Title

Antoine Gréa

Abstract

Table of Content

Acknowledgements

Introduction

0.1 Thesis Context

0.2 Motivations

0.2.1 Observation

0.2.2 Abstraction

0.2.3 Cognition

0.3 Issues

0.4 Contributions

0.5 Plan

1 Knowledge Representation

SAMIR:

- Cite slow start 1.2
- System : implentation -> Model : theory
- Examples
- theorems or not
- s : D -> D notation in table
- Double arrow too for ±
- to port : porter une notion in smth
- 1.2.2 : criterion
- Example of parsing
- 1.2.6.1 : light defeasible logic
- lacks properties and proofs: what are the advantages ?
- **TODO**: Chap 1

Knowledge representation is at the intersection of maths, logic, language and computer sciences. Its research starts at the end of the 19^{th} century, with Cantor inventing set theory (Cantor 1874). Then after a crisis in the beginning of the 20^{th} century with Russel's paradox and Gödel's incompletude theorem, revised versions of the set theory become one of the foundations of mathematics. The most accepted version is the Zermelo-Fraenkel axiomatic set theory with the axiom of Choice (ZFC) (Fraenkel *et al.* 1973, vol. 67; Ciesielski 1997). This effort leads to a formalization of mathematics itself, at least to a certain degree.

Knowledge description systems rely on syntax to interoperate systems and users to one another. The base of such languages comes from the formalization of automated grammars by Chomsky (1956). It mostly consists of a set of hierarchical rules aiming to deconstruct an input string into a sequence of terminal symbols. This deconstruction is called parsing and is a common operation in computer science. More tools for the characterization of computer language emerged soon after thanks to Backus (1959) while working on a programming language at IBM. This is how the Backus-Naur Form (BNF) metalanguage was created on top of Chomsky's formalization.

A similar process happened in the 1970's, when logic based knowledge representation gained popularity among computer scientists (Baader 2003). Systems at the time explored notions such as rules and networks to try and organize knowledge into a rigorous structure. At the same time other systems were built based on First Order Logic (FOL). Then, around the 1990's, the research began to merge in search of common semantics in what led to the development of Description Logics (DL). This domain is expressing knowledge as a hierarchy of classes containing individuals.

From there and with the advent of the world wide web, engineers were on the lookout for standardization and interoperability of computer systems. One such standardization took the name of "semantic web" and aimed to create a widespread network of connected services sharing knowledge between one another in a common language. At the beginning of the 21^{st} century, several languages were created, all based on the World Wide Web Consortium (W3C) specifications called Resource Description Framework

Table 1.1: List of classical symbols for logic.

Symbol	Description
$=, \neq$	Equal and not equal.
$e : ?(e)$	The colon is a separator to be read as "such that". Also used for typing.
$\top, \bot$	Top and bottom symbols used as true and false respectively.
$?(e)$	Predicate over e.
$\neg, \wedge, \vee, \veebar$	Negation (not), conjunction (and), disjunction (logical or) and either.
$\vdash$	Entails, used for logical implication and consequence.
$\forall, \exists, \exists!, \nexists, \S$	Universal, existential, uniqueness, exclusive and solution quantifiers.
$[?(e)]$	Iverson's brackets: $[\bot] = 0$ and $[\top] = 1$.

(RDF) (Klyne and Carroll 2004). This language is based on the notion of statements as triples. Each can express a unit of knowledge. All the underlying theoretical work of DL continued with it and created more expressive derivatives. One such derivative is the family of languages called Web Ontology Language (OWL) (Horrocks *et al.* 2003). Ontologies and knowledge graphs are more recent names for the representation and definition of categories (DL classes), properties and relation between concepts, data and entities.

All these tools are the base for all modern knowledge representations. In the rest of this chapter, we discuss the fundamentals of each of the aspects of knowledge description, then we propose a knowledge description framework that is able to adapt to its usage.

1.1 Fundamentals

First, we present the list of notations in this document. While trying to stick to traditional notations, we also aim for an unambiguous symbols across several domains while remaining concise and precise.

1.1.1 Foundation of maths and logic systems

In order to understand knowledge representation, some mathematical and logical tools need to be presented.

1.1.1.1 First Order Logic

The first mathematical notion we define is logic. More precisely First Order Logic (FOL) in the context of DL. All notations are presented in table 1.1. FOL is based on boolean logic with the two values $\top$ *true* and $\bot$ *false* along with the classical boolean operators $\neg$ *not*, $\wedge$ *and* and $\vee$ *or*. These are defined in the following way :

- $\neg\top = \bot$, not true is the same as false.
- $a \wedge b \vdash (a = b = \top)$, a and b is true when they are both simultaneously true.
- $\neg(a \vee b) \vdash (a = b = \bot)$, a or b is true if both variables are not false.

With $\vdash$ being the logical implication also called entailement and $=$ being the identity relation. When combining logical operators with boolean variables, we form *expressions* also called formulas. These expressions can be evaluated given an interpretation of the variable to return a boolean value. Any function that returns a boolean is called a *predicate* (noted $?(e)$). Relations that takes an expression as parameter are called *modifiers*. FOL introduce a useful kind of modifer used to generalize expressions:

Table 1.2: List of classical symbols and syntax for sets.

Symbol	Description
$\varnothing$	Empty set, also noted $\{\}$.
$e \in \mathcal{S}$	Element e is a member of set $\mathcal{S}$.
$\subset, \cup, \cap, \setminus, \times$	Set inclusion, union, intersection, difference and cartesian product.
$\lvert\mathcal{S}\rvert$	Cardinal (number of elements) of set $\mathcal{S}$.
$\{e : ?(e)\}$	Set builder notation, set of all e such that $?(e)$ is true.
$\wp(\mathcal{S})$	Powerset: set of all subsets of $\mathcal{S}$.

quantifiers. Quantifiers can specify restriction on a variable. These restrictions forces the expression to be true in specific cases depending on the quantifier used.

The classical quantifiers includes the following:

- The *universal quantifier* $\forall$ meaning *"for all"*.
- The *existential quantifier* $\exists$ meaning *"it exists"*.
- The *uniqueness quantifier* $\exists!$ meaning *"it exists a unique"*.
- The *exclusive quantifier* $\nexists$ meaning *"it doesn't exist"*.
- The *solution quantifier* $\S$ meaning *"those"* (Hehner 2012).

The last three quantifiers are optional in FOL but will be conducive later on. It is interesting to note that most quantified expression can be expressed using the set builder notation discussed in the following section.

1.1.1.2 Set Theory

Since we need to represent knowledge, we will handle more complex data than simple booleans. At the beginning of his funding work on set theory, Cantor wrote:

> "*A set is a gathering together into a whole of definite, distinct objects of our perception or of our thought–which are called elements of the set.*" George Cantor (1895)

For Cantor, a set is a collection of concepts and percepts. We define a set using the notations in table 1.2.
Definition 1 (Set). A collection of *distinct* objects considered as an object in its own right. We define a set one of two ways (always using braces):

- In extension by listing all the elements in the set: $\{0, 1, 2, 3, 4\}$
- In intension by specifying the rule that all elements follow: $\{n : n \in \mathbb{N} \land (n \leq 4)\}$

The member relation is noted $e \in \mathcal{S}$ to indicate that e is an element of $\mathcal{S}$. We note $\mathcal{S} \subset \mathcal{T} \vdash ((e \in \mathcal{S} \vdash e \in \mathcal{T}) \land \mathcal{S} \neq \mathcal{T})$, that a set $\mathcal{S}$ is a proper subset of a more general set $\mathcal{T}$.

We also define the union, intersection and difference as following:

- $\mathcal{S} \cup \mathcal{T} = \{e : e \in \mathcal{S} \lor e \in \mathcal{T}\}$
- $\mathcal{S} \cap \mathcal{T} = \{e : e \in \mathcal{S} \land e \in \mathcal{T}\}$
- $\mathcal{S} \setminus \mathcal{T} = \{e : e \in \mathcal{S} \land e \notin \mathcal{T}\}$

An interesting way to visualize relationsips with sets is by using Venn diagrams. In figure 1.1 we present the classical set operations.

