal component of the proposal as it fosters its usage providing a way to write problems more conveniently.

### 4.1.2 Explanation module

In the next paragraphs, we introduce the core elements of the explanations. This second part is the core of our proposal as it defines the components used to provide the explanations. As for the planning system, these elements are an implementation for the formal model designed in Chapter 3. A UML representation of the module's core elements is in fig. 4.9, as one can see, the system is defined by a minimal set of components, that can satisfy a variety of use cases.

To improve the readability of the diagram we decide not to explicitly represent usage dependencies from the planning system.

#### Building blocks

In this paragraph, we give a comprehensive overview of the main components of the explanation module.

**Question:** it is the interface that reify the *Question* abstraction proposed in Section 3.1.1. Thus, it is an entity that represents the demand the user wants to ask the system. This entity is meant to bundle all the questions presented in table 3.1. To this extent, it is the top of the hierarchy that is shown in fig. 4.10. More specifically, the figure shows the following class:

- `QuestionRemoveOperator` that is software implementation of *Question 1*, defined in Section 3.1.1,
- `QuestionAddOperator` that is reification of *Question 2*, proposed in Section 3.1.1,

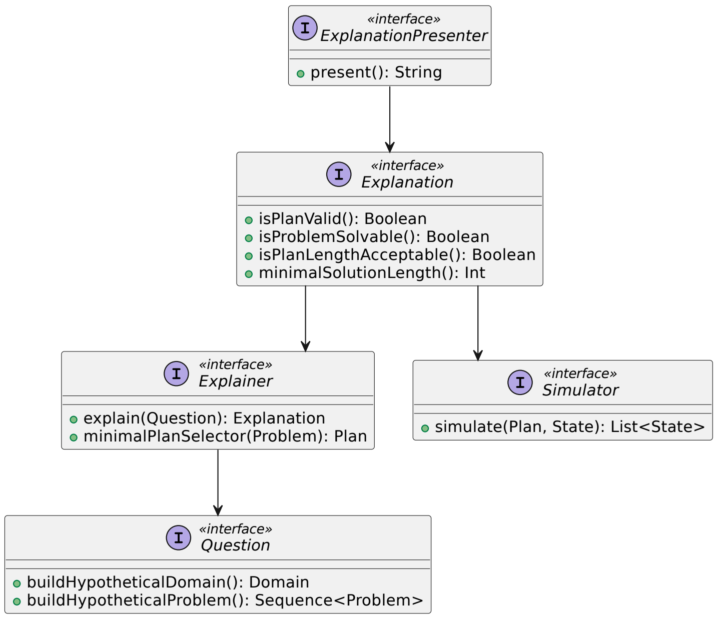
«interface» ExplanationPresenter
present(): String
«interface» Explanation
isPlanValid(): Boolean
isProblemSolvable(): Boolean
isPlanLengthAcceptable(): Boolean
minimalSolutionLength(): Int
«interface» Explainer
explain(Question): Explanation
minimalPlanSelector(Problem): Plan
«interface» Simulator
simulate(Plan, State): List<State>
«interface» Question
buildHypotheticalDomain(): Domain
buildHypotheticalProblem(): Sequence<Problem>

Figure 4.9: UML diagram including the core entities of the Explanation module.

- `QuestionReplaceOperator` that represents the abstraction *Question 3* shown in Section 3.1.1,
- `QuestionPlanProposal` that is software implementation of *Question 1*, defined in Section 3.1.1,
- `QuestionPlanSasfiability` that reify the concept of *Question 5* presented in Section 3.1.1.

From the fig. 4.10, we can see that we bundle the elements leveraged from all the questions in a base class, whereas each sub-class has its sub-set of parameters needed by the user to formulate that specific query. These parameters are indeed the constraints, to add to the model of the problem to reconcile it with the user model. At the top of the hierarchy we also define two methods:

- `buildHypotheticalDomain()`,
- `buildHypotheticalProblem()`

which are responsible for the compilation process formally defined in Section 3.2. Along these lines, we conceive the compilation process as an internal operation executed in two steps, one performed by calling the first method, `buildHypotheticalDomain()`, that is in charge of the creation of

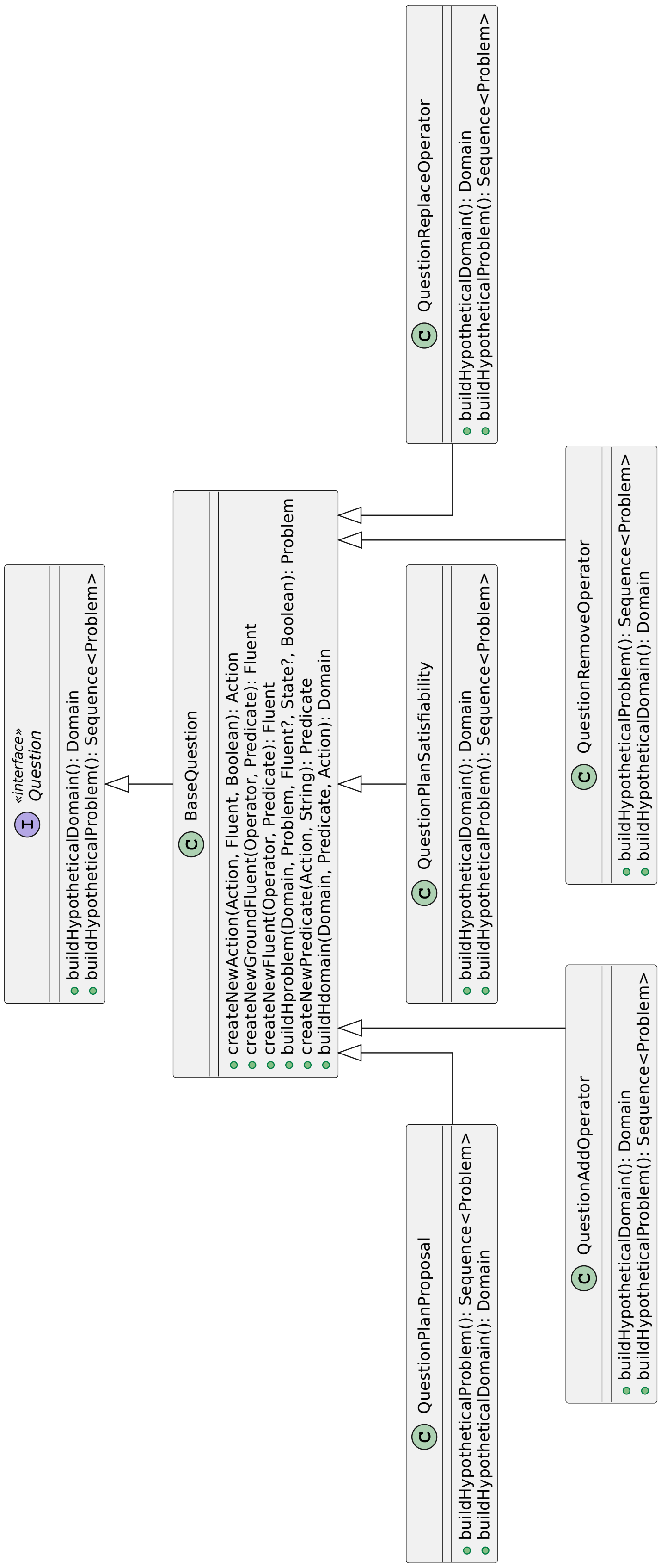
I «interface» Question
buildHypotheticalDomain(): Domain
buildHypotheticalProblem(): Sequence<Problem>
C BaseQuestion
createNewAction(Action, Fluent, Boolean): Action
createNewGroundFluent(Operator, Predicate): Fluent
createNewFluent(Operator, Predicate): Fluent
buildHproblem(Domain, Problem, Fluent?, State?, Boolean): Problem
createNewPredicate(Action, String): Predicate
buildHdomain(Domain, Predicate, Action): Domain
C QuestionReplaceOperator
buildHypotheticalDomain(): Domain
buildHypotheticalProblem(): Sequence<Problem>
C QuestionRemoveOperator
buildHypotheticalProblem(): Sequence<Problem>
buildHypotheticalDomain(): Domain
C QuestionPlanSatisfiability
buildHypotheticalDomain(): Domain
buildHypotheticalProblem(): Sequence<Problem>
C QuestionAddOperator
buildHypotheticalDomain(): Domain
buildHypotheticalProblem(): Sequence<Problem>
C QuestionPlanProposal
buildHypotheticalProblem(): Sequence<Problem>
buildHypotheticalDomain(): Domain

Figure 4.10: UML diagram showing the hierarchy of *Questions*.

a new domain containing the constraints required by the user and the other using the second, `buildHypotheticalProblem()`, that is responsible for the generation of a new problem according to the user suggestions. More specifically, reifying the model proposed in Section 3.2, we devise each Question to implement the above-mentioned methods differ according to its type.

**Explanation:** is element that reify the abstraction of *Explanation* presented in Section 3.1.2. The *Explanation* is the second pivotal entity of the explanation module, it encapsulates the answer to the question posed by the user. This entity comprises both the general and the contrastive explanation. Indeed despite the formal model presented in Section 3.1.2 the explanation has not devised a hierarchy but as a single abstraction. To this extent, we evaluate that from a software implementation point of view there was no need to decouple the concepts of *General Explanation* and the *Contrastive Explanation*. Consequently, we delegate to the `Explainer` and the `ExplanationPresenter` the task of modelling the different types of explanation whenever a user asks for their presentation. Thus, these two elements are crucial to keep untangle the model of the explanation, from its actual implementation and presentation.