These diagrams have a lack of expressivity regarding complex operations on sets. Indeed, from their plannar form it is complicated to express numerous sets having intersection and disjunctions. One

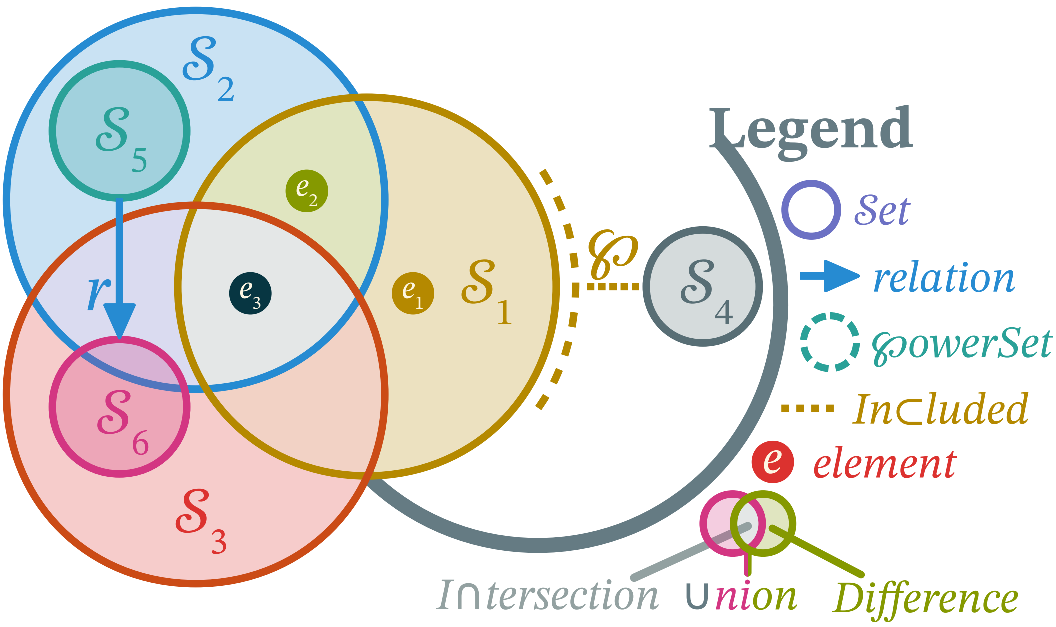

Figure 1.1: Example of Venn diagram to illustrate operations on sets.

exemple is the cartesian product that is defined as $\mathcal{S} \times \mathcal{T} = \{\langle e_\mathcal{S}, e_\mathcal{T} \rangle : e_\mathcal{S} \in \mathcal{S} \wedge e_\mathcal{T} \in \mathcal{T}\}$.

We can also define the set power recursively by $\mathcal{S}^1 = \mathcal{S}$ and $\mathcal{S}^n = \mathcal{S} \times \mathcal{S}^{n-1}$.

The most common axiomatic set theory is ZFC. In that definition of sets there are a few notions that comes from its axioms. By being able to distinguish elements in the set from one another we assert that elements have an identity and we can derive equality from there:
Axiom (Extensionality). $\forall \mathcal{S} \forall \mathcal{T} : \forall e((e \in \mathcal{S}) = (e \in \mathcal{T})) \vdash \mathcal{S} = \mathcal{T}$

Another axiom of ZFC that is crucial in avoiding Russel's paradox ($\mathcal{S} \in \mathcal{S}$) is the following:
Axiom (Foundation). $\forall \mathcal{S} : (\mathcal{S} \neq \varnothing \vdash \exists \mathcal{T} \in \mathcal{S}, (\mathcal{T} \cap \mathcal{S} = \varnothing))$

This axiom uses the empty set $\varnothing$ (also noted $\{\}$) as the set with no elements. Since two sets are equals if and only if they have precisely the same elements, the empty set is unique.

The definition by intention uses *set builder notation* to define a set. It is composed of an expression and a predicate $?$ that will make any element e in a set $\mathcal{T}$ satisfying it part of the resulting set $\mathcal{S}$, or as formulated in ZFC:
Axiom (Specification). $\forall ? \forall \mathcal{T} \exists \mathcal{S} : (\forall e \in \mathcal{S} : (e \in \mathcal{T} \wedge ?(e)))$

The last axiom of ZFC we use is to define the power set $\wp(\mathcal{S})$ as the set containing all subsets of a set $\mathcal{S}$:
Axiom (Power set). $\wp(\mathcal{S}) = \{\mathcal{T} : \mathcal{T} \subseteq \mathcal{S}\}$

With the symbol $\mathcal{S} \subseteq \mathcal{T} \vdash (\mathcal{S} \subset \mathcal{T} \vee \mathcal{S} = \mathcal{T})$. These symbols have an interesting property as they are often used as a partial order over sets.

1.1.1.3 Relational algebra

From set theory, it is possible to add relations between sets.

Table 1.3: List of classical symbols and syntax for relational algebra.

Symbol	Description
$f \circ g$	Function composition also noted $g(f(x))$
σ, π	Selection and projection of a relation.
$\langle e_1, e_2, e_n \rangle$	n-uple also called a relation.
$\mathcal{S} \to \mathcal{S}$	Association relation. Used for graph edges and domain definition.
$e \mapsto f(e)$	Substitution relation.

Table 1.4: List of classical symbols and syntax for graphs.

Symbol	Description
$g = (V, E)$	Graph g with set of vertices V and edges E.
$\phi^{\pm\|n\|*}(e\|v)$	Incidence (edge) and adjacence (vertex) function for graphs:
ϕ	• A tuple or set representing the edge or all adjacent edges of a vertex.
ϕ^-	• Source vertex (subject) or set of all incoming edges of a vertex.
ϕ^+	• Target vertex (object) or set of all outgoing edges of a vertex.
ϕ^0	• Label of edges and vertex : property of a statement or cause of a causal link.
$\chi(g)^+$	Transitive closure of graph g.
$\div$	Graph quotient.

Definition 2 (Relation)**.** A relation is effectively a subset of the cartesian product between several sets: $r = \sigma\wp\left(\bigtimes_{i=1}^{n} \mathcal{S}_i\right)$ with σ being the selection relation and n being the *arity* of the relation r.

It can also be noted as a set of tuples each noted $\langle e_1, e_2, e_n \rangle$.

We need to define some special relations often used for set manipulation:

- The **substitution** that replace a variable in an expression $\mathbb{e}$ such that: $(e \mapsto f(e))(\mathbb{e}(e)) = \mathbb{e}(f(e))$. The substitution is often used for function definition.
- The **selection** that selects elements given a predicate ? such that: $\sigma_?(\mathcal{S}) = \{e : ?(e) \wedge e \in \mathcal{S}\}$. The choice selection $\sigma_!(\mathcal{S})$ is a non deterministic choice of one element in $\mathcal{S}$.
- The **projection** that merges elements of a set given a filter f such that: $\pi_f(\mathcal{S}) = \{f(e) : e \in \mathcal{S}\}$. The default projection uses the identity relation $=$ instead of f.

Functions are a special case of relations that takes as value the selected element of the last set. We note them $f : \left(\bigtimes_{i=1}^{n-1} \mathcal{S}_i\right) \to \mathcal{S}_n$. The set $\bigtimes_{i=1}^{n-1} \mathcal{S}_i$ is called the *domain* of the function $\mathcal{D}(f)$ and the set $\mathcal{S}_n$ is caled the *codomain* $\mathcal{D}^{-1}(f)$. The number $n-1$ is called the degree of the function.

We can combine functions using the *function composition operator* $g \circ f = x \mapsto g(f(x))$. The functional power of f noted f^n is defined recursively as :

- $f^0 = x \mapsto x$, $f^1 = x \mapsto f(x)$ and $f^{-1} = f(x) \mapsto x$
- $f^n = f \circ f^{n+[n<0]-[n>0]}$ with $[?(n)]$ being an Iverson bracket ($[\top] = 1$ and $[\bot] = 0$).

1.1.1.4 Graphs

Next in line, we need to define a few notions of graph theory.
Definition 3 (Graph)**.** A graph is a mathematical structure $g = (V, E)$ consisting of vertices V (also called nodes) and edges E (arcs) that links two vertices together. Each edge is basically a pair of vertices ordered or not depending on if the graph is directed or not. We can write $E = \sigma\wp(V^2)$

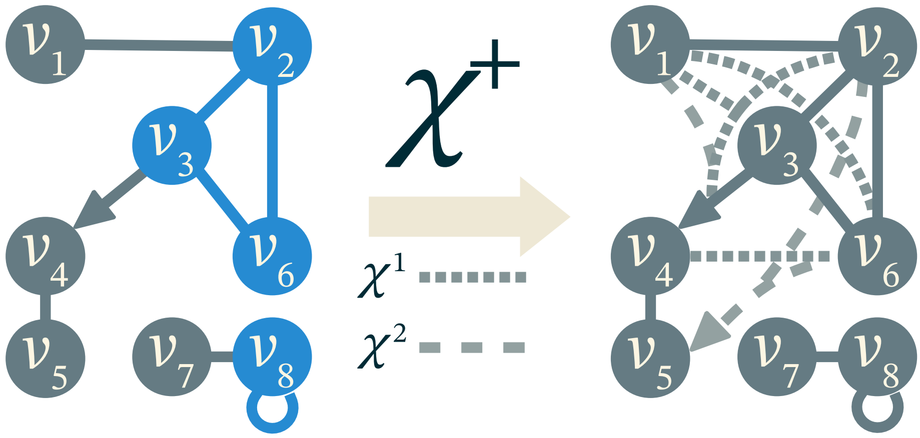

Figure 1.2: Example of the recursive application of the transitive cover to a graph.

A graph is often represented with lines or arrows linking points together like illustrated in figure 1.2. In that figure, the vertices v_1 and v_2 are connected through an undirected edge. Similarly v_3 connects to v_4 but not the opposite since they are bonded with a directed edge. The vertex v_8 is also connected to itself.

From that definition, some other relations are needed to express most properties of graphs. In the following, the signed symbol only applies to directed graphs.

We provide graphs with an adjacence function ϕ over any vertex $v \in V$ such that:

- $\phi(v) = \{e : e \in E \wedge v \in e\}$
- $\phi^+(v) = \{\langle v \to v' \rangle \in E : v' \in V\}$ and $\phi^-(v) = \{\langle v' \to v \rangle \in E : v' \in V\}$

This relation gives the set of incoming or outgoing edges from any vertex. In non directed graphs, the relation gives edges adjacent to the vertex. For example: in figure 1.2, $\phi(v_1) = \{\langle v_1, v_2 \rangle\}$. In that example, using directed graph notation we can note $\phi^+(v_3) = \{\langle v_3 \to v_4 \rangle\}$.

Using types, it is possible to reuse the same symbol to define an incidence function over any edges $e = \langle v, v' \rangle$ such that:

- $\phi(e) = \langle v, v' \rangle$
- $\phi^-(e) = v$ and $\phi^+(e) = v'$

Most of the intrinsic information of a graph is contained within its structure. Exploring its properties require to study the "shape" of a graph and to find relationships between vertices. That is why graph properties are easier to explain using the *transitive cover* χ^+ of any graph $g = (V, e)$ defined as follows:

- $\chi(g) = (V, e') : e' = e \cup \{\langle v_1, v_3 \rangle : \{\langle v_1, v_2 \rangle, \langle v_2, v_3 \rangle\} \subset e\}$
- $\chi^+ = \chi^\infty$

This transitive cover will create another graph in which two vertices are connected through an edge if and only if it exists a path between them in the original graph g. We illustrate this process in figure 1.2. Note how there is no edge in $\chi(g)$ between v_5 and v_6 and the one in $\chi^2(g)$ is directed towards v_5 because there is no path back to v_6 since the edge between v_3 and v_4 is directed.

Definition 4 (Path). We say that vertices v_1 and v_2 are *connected* if it exists a path from one to the other. Said otherwise, there is a path from v_1 to v_2 if and only if $\langle v_1, v_2 \rangle \in E_{\chi^+(g)}$.

The notion of connection can be extended to entire graphs. An undirected graph g is said to be *connected* if and only if $\forall e \in V^2 (e \in E_{\chi^+(g)})$.

Similarly we define *cycles* as the existence of a path from a given vertex to itself. For example, in figure 1.2, the cycles of the original graph are colored in blue. Some graphs can be strictly acyclical, enforcing the absence of cycles.

A **tree** is a special case of a graph. A tree is an acyclical connected graph. If a special vertex called a *root* is chosen we call the tree a *rooted tree*. It can then be a directed graph with all edge pointing away from the root. When progressing away from the root, we call the current vertex *parent* of all exterior *children* vertices. Vertex with no children are called *leaves* of the tree and the rest are called *branches*.

An interesting application of trees to FOL is called *and/or trees* where each vertex has two sets of children: one for conjunction and the other for disjunction. Each vertex is a logic formula and the leaves are atomic logic propositions. This is often used for logic problem reduction. In figure 1.3 we illustrate how and/or trees are often depicted.

Figure 1.3: Example of and/or tree.

Another notion often used for reducing big graphs is the quotiening as illustrated in figure 1.4.
Definition 5 (Graph Quotient). A quotient over a graph is the act of reducing a subgraph into a node while preserving the external connections. All internal structure becomes ignored and the subgraph now acts like a regular node. We note it $\div_f(g) = (\pi_f(V), \{\pi_f(e) : e \in E\})$ with f being a function that maps any vertex either toward itself or toward its quotiened vertex.

We can also combine several graphs into one using fusion: $g_1 + g_2 = (V_1 \cup V_2, E_1 \cup E_2)$.

Figure 1.4: Example of graph quotient.

1.1.1.5 Hypergraphs

A generalization of graphs are **hypergraphs** where the edges are allowed to connect to more than two vertices. They are often represented using Venn-like representations but can also be represented with edges "gluing" several vertex like in figure 1.5.

An hypergraph is said to be n*-uniform* if the edges are restricted to connect to only n vertices together. In that regard, classical graphs are 2-uniform hypergraphs.

Hypergraphs have a special case where $E \subset V$. This means that edges are allowed to connect to other edges. In figure 1.5, this is illustrated by the edge e_3 connecting to three other edges. Information about these kinds of structures for knowledge representation is hard to come by and rely mostly on a form of "folk wisdom" within the mathematics community where knowledge is rarely published and mostly transmitted orally during lessons. One of the closest information available is this forum post (Kovitz 2018) that associated this type of graph to port graphs (Silberschatz 1981). Additional information was found in the form of a contribution of Vepstas (2008) on an encyclopedia article about hypergraphs. In that contribution, he says that a generalization of hypergraph allowing for edge-to-edge connections violate the axiom of Foundation of ZFC by allowing edge-loops. Indeed, like in figure 1.5, an edge $e_9 = \{e_{10}\}$ can connect to another edge $e_{10} = \{e_9\}$ causing an infinite descent inside the $\in$ relation in direct contradiction with ZFC.

This shows the limits of standard mathematics especially on the field of knowledge representation. Some structures needs higher dimensions than allowed by the one-dimensional structure of ZFC and FOL. However, it is important not to be mistaken: such non-standard set theories are more general than ZFC and therefore contains ZFC as a special case. All is a matter of restrictions.

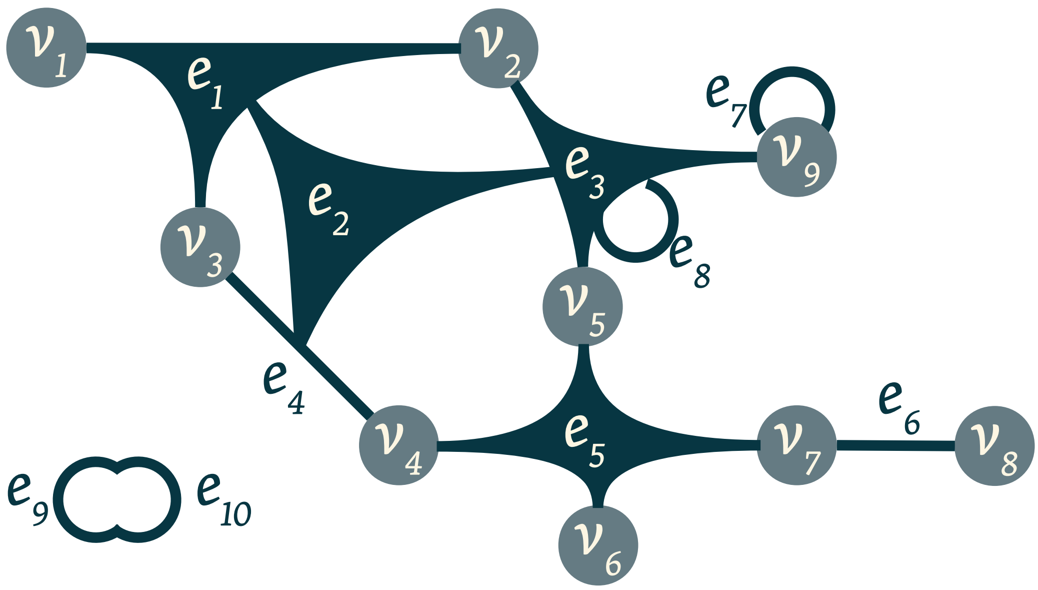

Figure 1.5: Example of hypergraph with total freedom on the edges specification.

Table 1.5: List of symbols and syntax for sheaves.

Symbol	Description
$\bullet, \ast, \multimap$	Germ, seed and connector.
$\mathcal{F}$	Sheaf (from French *faisceau*).

1.1.1.6 Sheaf

In order to understand sheaves, we need to present a few auxiliary notions. Most of these definitions are adapted from (Vepštas 2008). The first of which is a seed.

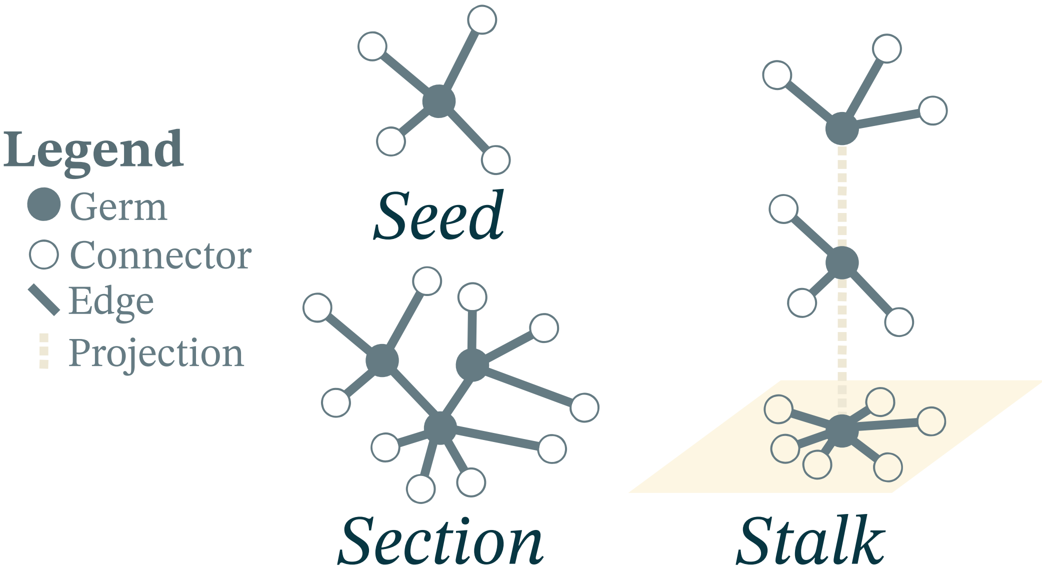

Figure 1.6: Example of a seed, a section and a stalk.