**Explainer:** the interface that reify the entity *Explainer* proposed in 3.1.3. The *Explainer* is the element responsible for the communication between the lay user and the explanation system. To this extent, the main purpose of the explainer is to provide a way for a user to express a question to submit to the explanation system along with a planner. Indeed, aiming to envisage a planner-agnostic software proposal for XAIP, the explainer must also require the user to provide the planner that will be used within the Explanation entity to design the new solutions and explanation for the problem.

**ExplanationPresenter:** is the component used to present an explanation in a user-friendly format. Indeed, this element is meant to uncouple the design of the explanation from its presentation. Thus, it provides a convenient way to display general and contrastive explanations.

**Simulator** : is the element leveraged by the Explanation to determine if the updated model or the plan proposed by the user provides a valid solution for the starting problem. Therefore this component must be able to simulate the execution of a given plan leveraging a convenient way to understand if it is an acceptable solution for a problem or not. That is a pivotal element for the explanation system as it provides an opportune way to test the user's proposals for new plan constraints as well as alternative plans.

## 4.2 Interaction

The central contribution of this thesis is the design of a model for XAIP and the implementation of a software proposal compliant with it; thus, our proposal is mainly composed of structural entities. Because of the lack of active entities, such as *agents*, we can reduce the behaviour of the system to the major components of the planning and explanation systems. More specifically, the project consists of a software library that has no active entity; therefore, there is no active interaction among the components. Once the user defines the domain, the problem, or the question (whether using the planner, the DSL or the explanation system), the library passively utilizes these latter, and the elements therein contained to compute the plans to solve the problem, if they exist, or the explanation the user required. However, from an interaction perspective, one can distinguish at least two active entities for the planner and the explanation system.

### 4.2.1 Planning system

This section analyses the main concepts involved when a user wants to create a planning problem and solve it. To this extent, we do not focus on single elements of the planning module used for the elaboration, but we give a high-level overview of the process. For the planning module one can find three (or four in case one is using the DSL) entities:

- the **user**, who wants to utilize the planning system to define the domain for a problem and later the problem itself,
- the **planning system** who is responsible for the generations of the two elements required by the user,
- the **planner** who is asked by the planning system to generate the plans to solve a given problem and,
- optionally the **DSL**, in case one decided to exploit it. The DSL is responsible for converting the call made by the user behind the scenes into terms understandable for the planning system and calling it to solve the query required by the user.

In fig. 4.11 is shown the sequence diagram for the planning system, whereas the fig. 4.12 illustrates the sequence diagram for leveraging the DSL.

The figures show that we divide the process into two pivotal phases: the system's initialization and the planning phase.

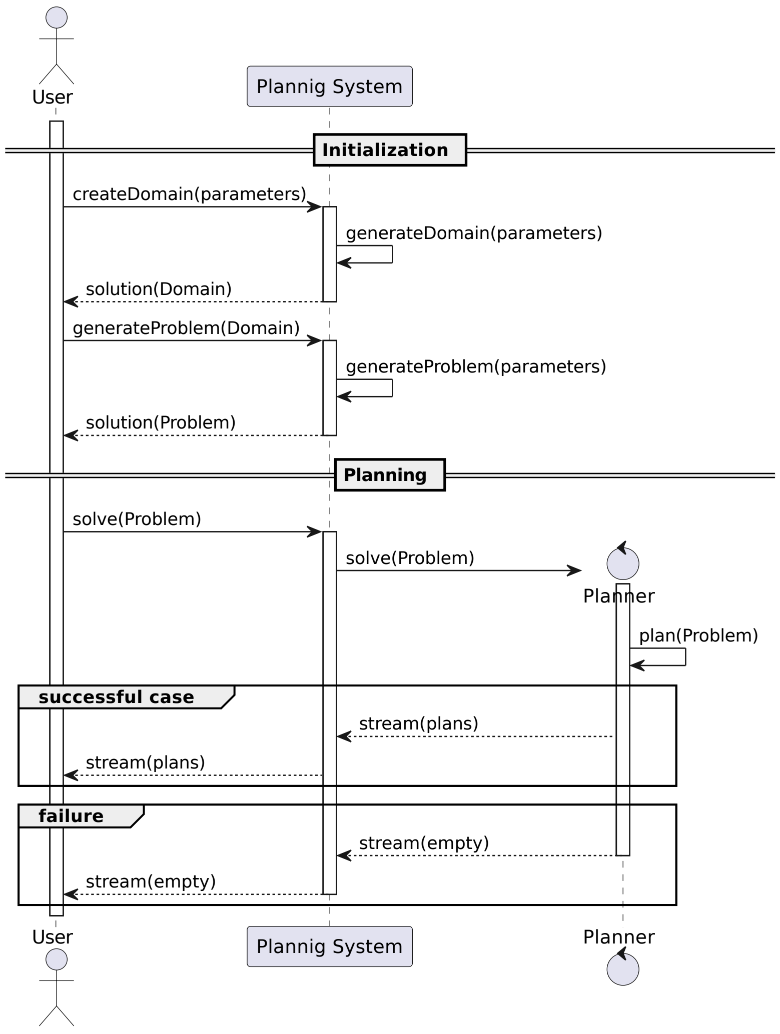

Figure 4.11: Sequence diagram showing the plan generation without the usage of the DSL module.

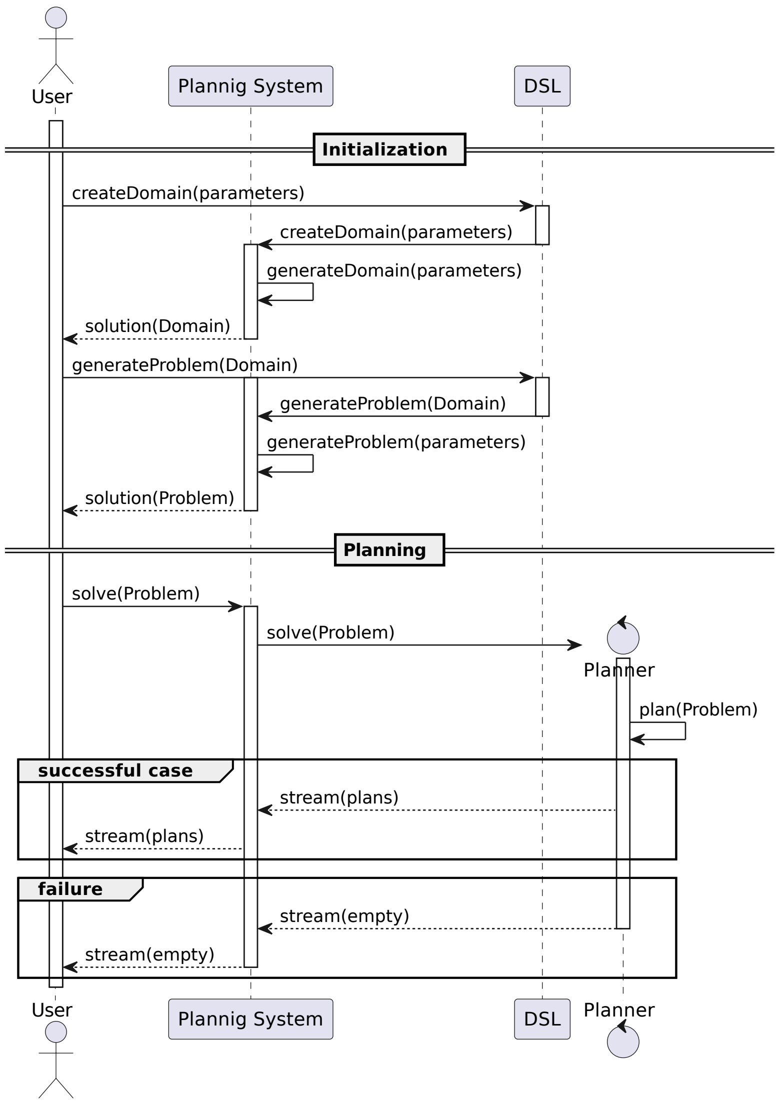

Figure 4.12: Sequence diagram showing the plan generation leveraging the DSL module.

**Initialization phase:** it is the first phase concerning the user-system interaction when a user aims to devise a plan to solve a given problem. This phase is itself divided into two consequent phases. First, the planning system employs the parameters received from the user to build a domain model complaint with them. Next, it exploits that new domain along with the parameters provided by the user to create a new problem for the given domain.

**Planning phase:** the planning phase is the core of the computation; it takes place whenever a user requires the planning system to solve a given problem. At this point, the planning system leverage an internal planner to compute the solutions for the problem and return them to the user. In this context, it is relevant to underline that the problem provided by the user could not be solvable; in this case, the planner would not be able to devise any plan and would return an empty solution.

From the figures, one can notice that the computation for the generation of the domain and the problem is almost identical in leveraging or not the DSL module because the DSL only introduces a small software layer built on top of the planning module. However, this layer is essential to provide users with a compact means to instantiate and use planning entities.

### 4.2.2 Explanation system

This section examines how the user can leverage the explanation system to obtain an explanation. As in Section 4.2.1, we will not focus on the single elements of the system; however, we provide a concise overview of the main concepts involved in the interaction.

From the interaction perspective, the main entities involved when an explainee asks for an explanation are:

- the **explainee**, thus a user that wants to inquire about the system posing its questions,
- the **explanation system** that is responsible for the elaboration of the explanation,
- the **planning system** is exploited behind the scenes during the compilation process to generate the *hypothetical problem* and optionally the *hypothetical domain*.

Figure 4.13 shows a general interaction between an explainee and our system proposal when the explainee demands either of the following types of questions: $\Theta_1$, $\Theta_2$, or $\Theta_3$.

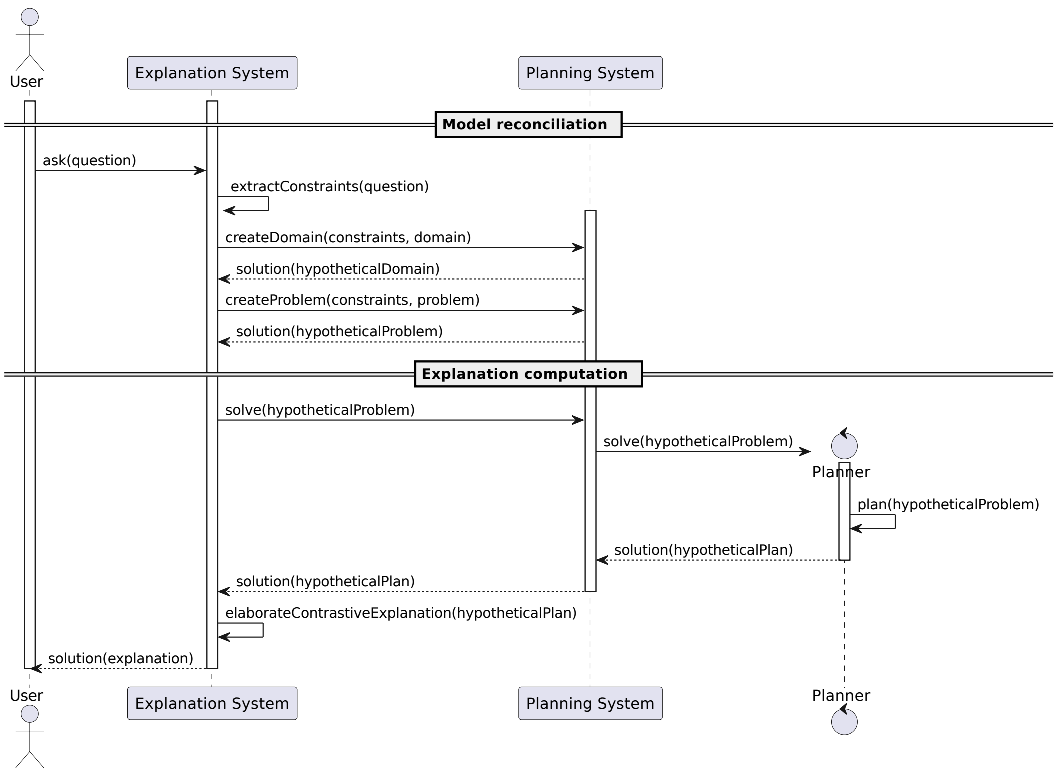
User
Explanation System
Planning System
Model reconciliation
ask(question)
extractConstraints(question)
createDomain(constraints, domain)
solution(hypotheticalDomain)
createProblem(constraints, problem)
solution(hypotheticalProblem)
Explanation computation
solve(hypotheticalProblem)
solve(hypotheticalProblem)
Planner
plan(hypotheticalProblem)
solution(hypotheticalPlan)
solution(hypotheticalPlan)
elaborateContrastiveExplanation(hypotheticalPlan)
solution(explanation)
User
Explanation System
Planning System
Planner

Figure 4.13: Diagram describing the user-system interaction in case the user demands the system an explanation for $\Theta_1$, $\Theta_2$, or $\Theta_3$.

From the figure, one can see that we leverage two crucial steps to elaborate the explanation for those types of questions:

- the model reconciliation (described in Section 3.1.2),
- the explanation generation.

Within this framework, we can see that when there is a mismatch between the expectation of the explainee and the plan proposed by an automatic solver, the explainee may start an interaction with the system asking it for an explanation. To this extent, the explainee formulates a question for the explanation system, where the explainee makes some suggestions to make the planner model adhere to the explainee mental one.

When the explanation system receives a question from an explainee performs two major activities: the model reconciliation and the explanation generation.

**Model reconciliation:** the explanation system is firstly responsible for the reconciliation process. The reconciliation process has two pivotal steps: the extraction of user suggestions and the generation of the model. More specifically, firstly, the explanation system retrieves the suggestions of the explainee from the question and translates them into new constraints to add to the problem model. Consequently, it exploits the planning system to elaborate a new domain and problem complaint with user suggestions. The result of this second step is the *Hypothetical Domain* and *Hypothetical Problem* examined in Section 3.2; the generation of these two elements terminates the compilation process.

**Explanation generation:** after devising the hypothetical domain and problem, the planning system computes the hypothetical plan and returns it to the explanation system to build the explanation to present to the user.

Otherwise, if the user requires a question of the type: $\Theta_4$ or $\Theta_5$, the explanation system will not need any phase of model reconciliation. Indeed, as shown fig. 4.14, in this case, the user only requires the system to have insight into the properties of the plan, or compare two solutions; therefore, there would be no need for it to generate different a different model.

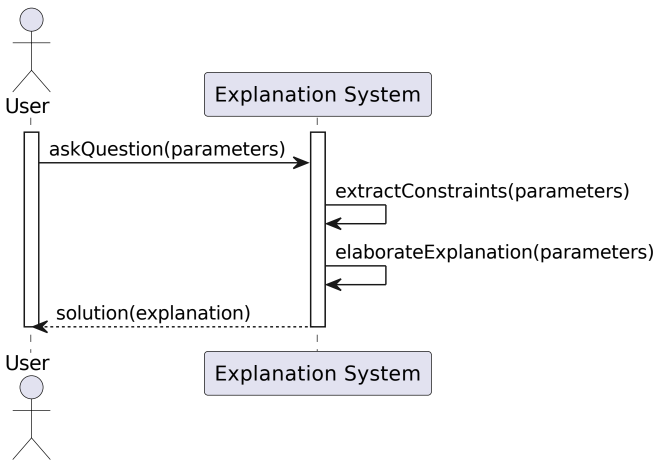

Figure 4.14: Diagram describing the user-system interaction in case the user has an explanation for $\Theta_4$ and $\Theta_5$.

# Chapter 5

# Implementation and Validation

This chapter is about two crucial aspects of our proposal: its software implementation and validation. To this extent, we begin the chapter by presenting how we implement our software solution on top of the design details discussed in Chapter 4, focusing only on the most significant implementation choices. Afterwards, we propose an assessment of our proposal along with some validation metrics. More specifically, in Section 5.2.1 we describe how we test our implementation. Section 5.2.2 shows a brief hands-on demonstration of how we can leverage our proposal to inquire a planner about a solution proposed by an automatic solver and require it to add constraints to the initial model and generate an explanation for the novel plan. Finally, in the last section of this chapter, Section 5.2.3, we provide performance benchmarks of our XAIP library for a set of running examples.

It is worth mentioning that our software implementation is available on GitHub[1] as an open-source project under the Apache 2.0 licence.

## 5.1 Implementation

This section provides an in-depth tour of some significant implementation choices of the central modules of the proposal: the planning and explanation modules, besides discussing some details of the others.

**Abstraction reification:** we implement all the core types described in Section 4.1 as interfaces. Each interface designed has a companion object (*Singleton*) containing several *static factory* methods, named "of", that handle the actual creation of instances. These methods usually follow the same pattern: `<Type>.of(<args>)`, their purpose is to let the user decide which type of ar-

[1] `https://github.com/pikalab-unibo-students/xaip-lib`

guments to use for the object instantiation preventing the user to perform the conversion.

We reified almost every interface in a class, more specifically given that most of our entities were structural elements we implemented them as Kotlin `Data Class`. Accordingly, following Kotlin guidelines, we decide to implement all structural entities using that data structure to ensure *immutability* while avoiding *boilerplate* code. Immutability is a code design practice that guarantees several advantages that range from preventing memory waste on copies of the same instances to providing *thread-safe* objects. Within our software proposal, we ensure immutability by defining all the properties of the data classes as *read-only* variables. To this extent, immutability is a key aspect of our library. Indeed, as several parts of our software proposal share instances of the same elements, we exploit the opportunity to perform expensive computations only once. We then save their results to return them in every subsequent request.

**Self type support:** unlike other languages such as Scala or Java, Kotlin does not natively support *self types*, to overcome that limitation, and let inherited methods know the type of the class on which they are called we exploit the *Curiously Recurring Template Pattern*. Within this context, the *Curiously Recurring Template Pattern*(CRTP) is a C++ idiom in which a `class X` derives from a class template instantiation using `X` itself as template argument; so that the base class can know the derived type [39]. In the planning module, we exploit the CRTP to implement the `Applicable` interface. As outlined in the Chapter 4, `Applicable` represents a signature for each entity that should perform a substitution and defines two methods `apply()` and `refresh()`, both of which can only perform their internal computation by knowing the base class from which the methods are called.