Definition 6 (Seed)**.** A seed corresponds to a vertex along with the set of adjacent edges. Formally we note a seed $\ast = (\bullet, \phi_g(\bullet))$ that means that a seed build from the vertex $\bullet$ in the graph g contains a set of adjacent edges $\phi_g(\bullet)$. We call the vertex $\bullet$ the *germ* of the seed. The edges in a seed does not connect to the other vertices but keep the information and are able to match the correct vertices through typing (often a type of a single individual). We call the edges in a seed *connectors*.

Seeds are extracts of graphs that contains all information about a vertex. Illustrated in the figure 1.6, seeds have a central germ (represented with discs) and connectors leading to a typed vertex (outlined circles). Those external vertices are not directly contained in the seed but the information about what vertex can fit in them is kept. It is useful to represent connectors like jigsaw puzzle pieces: they can match only a restricted number of other pieces that match their shape.

From there, it is useful to build a kind of partial graph from seeds called sections.
Definition 7 (Section)**.** A section is a set of seeds that have their common edges connected. This means that if two seeds have an edge in common connecting both germs, then the seeds are connected in the section and the edges are merged. We note $g_\ast = (\bullet, ⌕)$ the graph formed by the section.

In figure 1.6, a section is represented. It is a connected section composed of seeds along with the additional seeds of any vertices they have in common. They are very similar to subgraph but with an additional border of typed connectors. This tool was originally mostly meant for big data and categorization over large graphs. As graph quotient is often used in that domain, it was ported to sections instead of graphs allows us to define stalks.
Definition 8 (Stalk)**.** Given a projection function $f : \bullet \to \bullet'$ over the germs of a section $\ast$, the stalk above the vertex $\bullet' \in \bullet'$ is the quotient of all seeds that have their germ follow $f(\bullet) = \bullet'$.

The quotienning is used in stalks for their projection. Indeed, as shown in figure 1.6, the stalks are simply a collection of seeds with their germs quotiened into their common projection. The projection can be any process of transformation getting a set of seeds in one side and gives object in any base space called the image. Sheaves are a generalization of this concept to sections.

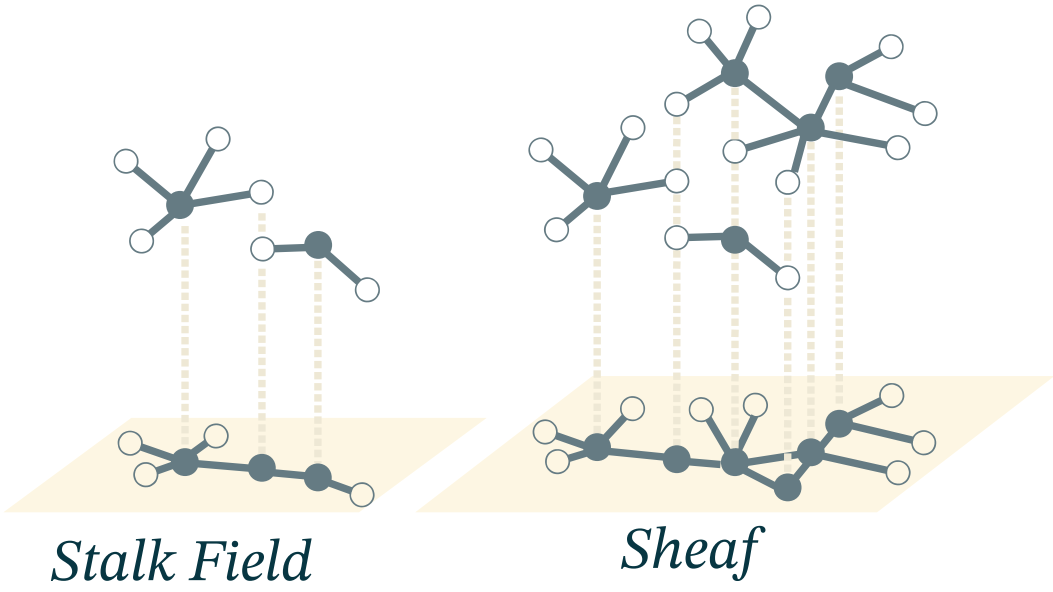

Figure 1.7: Example of sheaves.

Definition 9 (Sheaf). A sheaf is a collection of sections, together with a projection. We note it $\mathcal{F} = \langle G_\star, \pi_g \rangle$ with the function g being the gluing axioms that the projection should respect depending on the application. The projected sheaf graph is noted $g_\mathcal{F} = \sum_{g_\star \in G_\star} \div_g(g_\star)$ as the fusion of all quotiened sections.

By merging common vertices into a section, we can build stack fields. These fields are simply a subcategory of sheaves. Illustrated in figure 1.7, a sheaf is a set of section with a projection relation.

1.1.2 Grammar and Parsing

Grammar is an old tool that used to be dedicated to linguists. With the funding works by Chomsky and his Contex-Free Grammars (CFG), these tools became available to mathematicians and shortly after to computer scientists.

A CFG is a formal grammar that aims to generate a formal language given a set of hierarchical rules. Each rule is given a symbol as a name. From any finite input of text in a given alphabet, the grammar should be able to determine if the input is part of the language it generates.

1.1.2.1 BNF

In computer science, popular metalanguage called BNF was created shortly after Chomsky's work on CFG. The syntax is of the following form :

```
<rule> ::= <other_rule> | <terminal_symbol> | "literals"
```

A terminal symbol is a rule that does not depend on any other rule. It is possible to use recursion, meaning that a rule will use itself in its definition. This actually allows for infinite languages. Despite its expressive power, BNF is often used in one of its extended forms.

In this context, we present a widely used form of BNF syntax that is meant to be human readable despite not being very formal. We add the repetition operators * and + that respectively repeat 0 and 1 times or more the preceding expression. We also add the negation operator ~ that matches only if the following expression does not match. We also add parentheses for grouping expression and brackets to group literals.

1.1.2.2 Dynamic Grammar

A regular grammar is static, it is set once and for all and will always produce the same language. In order to be more flexible we need to talk about dynamic grammars and their associated tools.

One of the main tools for both static and dynamic grammar is a parser. It is the program that will interpret the input into whatever usage it is meant for. Most of the time, a parser will transform the input into another similarly structured language. It can be a storage inside objects or memory, or compiled into another format, or even just for syntax coloration. Since a lot of usage requires the same kind of function, a new kind of tool emerged to make the creation of a parser simpler. We call those tools parser or compiler generators (Paulson 1982). They take a grammar description as input and gives the program of a parser of the generated language as an output.

For dynamic grammar, these tools can get more complicated. There are a few ways a grammar can become dynamic. The most straightforward way to make a parser dynamic is to introduce code in the rule handling that will tweak variables affecting the parser itself (Souto *et al.* 1998). This allows for handling context in CFG without needing to rewrite the grammar.

Another kind of dynamic grammar is grammar that can modify themselves. In order to do this a grammar is valuated with reified objects representing parts of itself (Hutton and Meijer 1996). These parts can be modified dynamically by rules as the input gets parsed (Renggli *et al.* 2010; Alessandro and Piumarta 2007). This approach uses Parsing Expression Grammars (PEG)(Ford 2004) with Packrat parsing that Packrat parsing backtracks by ensuring that each production rule in the grammar is not tested more than once against each position in the input stream (Ford 2002). While PEG is easier to implement and more efficient in practice than their classical counterparts (Loff *et al.* 2018; Henglein and Rasmussen 2017), it offset the computation load in memory making it actually less efficient in general (Becket and Somogyi 2008).

Some tools actually just infer entire grammars from inputs and software (Höschele and Zeller 2017; Grünwald 1996). However, these kinds of approaches require a lot of input data to perform well. They also simply provide the grammar after expensive computations.

1.1.2.3 Description Logics

On of the most standard and flexible way of representing knowledge for databases is by using ontologies. They are based mostly on the formalism of Description Logics (DL). It is based on the notion of classes (or types) as a way to make the knowledge hierarchically structured. A class is a set of individuals that are called instances of the classes. Classes got the same basic properties as sets but can also be constrained with logic formula. Constraints can be on anything about the class or its individuals. Knowledge is also encoded in relations that are predicates over attributes of individuals.