**Laziness:** to minimize the number of "heavy operations" as well as the waste of memory we leverage two significant Kotlin strategies: *lazy initialization* and *lazy evaluation*.

**Lazy initialization:** it is widely known that class initialization can be a "heavy process" that in the worst cases can result in delays for the whole application. To overcome this issue, we make extensive use of the lazy initialization methods of Kotlin. Kotlin provides two keywords to implement lazy initialization, `lateinit` and `by lazy`, and we make use of both of them within our library. Particularly in the explanation module, we make relevant use of the second keyword, defining `by lazy` a large set of fields. To this extent, the usage of that keyword allows the system to initialize each field only if and when it is required. This saves memory and prevents overloading the class creation process with initializations that may be delayed.

**Lazy evaluation:** is an evaluation strategy that postpones the evaluation of an expression until its value is needed. This strategy is particularly well suited to all those elements that do not need repeated evaluation or must handle potentially infinite data structures. Kotlin provides a convenient data type to evaluate expressions only if and when they become necessary at runtime: `Sequences`. We leverage Kotlin `Sequences` in all those entities that handle nondeterministic scenarios. We rely on `Sequences` instead of other data structures eagerly evaluated for efficiency and performance. This is because lazy evaluation allows us to avoid computing all the possible solutions for a given scenario but only a given subset of them. Avoiding possible long computations is a central requirement for those components that must work with large state spaces; a blatant example is the `Planner`. Thus, in general, a planner may devise numerous plans that satisfy a goal for a problem; however, the user hardly ever wants to see all of them. For this reason, it is usually unnecessary for the planner to perform such long computations.

**Variable refreshing:** is a basic mechanism that 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 [16]. That aspect is a central issue to be considered for the project, as without proper variable refreshing, *spurious substitution* could happen. To avoid that issue all the entities can perform a logic substitution; thus, all the entities inherited from `Applicable` provide a `refresh()` method which takes an instance of the `context` as a parameter to perform the refreshing operation. Knowing the `context` is essential to avoid renaming variables that belong to the same environment.

## 5.2 Validation

### 5.2.1 Testing

It is widely agreed that testing is a crucial activity in software engineering. Thus, in this section, we examine the test suite developed to validate our software proposal. We organize our project into modules, each with its own testing suite. Accordingly, each testing suite includes a set of automated tests to corroborate the compliance of the module classes with the awaited behaviour and eavesdrop on possible blunders. We designed three types of tests:

**Unit tests:** the central goal of this kind of test is to verify the behaviour of single functionalities of one class (i.e. single methods or even single lines of code), thus they require detailed knowledge of the internal program design and code.

**Integration tests:** these tests aim to examine how a limited number of parts of the system work correctly together.

**System tests:** are the pivotal part of the testing suite they focus on validating the whole system behaviour and attempting to reproduce the final user usage of the proposal.

All the test cases are implemented with *Kotest*[2] a multi-platform testing framework for Kotlin.

### 5.2.2 Proof of Concept Demonstration

This section provides some usage examples of our software proposal. For the demonstration, we leveraged two problems, one from the Block World and one from the Logistics domain; demonstrating how to create problems and perform the reconciliation process to determine explanations for plans. Within our examples, we assume the user already has a plan from an automatic planner that the user doesn't fully understand, and as a result, wants to get clarifications about it.

In general, we can summarize the core steps the user must perform to get an explanation as follows:

- to begin, the user defines a model for a problem and a domain;
- the user then instantiates an appropriate question specifying its parameters; commonly: the problem and the domain defined in the previous step, the plan retrieved from the automatic planner and the user's suggestions;
- consequently, the user must instantiate an `Explainer`; thus, the component responsible for building an explanation from a given question and an `ExplanationPresenter`, which is the entity entitled to show the explanation in a user-friendly way.

#### Logistics

The first end-to-end example we examine consists of a simple transportation problem within the Logistics domain. In this context, we assume a user wants to move

[2]`https://kotest.io/`

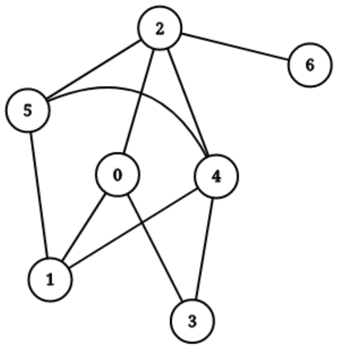

Figure 5.1: Graph representing the configuration of the locations of the Logistics problem.

some objects that we call containers in a given location and wants to exploit an agent, which we call a robot, to do so.

As mentioned before, to begin, the user must define a model for the problem. The user can do so by leveraging the DSL, as shown in listing 5.1 and listing 5.2. The model defined reifies the description proposed in Section 2.2.2; within this context, we call:

- `r`: the robot agent;
- `c1` and `c2`: the two containers;
- `l1`, . . . , `l7`: the locations. Particularly, we arrange the locations as a graph shown in fig. 5.1.

Thus, the listing 5.1 models an agent, two objects and a set of positions within which the agent can move to displace the objects. Listing 5.2 presents a Logistics problem in which the user aims the robot both to move the containers from their former locations to some new ones and eventually for the robot to reach a new site. Within this framework, to solve the problem an automatic planner may propose the plan shown in listing 5.3.

However, a user may have a different idea in mind; the user may want to know why the robot did not go to `l6` to arrive at `l5`. The user may ask that question for several reasons; for example, the user may know something that is not modelled in the problem, for instance, that the route from `l4` to `l5` is momentary, not accessible for some reason or that the path from `l6` is more rapid. Whatsoever the reason, the user may want to force the system to check if there is a feasible solution for the robot to go to `l6` instead of `l4`.

To require the system to do so, the user should define an appropriate question, `QuestionReplaceOperator`, instantiating it with the proper parameters, as shown

Listing 5.1: Snippet showing the Logistics domain.

```
val domain = domain {
        name = "logistics_world"
        types {
            +"anything"
            +"strings"("anything")
            +"locations"("strings")
            +"robots"("strings")
            +"containers"("strings")
        }
        predicates {
            +"connected"("locations", "locations")
            +"atLocation"("robots", "locations")
            +"loaded"("robots", "containers")
            +"unloaded"("robots")
            +"inContainerLocation"("containers", "robots")
        }
        actions {
            "move" {
                parameters {
                    "X" ofType "robots"
                    "Y" ofType "locations"
                    "Z" ofType "locations"
                }
                preconditions {
                    +"connected"("Y", "Z")
                    +"atLocation"("X", "Y")
                }
                effects {
                    +"atLocation"("X", "Z")
                    -"atLocation"("X", "Y")
                }
            }
            "load" {
                parameters {
                    "Z" ofType "locations"
                    "Y" ofType "containers"
                    "X" ofType "robots"
                }
                preconditions {
                    +"atLocation"("X", "Z")
                    +"inContainerLocation"("Y", "Z")
                }
                effects {
                    +"loaded"("X", "Y")
                    -"inContainerLocation"("Y", "Z")
                }
            }
            "unload" {
                parameters {
                    "Z" ofType "locations"
                    "Y" ofType "containers"
                    "X" ofType "robots"
                }
                preconditions {
                    +"atLocation"("X", "Z")
                    +"loaded"("X", "Y")
                }
                effects {
                    +"inContainerLocation"("Y", "Z")
                    -"loaded"("X", "Y")
                }
            }
        }
    }
```

Listing 5.2: Snippet showing the Logistics problem.

```
val problem = problem(domain) {
        objects {
            +"robots"("r")
            +"locations"("l1", "l2", "l3", "l4", "l5", "l6", "l7")
            +"containers"("c1", "c2")
        }
        initialState {
            +"atLocation"("r", "l1")
            +"inContainerLocation"("c1", "l2")
            +"inContainerLocation"("c2", "l3")
            +"connected"("l1", "l2")
            +"connected"("l1", "l3")
            +"connected"("l2", "l4")
            +"connected"("l3", "l4")
            +"connected"("l4", "l5")
            +"connected"("l1", "l6")
            +"connected"("l5", "l6")
            +"connected"("l5", "l7")
            +"connected"("l1", "l5")
            +"connected"("l2", "l1")
            +"connected"("l3", "l1")
            +"connected"("l4", "l2")
            +"connected"("l4", "l3")
            +"connected"("l5", "l4")
            +"connected"("l6", "l2")
            +"connected"("l6", "l5")
            +"connected"("l7", "l5")
            +"connected"("l5", "l1")
        }
        goals {
            +"atLocation"("r", "l5")
            +"inContainerLocation"("c1", "l4")
            +"inContainerLocation"("c2", "l1")
        }
    }
```

Listing 5.3: Snippet showing the plan proposed by the automatic planner to solve the problem defined in listing 5.2.

```
val formerPlan = Plan.of(
            listOf(
                moveRfromL1toL3,
                loadC2fromL3onR,
                moveRfromL3toL1,
                unloadC2fromRtoL1,
                moveRfromL1toL2,
                loadC1fromL2onR,
                moveRfromL2toL4,
                unloadC1fromRtoL4,
                moveRfromL4toL5
            )
        )
```

Listing 5.4: Snippet showing the question of type $\Theta_1$ for the problem defined in listing 5.2.

```
val question = QuestionReplaceOperator(
            problem,
            formerPlan,
            moveRfromL4toL6,
            8,
            alternativeState
        )
```

Listing 5.5: Snippet showing an instance of the `Explainer`.

```
val explainer = Explainer.of(Planner.strips())

val explanation = explainer.explain(question)
```

in listing 5.4. Thus, specifying the problem to analyze, the plan proposed by the automatic planner, the operator that the user wants to add to the plan, the position it should take within it and the state in which the replacement should take place.