It is common when using DLs to store statements into three boxes (Baader 2003):

- The TBox for terminology (statements about types)
- The RBox for rules (statements about properties) (Bürckert 1994)
- The ABox for assertions (statements about individual entities)

These are used mostly to separate knowledge about general facts (intentional knowledge) from specific knowledge of individual instances (extensional knowledge). The extra RBox is for "knowhow" or knowledge about entity behavior. It restricts usages of roles (properties) in the ABox. The terminology is often hierarchically ordered using a subsumption relation noted $\subseteq$. If we represent classes or type as a set of individuals then this relation is akin to the subset relation of set theory.

There are several versions and extensions of DL. They all vary in expressivity. Improving the expressivity of DL system often comes at the cost of less efficient inference engines that can even become undecidable for some extensions of DL.

1.1.3 Ontologies and their Languages

Most AI problem needs a way to represent data. The classical way to represent knowledge has been more and more specialized for each AI community. Each their Domain Specific Language (DSL) that neatly fit the specific use it is intended to do. There was a time when the branch of AI wanted to unify knowledge description under the banner of the "semantic web". From numerous works, a repeated limitation of the "semantic web" seems to come from the languages used. In order to guarantee performance of generalist inference engines, these languages have been restricted so much that they became quite complicated to use and quickly cause huge amounts of recurrent data to be stored because of some forbidden representation that will push any generalist inference engine into undecidability.

The most basic of these languages is perhaps RDF Turtle (Beckett and Berners-Lee 2011). It is based on triples with an XML syntax and has a graph as its knowledge structure (Klyne and Carroll 2004). A RDF graph is a set of RDF triples $\langle sub, pro, obj\rangle$ which fields are respectively called subject, property and object. It can also be seen as a partially labeled directed graph (V, E) with V being the set of RDF nodes and E being the set of edges. This graph also comes with an incomplete label $\phi^0 : (V \cup E) \to L^{URI}_{String}$ relation. Nodes without an URI are called blank nodes. It is important that, while not named, blank nodes have a distinct internal identifier from one another that allows to differentiate them.

Built on top of RDF, the W3C recommended another standard called OWL. It adds the ability to have hierarchical classes and properties along with more advanced description of their arrity and constraints. OWL is, in a way, more expressive than RDF (Van Harmelen *et al.* 2008, vol. 1 p825). It adds most formalism used in knowledge representation and is widely used and interconnected. OWL comes in three versions: OWL Lite, OWL DL and OWL Full. The lite version is less advanced but its inference is decidable, OWL DL contains all notions of DL and the full version contains all features of OWL but is strongly undecidable.

The expressivity can also come from a lack of restriction. If we allow some freedom of expression in RDF statements, its inference can quickly become undecidable (Motik 2007). This kind of extremely permissive language is better suited for specific usage for other branches of AI. Even with this expressivity, several works still deem existing ontology system as not expressive enough, mostly due to the lack of classical constructs like lists, parameters and quantifiers that don't fit the triple representation of RDF.

One of the ways which have been explored to overcome these limitations is by adding a 4th field in RDF. This field is meant for context and annotations. This field is used for information about any statement

Table 1.6: List of classical symbols and syntax for self.

Symbol	Description
$\mathcal{D}, P, \mathbb{Q}, S, T, \mathbb{U}$	Sets for domains, properties, quantifiers, statements, types and universe.
$\mu^{\pm}$	Meta-relation for abstraction ($+$) and reification ($-$).
ν	Name relation.
ρ	Parameter relation.

represented as a triple, such as access rights, beliefs and probabilities, or most of the time the source of the data (Tolksdorf *et al.* 2004). One of the other uses of the fourth field of RDF is to reify statements (Hernández *et al.* 2015). Indeed by identifying each statement, it becomes possible to efficiently for statements about statements.

A completely different approach is done by Hart and Goertzel (2008) in his framework for Artificial General Intelligence (AGI) called OpenCog. The structure of the knowledge is based on a rhizome, a collection of trees, linked to one another. This structure is called Atomspace. Each vertex in the tree is an atom, leaf-vertexes are nodes, the others are links. Atoms are immutable, indexed objects. They can be given values that can be dynamic and, since they are not part of the rhizome, are an order of magnitude faster to access. Atoms and values alike are typed.

The goal of such a structure is to be able to merge concepts from widely different domains of AI. The major drawback being that the whole system is very slow compared to pretty much any domain specific software.

1.2 Self

As we have seen, most existing knowledge description systems have a common drawback: they are static. This means that they are either too inefficient or too specific. To fix this issue, a new knowledge representation system must be presented. The goal is to make a minimal language framework that can adapt to its use to become as specific as needed. If it becomes specific is must start from a generic base. Since that base language must be able to evolve to fit the most cases possible, it must be neutral and simple.

To summarize, that framework must maximize the following criteria:

1. **Neutral**: Must be independent from preferences and be localization.
2. **Permissive**: Must allow as many data representation as possible.
3. **Minimalist**: Must have the minimum number of base axioms and as little native notions as possible.
4. **Adaptive**: Must be able to react to user input and be as flexible as possible.

In order to respect these requirements, we developed a framework for knowledge description. This Structurally Expressive Language Framework (SELF) is our answer to these criteria. SELF is inspired by RDF Turtle and Description Logic.

1.2.1 Knowledge Structure

SELF extends the RDF graphs by adding another label to the edges of the graph to uniquely identify each statement. This basically turns the system into a quadruple storage even if this forth field is transparent to the user.

Axiom (Structure)**.** A SELF graph is a set of statements that transparently include their own identity. The closest representation of the underlying structure of SELF is as follows:

$$g_{\mathbb{U}} = (\mathbb{U}, S) : S = \{s = \langle sub, pro, obj \rangle : s \in \mathcal{D} \vdash s \wedge \mathcal{D}\}$$

with:

- $sub, obj \in \mathbb{U}$ being entities representing the *subject* and *object* of the *statement* s,
- $pro \in P$ being the *property* of the statement s,
- $\mathcal{D} \subset S$ is the *domain* of the *world* $g_{\mathbb{U}}$,
- $S, P \subset \mathbb{U}$ with S the set of statements and P the set of properties,

This means that the world $g_{\mathbb{U}}$ is a graph with the set of entities $\mathbb{U}$ as vertices and the set of statements S as edges. This model also suppose that every statement s must be true if they belong to the domain $\mathcal{D}$. This graph is a directed 3-uniform hypergraph.

Since sheaves are a representation of hypergraphs, we can encode the structure of SELF into a sheaf-like form. Each seed is a statement, the germ being the statement vertex. It is always accompanied of an incoming connector (its subject), an outgoing connector (its object) and a non-directed connector (its property). The sections are domains and must be coherent. Each statement, along with its property, makes a stalk as illustrated in figure 1.8.

Figure 1.8: Projection of a statement from the SELF to RDF space.

The difference with a sheaf is that the projection function is able to map the pair statement-property into a labeled edge in its projection space. We map this pair into a classical labeled edge that connects the subject to the object of the statement in a directed fashion. This results in the projected structure being a correct RDF graph.

1.2.1.1 Consequences

The base knowledge structure is more than simply convenience. The fact that statements have their own identity, changes the degrees of freedom of the representation. RDF has a way to represent reified statements that are basically blank nodes with properties that are related to information about the subject, property and object of a designated statement. The problem is that such statements are very differently represented and need 3 regular statements just to define. Using the fourth field, it becomes possible to make statements about *any* statements. It also becomes possible to express modal logic about statements or to express, various traits like the probability or the access rights of a statement.

The knowledge structure holds several restrictions on the way to express knowledge. As a direct consequence, we can add several theorems to the logic system underlying SELF. The axiom of Structure is the only axiom of the system.

Theorem 1 (Identity). *Any entity is uniquely distinct from any other entity.*

This theorem comes from the axiom of Extensionality of ZFC. Indeed it is stated that a set is a unordered collection of distinct objects. Distinction is possible if and only if intrinsic identity is assumed. This notion of identity entails that a given entity cannot change in a way that would alter its identifier.

Theorem 2 (Consistency). *Any statement in a given domain is consistent with any other statements of this domain.*

Consistency comes from the need for a coherent knowledge database and is often a requirement of such constructs. This theorem also is a consequence of the axiom of Structure: $s \in \mathcal{D} \vdash s \wedge \mathcal{D}$.

Theorem 3 (Uniformity). *Any object in SELF is an entity. Any relations in SELF are restricted to $\mathbb{U}$.*

This also means that all native relations are closed under $\mathbb{U}$. This allows for a uniform knowledge database.

1.2.1.2 Native properties

Theorem 1 lead to the need for two native properties in the system : *equality* and *name*.

The **equality relation** $=: \mathbb{U} \to \mathbb{U}$, behaves like the classical operator. Since the knowledge database will be expressed through text, we also need to add an explicit way to identify entities. This identification is done through the **name relation** $\nu : \mathbb{U} \to L_{String}$ that affects a string literal to some entities. This lead us to introduce literals into SELF that is also entities that have a native value.