At that point, the user should create an `Explainer` and initialize it with a `Planner` as shown in listing 5.5. Therefore, the user should instantiate an appropriate `ExplanationPresent` according to the type of explanation the user would like to receive. In this case, we assume the user would like a general explanation; to this extent, the user can create a `BaseExplanationPresenter` and require it to display the explanation (listing 5.6). The explanation demonstrates that it is possible to reach `l5` from `l6`. However, one can notice that a general explanation can be quite verbose whereas a user may want a more concise answer. In such cases, the user can ask the system for a simplified version of the general explanation by calling `presentMinimalExplanation()`. Listing 5.8 shows the output of this second call. The two explanations provide a similar insight into the property of the solution proposed, but the second one presents it in a more concise way. To this extent, we design "minimal explanations" for proficient users who may not be interested in a detailed explanation in natural language.

Listing 5.6: Snippet showing an instance of the `BaseExplanationPresenter`.

```
ExplanationPresenter.of(explation).present()
```

Listing 5.7: Snippet showing a general explanation for the question proposed in listing 5.4.

```
The problem [atLocation(r, l5), inContainerLocation(c1, l4), inContainerLocation(
    c2, l1)] is solvable.
The former plan was: [move(r, l1, l3), load(l3, c2, r), move(r, l3, l1),
    unload(l1, c2, r), move(r, l1, l2), load(l2, c1, r), move(r, l2, l4),
    unload(l4, c1, r), move(r, l4, l5)]
The novel plan is: [move(r, l1, l3), load(l3, c2, r), move(r, l3, l1),
    unload(l1, c2, r), move(r, l1, l2), load(l2, c1, r), move(r, l2, l4),
    unload(l4, c1, r), move(r, l4, l6), move(r, l6, l5)].
The novel plan is a valid solution to the problem.
The minimal solution is: [move(r, l1, l3), load(l3, c2, r), move(r, l3, l1),
    unload(l1, c2, r), move(r, l1, l2), load(l2, c1, r), move(r, l2, l4),
    unload(l4, c1, r), move(r, l4, l5)]
The plan is not the minimal solution.
There are 2 additional operators with respect to the minimal solution:
    [move(r, l4, l6), move(r, l6, l5)].
```

Listing 5.8: Snippet showing a "minimal explanation" for the question proposed in listing 5.4.

```
The plan: [move(r, l1, l3), load(l3, c2, r), move(r, l3, l1), unload(l1, c2, r),
    move(r, l1, l2), load(l2, c1, r), move(r, l2, l4), unload(l4, c1, r),
    move(r, l4, l6), move(r, l6, l5)], is valid: true
 The length is acceptable: true
 Operators missing: [move(r, l4, l5)]
 Additional operators: [move(r, l4, l6), move(r, l6, l5)]
```

Listing 5.9: Snippet showing the Block World problem.

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

### Block World

The second proof of concept demonstration is a manipulation problem from the Block World Domain. As advanced in Section 5.2.2, firstly the user defines the domain[3] followed by the problem (listing 5.9).

The model is a reification of the Block World formal model of Section 2.2.1; within this context, we call:

- `a, b, c, d`: the blocks;
- `arm`: the agent in charge of moving the blocks;

In this example, the goal of the problem is to change the initial configuration of the blocks by "unstacking" all of them from their initial arrangement and disposing of them in the goal configuration. Listing 5.10 shows the plan proposed by the planner; however, a user may have a different idea in mind; more specifically, the user may ask why the `arm` did not put down the block `c` before putting it on the block `a`.

The reason for an expert is trivial; it is a pointless operation; indeed, if the `arm` already holds the block `c` because it has already performed an "unstack" operation, there is no need to put it down and then pick it up to stack it on the

[3]Due to the length of the Block World domain definition, we do not explicitly model it in this document; we assume it is the same as in Section 2.2.1.

Listing 5.10: Snippet showing the plan solving the problem defined in listing 5.9 that cause the mismatch.

```
val initialPlan = Plan.of(
            listOf(
                unstackAB,
                putdownA,
                unstackCD,
                stackCA,
                pickD,
                stackDC,
                pickB,
                stackBD
            )
        )
```

Listing 5.11: Snippet showing a question of type $\Theta_2$ for the problem defined in listing 5.9

```
val question = QuestionAddOperator(
            problem,
            initialPlan,
            putdownC,
            3
        )
```

block `a`. However, a user not proficient in the Block Domain may not think about it. Therefore, the user may ask why the planner has not done the "putdown" operation before the "stack" one. To this extent, listing 5.11 illustrates how the user can ask the system such a question. Thus, the user only has to create a proper question, in this case, a `QuestionAddOperator`, specifying the operator to add to the plan and the position it should take.

Within this context, it is relevant to notice that the user is unaware of the compilation process that is triggered when asking the question. The result of this operation is neither an instance of the hypothetical domain nor the hypothetical problem but one of the required question. Consequently, the user can transparently create an `Explainer` providing the `Planner` to use (listing 5.5) for the generation of the explanation and then instantiate an `ExplanationPresenter` to visualize it conveniently as shown in listing 5.6.

This first use example simulates the case in which a user asks for a general explanation. The output of the explanation reveals that the plan devised following the user's suggestions does not hold true; that is because if the `arm` has laid down the block `a`, then it has to pick it up at some point as well. For this reason in our explanation (listing 5.12), we point out that the user's suggestions result in a plan that is invalid because an operator is missing. However, it may also happen that a

Listing 5.12: Snippet showing the general explanation for the question proposed in listing 5.11.

```
The problem [clear(b), on(b, d), on(d, c), on(c, a), at(a, floor)] is  solvable.
The former plan was: [unstack(a, b), putdown(a), unstack(c, d), stack(c, a), pick(
    d), stack(d, c), pick(b), stack(b, d)]
The novel plan is: [unstack(a, b), putdown(a), unstack(c, d), putdown(c), stack(c,
     a), pick(d), stack(d, c), pick(b),
    stack(b, d)].
The novel plan is not a valid solution for the problem.
The minimal solution is: [unstack(a, b), putdown(a), unstack(c, d), stack(c, a),
    pick(d), stack(d, c), pick(b), stack(b, d)]
The plan is not the minimal solution. There is 1 additional operator with respect
    to the minimal solution: [putdown(c)].
```

Listing 5.13: Snippet showing a question of type $\Theta_4$ for the problem defined in listing 5.9.

```
val question = QuestionPlanProposal(problem, formerPlan, planProposal)
```

user has a valid solution in mind and wants the system to analyze it. In this case, the user may ask the system a different question, namely a `QuestionPlanProsal`.

Within this framework, the user only has to instantiate the new question (listing 5.13), pass it to the former `Explainer` and require the `ExplanationPresenter` an explanation.

In this case, in addition to a general explanation, the user may also want a contrastive one. To do so, the user only has to create a new `ContrastiveExplanationPresenter` (listing 5.14) and ask for the two explanations.

Within this framework, it is relevant to notice that to obtain a contrastive explanation the user must leverage a `ContrastiveExplanationPresenter`, which besides providing a contrastive explanation, can also show general ones. Listing 5.15 shows the output for the general explanation. At first glance, it is similar to the previous case; however, it is worthwhile to notice that besides having additional operations the system classifies the revised plan proposed by the user as a valid solution to the problem.

On the other hand, a contrastive explanation provides a concise way to compare

Listing 5.14: Snippet showing an instance of the `ContrastiveExplanationPresenter`.

```
val presenter = ContrastiveExplanationPresenter.of(explanation)

presenter.presentContrastiveExplanation()
```

Listing 5.15: Snippet showing the general explanation to the question defined in listing 5.13.

```
The problem [clear(b), on(b, d), on(d, c), on(c, a), at(a, floor)] is  solvable.
The former plan was: [unstack(a, b), putdown(a), unstack(c, d),
    stack(c, a), pick(d), stack(d, c), pick(b), stack(b, d)]
The novel plan is: [unstack(a, b), putdown(a), unstack(c, d), putdown(c), pick(c),
     stack(c, a), pick(d), stack(d, c), pick(b), stack(b, d)].
The novel plan is a valid solution to the problem.
The minimal solution is: [unstack(a, b), putdown(a), unstack(c, d), stack(c, a),
    pick(d), stack(d, c), pick(b), stack(b, d)]
The plan is not the minimal solution. There are 2 additional operators with
    respect to the minimal solution: [putdown(c), pick(c)].
```

Listing 5.16: Snippet showing the contrastive explanation to the question defined in listing 5.13.

```
problem:
    [clear(b), on(b, d), on(d, c), on(c, a), at(a, floor)]
    is solvable.
    originalPlan: [unstack(a, b), putdown(a), unstack(c, d), stack(c, a), pick(d),
        stack(d, c), pick(b), stack(b, d)],
    novelPlan: [unstack(a, b), putdown(a), unstack(c, d), putdown(c), pick(c),
        stack(c, a), pick(d),
        stack(d, c), pick(b), stack(b, d)] valid: true,
    addedList=[putdown(c), pick(c)],
    deletedList=[],
    sharedList=[unstack(a, b), putdown(a), unstack(c, d), stack(c, a), pick(d),
        stack(d, c), pick(b), stack(b, d)]
```

two different solutions; thus, from listing 5.15, we can see that the only difference among the plans concerns the addition of two operators: `putdown(c), pick(c)` to the novel plan.

### 5.2.3 Performance Benchmarks

In this section, we provide some information regarding the performance and execution time of our proposal in generating general and contrastive explanations for the questions defined in table 3.1.

We chose a problem from each domain and a maximum length (50 operators) for the plans to be devised. Accordingly, we formulate a set of valid plans of different lengths using ad-hoc scripts. As a result, we generated over 5000 valid plans for the Block World and Logistics problems. We then frame a set of questions for each type of explanation. These questions enable us to evaluate the system's performance when required to add, remove, or replace operators in different positions on the plan, or to compare two different solutions. Specifically, for all the plans devised, we devise multiple questions for each type presented in table 3.1, measuring the efficiency of the system in terms of time and amount of memory needed to compute the explanations. To this extent, given the JVM sources of non-determinism[4], we test its variation by exploiting a secondary thread that continuously samples memory allocation when computing the explanation. We create two datasets (for contrastive and general explanations) containing over 8000 explanations for each question within each domain.

We tested the performance of our proposal on three different operating systems: Linux, macOS and Windows obtaining fairly similar results. Given the negligible differences among outcomes from the three operating systems in the following lines, we propose an in-depth analysis of the results from the Windows system.

### Windows OS

This paragraph examines the results of the benchmark analysis using a Windows-based operating system.

**Hardware specification:** we measure the benchmarks on a local machine with an Intel Core i7-1280P CPU with 14 cores and 20 threads, 24MB of L3 cache, and a clock frequency variable from 1.8GHz to 4,8GHz. The installed operating system is Windows 11 Home x64 version 22H2 and build 22621.232.

**Results:** for the experiment we chose to generate plans containing a maximum of 50 operators, devising over 5000 valid plans for each domain. These plans were exploited to produce a dataset of over 8000 explanations for the questions in each domain. This paragraph provides an analysis of the results obtained at a different level of aggregation; we begin examining some line charts showing the resources needed according to the plan length, subsequently, we review some aggregate results illustrating the average resource needed to compute each question in the two domains.

Figure 5.2 fig. 5.3 display the time and memory variation according to the length of the plan. Particularly, we divide the figures into two parts: on the left, we present the results of the computation of the general explanations whereas, on the right, we analyze the contrastive ones. To this extent, the first column of each part represents with a red line the average time required to compute an

[4]It is a well-known issue that measuring performance in managed runtime systems, such as Java, is particularly challenging [6, 29, 31, 44]. This is because the JVM acts as a black box that affects the benchmark's runtime performance non-deterministically. Non-determinism is primarily caused by adaptive optimization, dynamic class loading, Just-in-Time (JIT) compilation, garbage collection, and thread scheduling. Indeed, each of these components is executed in one or more threads in addition to all application threads causing non-deterministic runtime overheads.

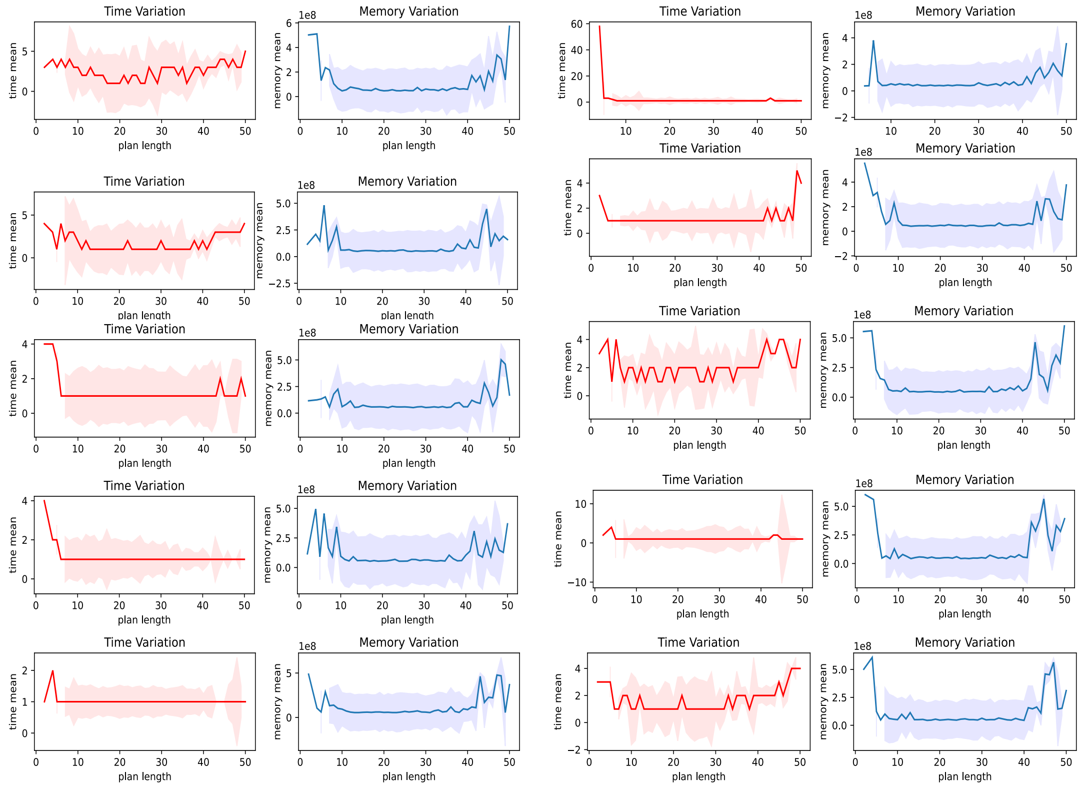

Figure 5.2: Performance results from the Block World's benchmarks. The figure presents on each row the resources needed to elaborate an explanation (general on the left and contrastive on the right) for a given type of question. Accordingly, the first provide the results for $\Theta_1$, the second for $\Theta_2$ until the last row that presents the outcomes for $\Theta_5$.

explanation of a given length. In the other column, we depict with blue lines the corresponding data for memory occupation. Furthermore, the figures show on each row the result of a different question. For example, on the first row of fig. 5.2, we illustrate on the left the resources required to compute a general explanation to $\Theta_1$ within the Block World domain whereas on the right we provide the respective data to elaborate a contrastive explanation for the same question in the same domain. In the second row of the image, we depict the corresponding data for $\Theta_2$, and so on until the last row where we represent the result from the benchmark analysis for $\Theta_5$. Accordingly, fig. 5.3 follows the same pattern in describing the data of the Logistics domain.

In addition to showing the average time and memory over time, the images also provide insight into the accuracy of the data. This is done by displaying a shadow area showing the interval of confidence. Overall, the results show that

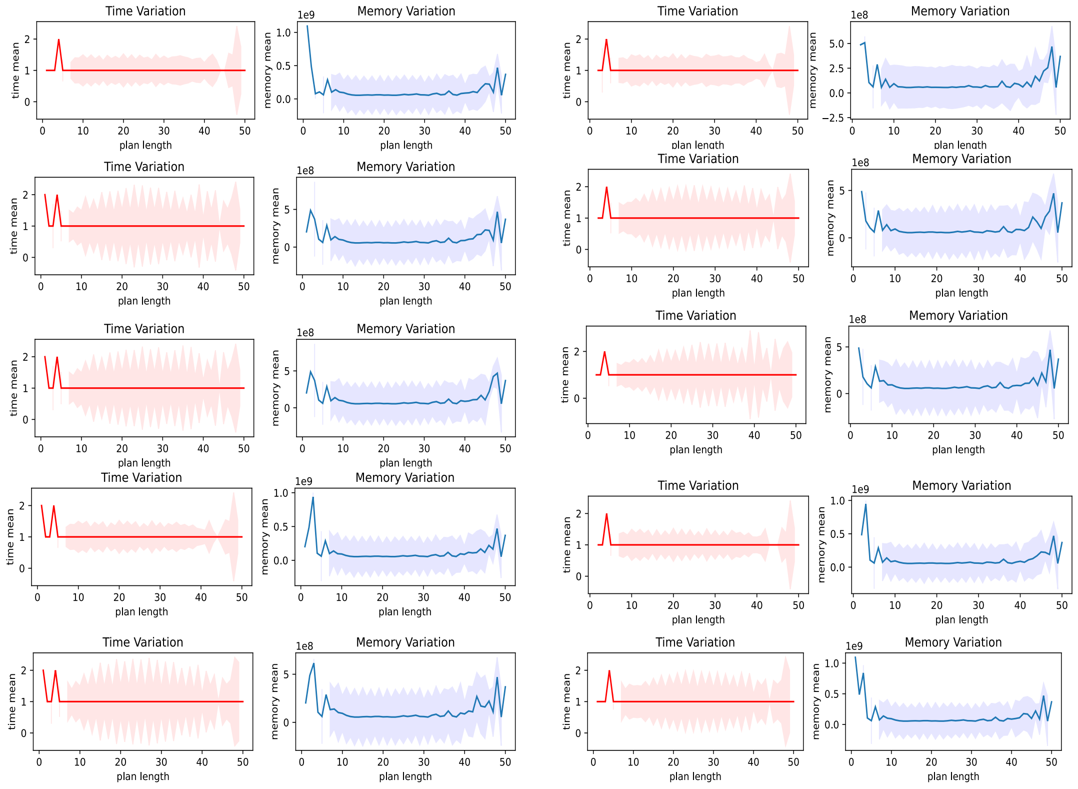

Figure 5.3: Performance results from the Logistics' benchmarks. The figure presents on each row the resources needed to elaborate an explanation (general on the left and contrastive on the right) for a given type of question. Accordingly, the first provide the results for $\Theta_1$, the second for $\Theta_2$ until the last row that presents the outcomes for $\Theta_5$.