The axiom of Structure puts a type restriction on property. Since it compartments $\mathbb{U}$ using various named subsets, we must adequately introduce an explicit type system into SELF. That type system requires a **type relation** (named using the colon) $: : \mathbb{U} \to T$. That relation is complete as all entities have a type. Theorem 3 causes the set of entities to be universal. Type theory, along with Description Logic (DL), introduces a **subsumption relation** $\subseteq : T \to T$ as a partial ordering relation to the types. Since types can be seen as sets of instances we simply use the the subset relation from set theory. In our case, the entity type is the greatest element of the lattice formed by the set of types with the subsumption relation $(T, \subseteq)$.

The theorem 3 also allows for some very interesting meta-constructs. That is why we also introduce a signed **Meta relation** $\mu^+ : \mathbb{U} \to D$ with $\mu^- = (\mu^+)^{-1}$. This allows to create domain from certain entities and to encapsulate domains into entities. μ^- is for reification and μ^+ is for abstraction. This Meta relation also allows to express value of entities, like lists or various containers.

To fulfill the principle of adaptability and in order to make the type system more useful, we introduce the **parameter relation** $\rho : \mathbb{U} \to \mathbb{U}$. This relation affects a list of parameters, using the meta relation, to some parameterized entities. This also allows for variables in statements.

Since axiom of Structure gives the structure of SELF a hypergraph shape, we must port some notions of graph theory into our framework. Introducing the **statement relation** $\phi : S \to \mathbb{U}$ reusing the same symbol as for the adjacency and incidence relation of graphs. This isn't a coincidence as this relation has the same properties. For example, $\phi^-(s)$ gives the subject of a statement s. Respectively, ϕ^+ and ϕ^0 give the object and property of any statement. For adjacencies, ϕ^- and ϕ^+ can give the set of statements any entity is respectively the object and subject of. For any property pro, the notation $\phi^0(pro)$ gives the set of statements using this property. This allows us to port all the other notions of graphs using this relation as a base.

In figure 1.9, we present all the native relations along with their domains and most subsets of $\mathbb{U}$.

Figure 1.9: Venn diagram of subsets of $\mathbb{U}$ along with their relations. Dotted lines mean that the sets are defined a subset of the wider set.

1.2.2 Syntax

Since we need to respect the requirements of the problem, the RDF syntax cannot be used to express the knowledge. Indeed, RDF states native properties as English nodes with a specific URI that isn't neutral. It also isn't minimalist since it uses an XML syntax so verbose that it is not used for most examples in the documents that defines RDF because it is too confusing and complex (W3C 2004a; W3C 2004b). The XML syntax is also quite restrictive and cannot evolve dynamically to adapt to the usage.

So we need to define a new language that is minimalist and neutral. At the same time the language must be permissive and dynamic. These two goals are incompatible and will end up needing different solutions. So the solution to the problem is to actually define two languages that fit the criteria : one minimalist and one adaptive. The issue is that we need not make a user learn two languages and the second kind of language must be very specific and that violates the principle of neutrality we try to respect.

The only solution is to make a mechanism to adapt the language as it is used. We start off with a simple framework that uses a grammar.

The description of $\mathfrak{g}_0$ is pretty straightforward: it mostly is just a triple representation separated by whitespaces. The goal is to add a minimal syntax consistent with the axiom of Structure. In listing 1.1, we give a simplified version of $\mathfrak{g}_0$. It is written in a pseudo-BNF fashion, which is extended with the classical repetition operators `*` and `+` along with the negation operator `~`. All tokens have names in uppercase. We also add the following rule modifiers:

- `<~name>` are ignored for the parsing. However, the tokens are consumed and therefore acts like separators for the other rules.
- `<?name>` are inferred rules and tokens. They play a key role for the process of derivation explained in section 1.2.3.

```
1 <~COMMENT: <INLINE: "//" (~["\n", "\r"])*>
2 | <BLOCK: "/*" (~["*/"])*> > //Ignored
3 <~WHITE_SPACE: " "|"\t"|"\n"|"\r"|"\f">
4 <LITERAL: <INT> | <FLOAT> | <CHAR> | <STRING>> //Java definition
5 <ID: <TYPE: <UPPERCASE>(<LETTERS>|<DIGITS>)* >
6 | <ENTITY:  <LOWERCASE>(<LETTERS>|<DIGITS>)*>
7 | <SYMBOL: (~[<LITERALS>, <LETTERS>, <DIGITS>])*>>
8
9 <worselfld> ::= <first> <statement>* <EOF>
10 <first> ::= <subject> <?EQUAL> <?SOLVE> <?EOS>
11 <statement> ::= <subject> <property> <object> <EOS>
12 <subject> ::= <entity>
13 <property> ::= <ID> | <?meta_property>
14 <object> ::= <entity>
15 <entity> ::= <ID> | <LITERAL> | <?meta_entity>
```

Listing 1.1: Simplified pseudo-BNF description for basic SELF.

In order to respect the principle of neutrality, the language must not suppose of any regional predisposition of the user. There are few exceptions for the sake of convenience and performance. The first exception is that the language is meant to be read from left to right and have an occidental biased `subject verb object` triple description. Another exception is for liberals that use the same grammar as in classical Java. This means that the decimal separator is the dot (`.`). This could be fixed in later version using dynamic definitions (see section 1.3.1).

Even if sticking to the ASCII subset of characters is a good idea for efficiency, SELF can work with UTF-8 and exploits the Unicode Character Database (UCD) for its token definitions (Unicode Consortium 2018a). This means that SELF comes keywords free and that the definition of each symbol is left to the user. Each notion and symbol is inferred (with the exception of the first statement which is closer to an imposed configuration file).

In $\mathfrak{g}_0$, the first two token definitions are ignored. This means that comments and white-spaces will act as separation and won't be interpreted. Comments are there only for convenience since they do not serve any real purpose in the language. It was arbitrarily decided to use Java-style comments. White-spaces are defined against UCD's definition of the separator category `Z&` (see Unicode Consortium 2018b, chap. 4).

Line 4 uses the basic Java definition for liberals. In order to keep the independence from any natural language, boolean laterals are not natively defined (since they are English words).

Another aspect of that language independence is found starting at line 5 where the definitions of `<UPPERCASE>`, `<LOWERCASE>`, `<LETTERS>` and `<DIGITS>` are defined from the UCD (respectively categories `Lu`, `Ll`, `L&`, `Nd`). This means that any language's upper case can be used in that context. For performance and simplicity reasons we will only use ASCII in our examples and application.

The rule at line 1 is used for the definition of three tokens that are important for the rest of the input. `<EQUAL>` is the symbol for equality and `<SOLVE>` is the symbol for the *solution quantifier* (and also the language pendant of μ^-). The most useful token `<EOS>` is used as a statement delimiter. This rule also permits the inclusion of other files if a string literal is used as a subject. The underlying logic of this first statement will be presented in section 1.2.5.1.

At line 11, we can see one of the most defining features of $\mathbb{g}_0$: statements. The input is nothing but a set of statements. Each component of the statements are entities. We defined two specific rules for the subject and object to allow for eventual runtime modifications. The property rule is more restricted in order to guarantee the non-ambiguity of the grammar.

1.2.3 Dynamic Grammar

The syntax we described is only valid for $\mathbb{g}_0$. As long as the input is conforming to these rules, the framework keeps the minimal behavior. In order to access more features, one needs to break a rule. We add a second outcome to handling with violations : **derivation**. There are several kinds of possible violations that will interrupt the normal parsing of the input :

- Violations of the `<first>` statement rule : This will cause a fatal error.
- Violations of the `<statement>` rule : This will cause a derivation if an unexpected additional token is found instead of `<EOS>`. If not enough tokens are present, a fatal error is triggered.
- Violations of the secondary rules (`<subject>`, `<entity>`, ...) : This will cause a fatal error except if there is also an excess of token in the current statement which will cause derivation to happen.

Derivation will cause the current input to be analyzed by a set of meta-rules. The main restriction of these rules is given in $\mathbb{g}_0$: each statement must be expressible using a triple notation. This means that the goal of the meta-rules is to find an interpretation of the input that is reducible to a triple and to augment $\mathbb{g}_0$ by adding an expression to any `<meta_*>` rules. If the input has fewer than 3 entities for a statement then the parsing fails. When there is extra input in a statement, there is a few ways the infringing input can be reduced back to a triple.

1.2.3.1 Containers

The first meta-rule is to infer a container. A container is delimited by at least a left and right delimiter (they can be the same symbol) and an optional middle delimiter. We infer the delimiters using the algorithm 1.

The function sortedDelimiters at line 11 is used to generate every ordered possibility and sort them using a few criteria. The default order is possibilities grouped from left to right. All coupled delimiters that are mirrors of each other following the UCD are preferred to other possibilities.

Checking the result of the choice is very important. At line 12 a function checks if the delimiters allow for triple reduction and enforce restrictions. For example, a property cannot be wrapped in a container (except if part of parameters). This is done in order to avoid a type mismatch later in the interpretation.