| **Question type** | **Explanation type** | **Block World domain** | | **Logistics domain** | |
|---|---|---|---|---|---|
| | | Time | Memory | Time | Memory |
| $\Theta_1$ | General | 2,5 ms | 119,251,308 bytes | 1,0 ms | 128,466,830 bytes |
| | Contrastive | 2,4 ms | 75,959,546 bytes | 1,0 ms | 114,332,657 bytes |
| $\Theta_2$ | General | 1,8 ms | 107,094,759 bytes | 1,0 ms | 117,929,601 bytes |
| | Contrastive | 1,3 ms | 102,365,046 bytes | 1,0 ms | 112,935,841 bytes |
| $\Theta_3$ | General | 1,2 ms | 105,806,778 bytes | 1,0 ms | 121,106,859 bytes |
| | Contrastive | 2,1 ms | 126,118,829 bytes | 1,0 ms | 109,128,915 bytes |
| $\Theta_4$ | General | 1,1 ms | 116,822,555 bytes | 1,0 ms | 128,069,367 bytes |
| | Contrastive | 2,1 ms | 132,912,780 bytes | 1,0 ms | 127,204,626 bytes |
| $\Theta_5$ | General | 1,0 ms | 126,442,108 bytes | 1,0 ms | 125,156,009 bytes |
| | Contrastive | 1,7 ms | 122,230,364 bytes | 1,0 ms | 142,391,108 bytes |

Table 5.1: Table providing a comparison between the time and the memory required to calculate an explanation (general, or contrastive) for each question type and domain defined in Windows.

the length of the plan has no significant impact on the time and memory needed to compute the explanations; indeed, the resources required for the explanations remain almost unchanged when increasing the length of the plan. The only peaks on the graphs have to be considered outliers due to the JVM non-deterministic behaviour.

Table 5.1 summarizes the average time required to produce an explanation for a question of a given type regardless of the length of the plan. The table shows that the resources needed to compute explanations within the two domains are somehow comparable. To this extent, the time needed to elaborate explanations is between 1,0 and 2,5 milliseconds for the Block World domain whereas is slightly lower in the Logistics domain (around 1 millisecond). Memory allocation, on the other hand, follows the opposite trend. The bytes required for the explanation within the Logistics domain are mildly higher than the amount of memory needed for the Block World problem.

Finally, table 5.2 and table 5.3 show respectively the amount of time and memory we require to compute the different explanations in the two domains.

The analysis reveals that the resources needed for the elaboration of the explanations for the various questions in the two domains are barely different from one another. Accordingly, we conclude that there is no appreciable performance difference between the questions that require the compilation process and those that do not, or between the general explanation and the contrastive explanation.

| **Domain** | **Explanation type** | **Time** |
|---|---|---|
| **Block World** | general | 1,7 ms |
| | contrastive | 1,5 ms |
| **Logistics** | general | 1,0 ms |
| | contrastive | 1,0 ms |

Table 5.2: Table comparing the time required to calculate general and contrastive explanations in the Logistics and Block World domains.

| **Domain** | **Explanation type** | **Memory** |
|---|---|---|
| **Block World** | general | 111,917,313 bytes |
| | contrastive | 115,083,501 bytes |
| **Logistics** | general | 121,198,629 bytes |
| | contrastive | 124,145,733 bytes |

Table 5.3: Table comparing the memory required to compute general and contrastive explanations in the Logistics and Block World domains.

# Chapter 6

# Conclusions

In this thesis, we present a formal model and its software implementation for XAIP, besides a comprehensive and concise overview of the related state-of-the-art and knowledge background. In this context, this thesis addresses two central issues within the XAIP research area; the absence of a proper taxonomy on the main research lines and techniques exploited, besides the gap between theoretical and practical contributions.

Accordingly, we begin this work by examining the founding knowledge and the state-of-the-art of the XAIP landscape, devising a concise road map of its central features, along with a taxonomy of the relevant approaches and lines of research. In doing so, we introduce the issue that led to the redaction of this thesis: the shortage of practical contributions in the XAIP panorama.

Consequently, we present the pivotal contributions of this thesis as our model for XAIP. To this extent, unlike most of the solutions in the XAIP panorama, we introduce a two-folded proposal; first, we provide a formal model for XAIP, and then we devise a coherent implementation of the model into a software system. More specifically, after dwelling on the details of the model from the conceptual and software perspective, we propose an in-depth validation of the software library employing an extensive test suite to prove the correctness of its operations along with some benchmarks to evaluate its performances and end-to-end examples for its usability.

## 6.1 Open Challenges and Future Work

We conclude this thesis by examining possible future works and open challenges that the current proposal left unexplored. Accordingly, our proposal is not complete. We consider this work as the starting point for several research directions that we will examine in the following paragraphs.

To enhance the usage of our project, we believe that our model would benefit from an improvement of the explanation model. Our thesis mainly focuses on one type of explanation, contrastive explanation; however, we believe that different kinds of explanations may be more appropriate, depending on the application scenario. We consider that the development of an explanation hierarchy which includes plan property dependencies and argument-based explanations would increase the relevance of the project, helping it better adapt to different users' needs.

Furthermore, to enrich our explanations, making them more valuable to users, we believe it would be worthwhile to increase the number of properties we can study. In this regard, we deem that the proposal should be empowered with a later version of PDDL. Indeed, in this first release of the project, we designed the library to operate on `PDDL1.2` that only supports a restricted subset of properties as compared to the following implementation of the planning language. Utilizing a later release of PDDL will improve the significance level of our explanation as well as the quality of our plan. In addition, it will enable us to define more complex problems.

Within this framework, we consider that our proposal would benefit from a refinement of the `Domain` module. Therefore, we value enlarging the module to provide a broader number of domains with different levels of complexity and diverse themes.

We also judge that it would be of service to add more solvers to our proposal to allow users to assess differences in the solutions proposed by manifold planners.

Furthermore, besides improving the meaningfulness of the explanations, we deem that it would be desirable to improve the questions. More specifically, we believe that it would be advisable for users to query the system about partial plans and, ideally, to express questions in natural language. In this context, we envisage that in the future, we should include a new module to tackle the issue of translating users' questions from natural language into formal questions understandable by our system.

Another module we consider adding to enhance the reuse of code is a parser module for PDDL. This module will enable users to provide problems and domains in the PDDL language without redefining them using our library. In this way, users could only leverage the software to compute the solution.

Furthermore, as stated on different occasions in this document, the main goal of this thesis was to design a formal model and its software implementation for XAIP, thus, not providing some convenient way for non-expert users to leverage the framework. Nevertheless, given the multidisciplinary nature of the proposal, we realize that a significant contribution to enhancing the usage of the system also for non-expert users would be a graphical user interface (GUI). Indeed, due to the recent usage of planning systems in many high stake domains, planning is no longer an apanage of scholars. Along these lines, a GUI would provide a suitable way to allow users not experts in coding to exploit the system.

Finally, from a research perspective, we believe that one of the most relevant oversights of the proposed release is the absence of user evaluation. To this extent, we evaluate that given that the project touches multiple domains, it would be essential to assess its performance with real users. This would enable us to investigate the effectiveness of the explanations provided.

# Bibliography

[1] Angeline Aguinaldo and William Regli. A graphical model-based representation for classical AI plans using category theory. In *ICAPS 2021 Workshop on Explainable AI Planning*, 2021.

[2] Sule Anjomshoae, Davide Calvaresi, Amro Najjar, and Kary Främling. Explainable agents and robots: Results from a systematic literature review. volume 2, 2019.

[3] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58, 2020.

[4] Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[5] Kim Baraka, Stephanie Rosenthal, and Manuela Veloso. Enhancing human understanding of a mobile robot's state and actions using expressive lights. 2016.

[6] Jonathan Mark Bull, L. A. Smith, Martin D. Westhead, David Henty, and R. A. Davey. A benchmark suite for high performance java. *Concurr. Pract. Exp.*, 12:375–388, 2000.

[7] Gerard Canal, Guillem Alenyà, and Carme Torras. Adapting robot task planning to user preferences: an assistive shoe dressing example. *Autonomous Robots*, 43, 2019.

[8] Yaniel Carreno, Alan Lindsay, and Ron Petrick. Explaining temporal plans with incomplete knowledge and sensing information. In *ICAPS 2021 Workshop on Explainable AI Planning*, 2021.

[9] Michael Cashmore, Anna Collins, Benjamin Krarup, Senka Krivic, Daniele Magazzeni, and David Smith. Towards explainable ai planning as a service. 8 2019.

[10] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, and Bram Ridder. Opportunistic planning in autonomous underwater missions. *IEEE Transactions on Automation Science and Engineering*, 15, 2018.

[11] Tathagata Chakraborti, Kshitij P. Fadnis, Kartik Talamadupula, Mishal Dholakia, Biplav Srivastava, Jeffrey O. Kephart, and Rachel K.E. Bellamy. Visualizations for an explainable planning agent. volume 2018-July, 2018.

[12] Tathagata Chakraborti, Sarath Sreedharan, and Subbarao Kambhampati. The emerging landscape of explainable AI planning and decision making. *CoRR*, abs/2002.11697, 2020.

[13] Tathagata Chakraborti, Sarath Sreedharan, Anagha Kulkarni, and Subbarao Kambhampati. Alternative modes of interaction in proximal human-in-the-loop operation of robots. *ArXiv*, abs/1703.08930, 3 2017.

[14] Michael Chromik and Andreas Butz. Human-xai interaction: A review and design principles for explanation user interfaces. volume 12933 LNCS, 2021.

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

[16] Giovanni Ciatto. Automatic inference on horn clauses, March 2022.

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

[18] Giovanni Ciatto, Michael I. Schumacher, Andrea Omicini, and Davide Calvaresi. Agent-based explanations in ai: Towards an abstract framework. volume 12175 LNAI, 2020.

[19] Anna Collins, Daniele Magazzeni, and Simon Parsons. Towards an argumentation-based approach to explainable planning, 2019.

[20] Stephen A. Cook and Yongmei Liu. A Complete Axiomatization for Blocks World. *Journal of Logic and Computation*, 13(4):581–594, 08 2003.

[21] Rebecca Eifler, Michael Cashmore, Jörg Hoffmann, Daniele Magazzeni, and Marcel Steinmetz. Explaining the space of plans through plan-property dependencies. *2nd International Workshop on Explainable AI Planning*, 2019.

[22] Rebecca Eifler, Michael Cashmore, Jörg Hoffmann, Daniele Magazzeni, and Marcel Steinmetz. A new approach to plan space explanation: Analyzing plan-property dependencies in oversubscription planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(06):9818–9826, Apr. 2020.

[23] Heike Felzmann, Eduard Fosch-Villaronga, Christoph Lutz, and Aurelia Tamo-Larrieux. Robots and transparency: The multiple dimensions of transparency in the context of robot technologies. volume 26, 2019.

[24] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

[25] James Forrest, Somayajulu Sripada, Wei Pang, and George M. Coghill. Are contrastive explanations useful? volume 2894, 2021.

[26] Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley, 2011.

[27] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20, 2003.

[28] Maria Fox, Derek Long, and Daniele Magazzeni. Explainable planning. 9 2017.

[29] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Method-level phase behavior in java workloads. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, page 270–287, New York, NY, USA, 2004. Association for Computing Machinery.

[30] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Method-level phase behavior in java workloads. *SIGPLAN Not.*, 39(10):270–287, oct 2004.

[31] Joseph Yossi Gil, Keren Lenz, and Yuval Shimron. A microbenchmark case study and lessons learned. In *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, &amp; VMIL'11*, SPLASH '11 Workshops, page 297–308, New York, NY, USA, 2011. Association for Computing Machinery.

[32] Adrian Salazar Gomez. Explainable artificial intelligence: A review of the literature.

[33] Bryce Goodman and Seth Flaxman. European union regulations on algorithmic decision making and a "right to explanation". *AI Magazine*, 38, 2017.

[34] 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.

[35] 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.

[36] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*, volume 13. 2019.

[37] Andreas Hinz, Sandi Klavžar, and Ciril Petr. *The Tower of Hanoi – Myths and Maths.* 04 2018.

[38] Jörg Hoffmann and Daniele Magazzeni. Explainable ai planning (xaip): overview and the case of contrastive explanation. *Reasoning Web. Explainable Artificial Intelligence*, pages 277–282, 2019.

[39] David S Hollman. Thoughts on curiously recurring template pattern. Technical report, Sandia National Lab.(SNL-CA), Livermore, CA (United States), 2019.

[40] Mir Riyanul Islam, Mobyen Uddin Ahmed, Shaibal Barua, and Shahina Begum. A systematic review of explainable artificial intelligence in terms of different application domains and tasks. *Applied Sciences (Switzerland)*, 12, 2022.

[41] Benjamin Krarup, Michael Cashmore, Daniele Magazzeni, and Tim Miller. Towards model-based contrastive explanations for explainable planning.

[42] Benjamin Krarup, Senka Krivic, Daniele Magazzeni, Derek Long, Michael Cashmore, and David E. Smith. Contrastive explanations of plans through model restrictions. *Journal of Artificial Intelligence Research*, 72, 2021.

[43] Ashwin Kumar, S. Vasileiou, Melanie Bancilhon, Alvitta Ottley, and William G. S. Yeoh. Vizxp: A visualization framework for conveying explanations to users in model reconciliation problems. In *ICAPS*, 2022.

[44] Michael Kuperberg, Fouad Omri, and Ralf H. Reussner. Automated benchmarking of java apis. In Gregor Engels, Markus Luckey, and Wilhelm Schäfer, editors, *Software Engineering 2010 - Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26.2.2010 in Paderborn*, volume P-159 of *LNI*, pages 57–68. GI, 2010.

[45] Pat Langley, Ben Meadows, Mohan Sridharan, and Dongkyu Choi. Explainable agency for intelligent autonomous systems. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 4762–4763. AAAI Press, 2017.

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

[47] He Li, Lu Yu, and Wu He. The impact of gdpr on global technology development. *Journal of Global Information Technology Management*, 22, 2019.

[48] Vladimir Lifschitz. On the semantics of strips. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, 1987.

[49] A. Lindsay. Using generic subproblems for understanding and answering queries in xaip. 2020.

[50] Drew M. McDermott. The 1998 ai planning systems competition. *AI Magazine*, 21(2):35, Jun. 2000.

[51] Tim Miller. Explanation in artificial intelligence: Insights from the social sciences, 2019.

[52] Bilge Mutlu, Jodi Forlizzi, Illah Nourbakhsh, and Jessica Hodgins. The use of abstraction and motion in the design of social interfaces. volume 2006, 2006.

[53] Mark A. Neerincx, Jasper van der Waa, Frank Kaptein, and Jurriaan van Diggelen. Using perceptual and cognitive explanations for enhanced human-agent team performance. volume 10906 LNAI, 2018.

[54] Nils J Nilsson and Donald L Nielson. Shakey the robot, 1984.

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

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

[57] Edwin PD Pednault. Adl and the state-transition model of action. *Journal of logic and computation*, 4(5):467–512, 1994.

[58] B. Prasad. A planning system for blocks-world domain. In *Proceedings ACS/IEEE International Conference on Computer Systems and Applications*, pages 59–64, 2001.

[59] Randy L Ribler, Marc Abrams, Randy Ribler, and Anup Mathur. Visualizing and modeling categorical time series data phd work with the network research group (nrg) at virginia tech view project visualizing and modeling categorical time series data. 1995.

[60] Andrea Roli. Planning, March 2022.

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

[62] Stuart Russel and Peter Norvig. *Artificial intelligence—a modern approach 3rd Edition.* 2012.

[63] Fatai Sado, Chu Kiong Loo, Wei Shiung Liew, Matthias Kerzel, and Stefan Wermter. Explainable goal-driven agents and robots – a comprehensive review. 4 2020.

[64] Susanne Seitinger, Daniel M. Taub, and Alex S. Taylor. Light bodies: Exploring interactions with responsive lights. 2010.

[65] Mauro Vallati, Daniele Magazzeni, Bart De Schutter, Lukáš Chrpa, and Thomas L. McCluskey. Efficient macroscopic urban traffic models for reducing congestion: A pddl+ planning approach. 2016.

[66] Giulia Vilone and Luca Longo. Explainable artificial intelligence: a systematic review. *CoRR*, abs/2006.00093, 2020.

[67] Giulia Vilone and Luca Longo. Classification of explainable artificial intelligence methods through their output formats. *Machine Learning and Knowledge Extraction*, 3, 2021.

[68] Paul Voigt and Axel von dem Bussche. *The EU General Data Protection Regulation (GDPR) A Practical Guide.* 2017.

[69] Sandra Wachter, Brent Mittelstadt, and Luciano Floridi. Why a right to explanation of automated decision-making does not exist in the general data protection regulation. *International Data Privacy Law*, 7, 2017.

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

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

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

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

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

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

[76] PETRI YLIKOSKI. *THE IDEA OF CONTRASTIVE EXPLANANDUM*, pages 27–42. Springer Netherlands, Dordrecht, 2007.