Algorithm 1 Container meta-rule

```
function container(Token current)
    lookahead(current, EOS)                              ▷ Populate all tokens of the statement
    for all token in horizon do
        if token is a new symbol then delimiters.append(token)
    if length(delimiters) <2 then
        if coherentDelimiters(horizon, delimiters[0]) then
            inferMiddle(delimiters[0])                   ▷ New middle delimiter in existing containers
            return Success
        return Failure
    while length(delimiters) > 0 do
        for all (left, middle, right) in sortedDelimiters(delimiters) do
            if coherentDelimiters(horizon, left, middle, right) then
                inferDelimiter(left, right)
                inferMiddle(middle)                      ▷ Ignored if null
                delimiters.remove(left, middle, right)
                break
        if length(delimiters) stayed the same then return Success
    return Success
```

Once the inference is done, the resulting calls to inferDelimiter will add the rules listed in listing 1.2 to $\mathfrak{g}_0$. This function will create a `<container>` rule and add it to the definition of `<meta_entity>`. Then it will create a rule for the container named after the UCD name of the left delimiter (searching in the `NamesList.txt` file for an entry starting with “left“ and the rest of the name or defaulting to the first entry). Those rules are added as a conjunction list to the rule `<container>`. It is worthy to note that the call to inferMiddle will add rules to the token `<MIDDLE>` independently from any container and therefore, all containers share the same pool of middle delimiters.

```
1 <meta_entity> ::= <container>
2 <container> :: = <parenthesis> | …
3 <parentesis> ::= "(" [<naked_entity>] (<?MIDDLE> <naked_entity>)* ")"
4 <naked_entity> ::= <statement> | <entity>
```

Listing 1.2: Rules added to the current grammar for handling the new container for parenthesis

The rule at line 4 is added once and enables the use of meta-statements inside containers. It is the language pendant of the μ^+ relation, allowing to wrap abstraction in a safe way.

1.2.3.2 Parameters

If the previous rule didn’t fix the parsing of the statement, we continue with the following meta-rule. Parameters are extra containers that are used after an entity. Every container can be used as parameters. We detail the analysis in algorithm 2.

The goal is to match extra containers with the preceding named entity. The container is then combined with the preceding entity into a parameterized entity.

The call to inferParameter will add the rule in listing 1.3, replacing `<?container>` with the name of the container used (for example `<parenthesis>`).

Algorithm 2 Parameter meta-rule

```
function parameter(Entity[] statement)
    reduced = statement
    while length(reduced) >3 do
        for i from 0 to length(reduced) - 1 do
            if name(reduced[i]) not null and
type(reduced[i+1]) = Container and
coherentParameters(reduced, i) then
                param = inferParameter(reduced[i], reduced[i+1])
                reduced.remove(reduced[i], reduced[i+1])
                reduced.insert(param, i)                    ▷ Replace parameterized entity
                break
        if length(statement) stayed the same  then return Success
    return Failure
```

```
<meta_entity> ::= <ID> <?container>
<meta_property> ::= <ID> <?container>
```

Listing 1.3: Rules added to the current grammar for handling parameters

1.2.3.3 Operators

A shorthand for parameters is the operator notation. It allows to affect a single parameter to an entity without using a container. It is most used for special entities like quantifiers or modificators. This is why, once used, the parent entity takes a polymorphic type, meaning that type inference will not issue errors for any usage of them. Details of the way the operators are reduced is exposed in algorithm 3.

Algorithm 3 Operator meta-rule

```
function operator(Entity[] statement)
    reduced = statement
    while length(reduced) >3 do
        for i from 0 to length(reduced) - 1 do
            if ν(reduced[i]) not null and
ν(reduced[i+1]) not null and
(ν(reduced[i]) is a new symbol or
reduced[i] has been parameterized before) and
coherentOperator(reduced, i) then
                op = inferOperator(reduced[i], reduced[i+1])
                reduced.remove(reduced[i], reduced[i+1])
                reduced.insert(op, i)                       ▷ Replace parameterized entity
                break
        if length(statement) stayed the same  then return Success
    return Failure
```

From the call of inferOperator, comes new rules explicated in listing 1.4. The call also adds the operator entity to an inferred token `<OP>`.

```
<meta_entity> ::= <?OP> <ID>
```

```
2 <meta_property> ::= <?OP> <ID>
```

Listing 1.4: Rules added to the current grammar for handling operators

If all meta-rules fail, then the parsing fails and returns an error to the user.

1.2.4 Contextual Interpretation

While parsing another important part of the processing is done after the success of a grammar rule. The grammar in SELF is valuated, meaning that each rule has to return an entity. A set of functions are used to then populate the database with the right entities or retrieve an existing one that correspond to what is being parsed.

When parsing, the rules `<entity>` and `<property>` will ask for the creation or retrieval of an entity. This mechanism will use the name of the entity and its type to retrieve an entity with the same name in a given scope.

1.2.4.1 Naming and Scope

When parsing an entity by name, the system will first request for an existing entity with the same name. If such an entity is retrieved, it is returned instead of creating a new one. The validity of a name is limited by the notion of scope.

A scope is the reach of an entity's direct influence. It affects the naming relation by removing variable's names. Scopes are delimited by containers and statements. This local context is useful when wanting to restrict the scope of the declaration of an entity. The main goal of such restriction is to allow for a similar mechanism as the RDF namespaces. This also makes the use of variable (RDF blank nodes) possible.

The scope of an entity has three special values :

- Variable: This scope restricts the scope of the entity to only the other entities in its scope.
- Local: This scope means that the parsing is still populating the scope of the entity. Its scope is limited to the currently parsing statement.
- Global: This scope means the name has no scope limitation.

The scope of an entity also contains all its parent entities, meaning all containers or statement the entity is part of. This is used when choosing between the special values of the scope. The process is detailed in algorithm 4.

The process happens for each entity created or requested by the parser. If a given entity is part of any other entity, the enclosing entity is added to its scope. When an entity is enclosed in any entity while already being a parameter of another entity, it becomes a variable.

1.2.4.2 Instanciation identification

When a parameterized entity is parsed, another process starts to identify if a compatible instance already exists. From theorem 1, it is impossible for two entities to share the same identifier. This makes mandatory to avoid creating an entity that is equal to an existing one. Given the order of which parsing is done, it is not always possible to determine the parameter of an entity before its creation. In that case a later examination will merge the new entity onto the older one and discard the new identifier.

Algorithm 4 Determination of the scope of an entity

```
function inferScope(Entity e)
    Entity[] reach = []
    if : (e) = S then
        for all i ∈ ϕ(e) do reach.append(inferVariable(i))            ▷ Adding scopes nested in statement e
    for all i ∈ μ⁻(e) do reach.append(inferVariable(i))               ▷ Adding scopes nested in container e
    if ∃ρ(e) then
        Entity[] param = inferScope(ρ(e))
        for all i ∈ param do param.remove(inferScope(i))              ▷ Remove duplicate scopes from parameters
        for all i ∈ param do reach.append(inferVariable(i))           ▷ Adding scopes from paramters of e
    scope(e) ← reach
    if GLOBAL ∉ scope(e) then scope(e) ← scope(e) ∪{LOCAL}
return reach
function inferVariable(Entity e)
    Entity[] reach = []
    if LOCAL ∈ scope(e) then
        for all i ∈ scope(e) do
            if ∃e⁺ ∈ 𝕌 : ρ(p) = i then                                 ▷ e is already a parameter of another entity e⁺
                scope(e) ← scope(e) \{LOCAL}
                scope(e⁺) ← scope(e⁺) ∪ scope(e)
                scope(e) ← scope(e) ∪{VARIABLE, p}
        reach.append(e)
        reach.append(scope(e))
return reach
```

1.2.5 Structure as a Definition

The derivation feature on its own does not allow to define most of the native properties. For that, one needs a light inference mechanism. This mechanism is part of the default inference engine. This engine only works on the principle of structure as a definition. Since all names must be neutral from any language, that engine cannot rely on regular mechanisms like configuration files with keys and values or predefined keywords.

To use SELF correctly, one must be familiar with the native properties and their structure or implement their own inference engine to override the default one.

1.2.5.1 Quantifiers

What are quantifiers? In SELF they differ from their mathematical counterparts. The quantifiers are special entities that are meant to be of a generic type that matches any entities including quantifiers. There are infinitely many quantifiers in SELF but they are all derived from a special one called the *solution quantifier*. We mentioned it briefly during the definition of the grammar $\mathfrak{g}_0$. It is the language pendant of μ^- and is used to extract and evaluate reified knowledge.

For example, the statement `bob is <SOLVE>(x)` will give either a default container filled with every value that the variable `x` can take or if the value is unique, it will take that value. If there is no value it will default to `<NULL>`, the exclusion quantifier.

How are these other quantifiers defined? We use a definition akin to Lindstöm quantifiers (1966) which is a generalization of counting quantifiers (Gradel *et al.* 1997). Meaning that a quantifier is defined as a constrained range over the quantified variable. We suppose five quantifiers as existing in SELF as native entities.

- The **solution quantifier** <`SOLVE`> noted § in classical mathematics, makes a query that turns the expression into the possible range of its variable.
- The **universal quantifier** <`ALL`> behaves like ∀ and forces the expression to affect every possible value of its variable.
- The **existential quantifier** <`SOME`> behaves like ∃ and forces the expression to match *at least one* value for its variable.
- The **uniqueness quantifier** <`ONE`> behaves like !∃ and forces the expression to match *exactly one* value for its variable.
- The **exclusion quantifier** <`NULL`> behaves like ¬∃ and forces the expression to match none of the value of its variable.

The last four quantifiers are inspired from Aristotle's square of opposition (D'Alfonso 2011).

Figure 1.10: Aristotle's square of opposition

In SELF, quantifiers are not always followed by a quantified variable and can be used as a value. In that case the variable is simply anonymous. We use the exclusion quantifier as a value to indicate that there is no value, sort of like `null` or `nil` in programming languages.

In listing 1.5, we present an example file that is meant to define most of the useful native properties along with default quantifiers.

```
* =? ;
?(x) = x; //Optional definition
?~ = { };
?_ ~(=) ~;
?!_ = {_};

(*e, !T) : (e :: T); *T : (T :: Type);
*T : (Entity / T);
```

```
9
10 :: :: Property(Entity, Type);
11 (___) :: Statement;
12 (~, !, _, *) :: Quantifier;
13 ( )::Group;
14 { }::Set;
15 [ ]::List;
16 < >::Tuple;
17 Collection/(Set,List,Tuple);
18 0 :: Integer; 0.0::Float;
19 '\0'::Character; ""::String;
20 Literal/(Boolean, Integer, Float, Character, String);
21
22 (*e, !(s::String)) : (e named s);
23 (*e(p), !p) : (e param p);
24 *(s p o):(((s p o) subject s),((s p o) property p),((s p o) object o));
```

Listing 1.5: The default lang.w file.

At line 1, we give the first statement that defines the solution quantifier's symbol. The reason this first statement is shaped like this is that global statements are always evaluated to be a true statement. This means that anything equaling the solution quantifier at this level will be evaluated as a domain. If it is a string literal, then it must be either a file path or URL or a valid SELF expression. If it is a single entity then it becomes synonymous to the entire SELF domain and therefore contains everything. We can infer that it becomes the universal quantifier.

All statements up to line 5 are quantifiers definitions. On the left side we got the quantifier symbol used as a parameter to the solution quantifier using the operator notation. On the right we got the domain of the quantifier. The exclusive quantifier has as a range the empty set. For the existential quantifier we have only a restriction of it not having an empty range. At last, the uniqueness quantifier got a set with only one element matching its variable (noting that anonymous variables doesn't match necessarily other anonymous variables in the same statement).

1.2.5.2 Inferring Native Properties

All native properties can be inferred by structure using quantified statements. Here is the structural definition for each of them:

- $=$ (at line 1) is the equality relation given in the first statement.
- $\subseteq$ (at line 8) is the first property to relate a particular type relative to all types. That type becomes the entity type.
- μ^- (at line 1) is the solution quantifier discussed above given in the first statement.
- μ^+ is represented using containers.
- ν (at line 22) is the first property affecting a string literal uniquely to each entity.
- ρ (at line 23) is the first property to effect to all entities a possible parameter list.
- $:$ (at line 7) is the first property that matches every entity to a type.
- ϕ (at line 24) is the first property to match for all statements:
  - ϕ^- its subject,
  - ϕ^0 its property,
  - ϕ^+ its object.

We limit the inference to one symbol to make it much simpler to implement and to retrieve but, except for false positives, there are no reason it should not be possible to define several notations for each relation.

1.2.6 Extended Inference Mechanisms

In this section we present the default inference engine. It has only a few functionalities. It isn't meant to be universal and the goal of SELF is to provide a framework that can be used by specialists to define and code exactly what tools they need.

Inference engines need to create new knowledge but this knowledge shouldn't be simply merged with the explicit domain. Since this knowledge is inferred, it is not exactly part of the domain but must remain consistent with it. This knowledge is stored in a special scope dedicated to each inference engine. This way, inference engines can use defeasible logic or have dynamic inference from any knowledge insertion in the system.

1.2.6.1 Type Inference

Type inference works on matching types in statements. The main mechanism consists in inferring the type of properties in a restrictive way. Properties have a parameterized type with the type of their subject and object. The goal is to make that type match the input subject and object.

For that we start by trying to match the types. If the types differ, the process tries to reduce the more general type against the lesser one (subsumption-wise). If they are incompatible, the inference uses some light defeasible logic to undo previous inferences. In that case the types are changed to the last common type in the subsumption tree.

However, this may not always be possible. Indeed, types can be explicitly specified as a safeguard against mistakes. If that's the case, an error is raised and the parsing or knowledge insertion is interrupted.

1.2.6.2 Instanciation

Another inference tool is instantiation. Since entities can be parameterized, they can also be defined against their parameters. When those parameters are variables, it allows entities to be instantiated later.

This is a complicated process because entities are immutable. Indeed, parsing happens from left to right and therefore an entity is often created before all the instantiation information are available. Even harder are completion of definition in several separate statements. In all cases, a new entity is created and then the inference realize that it is either matching a previous definition and will need to be merged with the older entity or it is a new instance and needs all properties duplicated and instantiated.

This gives us two mechanisms to take into account: merging and instanciating.

Merging is pretty straightforward: the new entity is replaced with the old one in all of the knowledge graph. containers, parameterized entities, quantifiers and statements must be duplicated with the correct value and the original destroyed. This is a heavy and complicated process but seemingly the only way to implement such a case with immutable entities.

Instanciating is similar to merging but even more complicated. It starts with computing a relation that maps each variable that needs replacing with their grounded value. Then it duplicates all knowledge about the parent entity while applying the replacement map.

1.3 Perspectives

Listing the contributions there are a couple that didn't make the cut. It is mainly ideas or projects that were too long to check or implement and needed more time to complete. SELF is still a prototype, and even if the implementation seemed to perform well on a simple example, no benchmarks have been done on it. It might be good to make a theoretical analysis of OWL compared to SELF along with some benchmark results.

On the theoretical parts there are some works that seems worthy of exposure even if unfinished.

1.3.1 Literal definition using Peano's axioms

The only real exceptions to the axioms and criteria are the first statement, the comments and the liberals.

For the first statement, there is yet to find a way to express both inclusion, the equality relation and solution quantifier. If such a convenient expression exists, then the language can become almost entirely self described.

Comments can be seen as a special kind of container. The difficult part is to find a clever way to differentiate them from regular containers and to ignore their content in the regular grammar. It might be possible to at first describe their structure but then they become parseable entities and fail at their purpose.

Lastly, and perhaps the most complicated violation to fix: laterals. It is quite possible to define literals by structure. First we can define boolean logic quite easily in SELF as demonstrated by listing 1.6.

```
1 ~(false) = true;
2 ( false, true ) :: Boolean;
3 true =?; //conflicts with the first statement!
4 *a : ((a | true) = true);
5 *a : ((false | a) = a);
6 *a : ((a & false) = false);
7 *a : ((true & a) = a);
```

Listing 1.6: Possible definition of boolean logic in SELF.

Starting with line 1, we simply define the negation using the exclusive quantifier. From there we define the boolean type as just the two truth values. And now it gets complicated. We could either arbitrarily say that the false literal is always parameters of the exclusion quantifier or that it comes first on either first two statements but that would just violate minimalism even more. We could use the solution quantifier to define truth but that collides with the first statement definition. There doesn't seem to be a good answer for now.

From line 4 going on, we state the definition of the logical operators $\wedge$ and $\vee$. The problem with this is that we either need to make a native property for those operators or the inference to compute boolean logic will be terribly inefficient.

We can use Peano's axioms (1889) to define integers in SELF. The attempt at this definition is presented in listing 1.7.

```
0 :: Integer;
*n : (++(n) :: Integer);
```

```
(*m, *n) : ((m=n) : (++m = ++n));
*n : (++n ~= 0);
*n : ((n + 0) = n);
(*n, *m) : ((n + ++m)= ++(n + m));
*n : ((n × 0) = 0);
(*n, *m) : ((n × ++m) = (n + (n × m)));
```

Listing 1.7: Possible integration of the Peano axioms in SELF.

We got several problems doing so. The symbols * and / are already taken in the default file and so would need replacement or we should use the non-ASCII × and ÷ symbols for multiplication and division. Another more fundamental issue is as previously discussed for booleans: the inference would be excruciatingly slow or we should revert to a kind of parsing similar to what we have already under the hood. The last problem is the definition of digits and bases that would quickly become exceedingly complicated and verbose.

For floating numbers this turns out even worse and complicated and such a description wasn't even attempted for now.

The last part concerns textual laterals. The issue is the same as the one with comments but even worse. We get to interpret the content as literal value and that would necessitate a similar system as we already have and wouldn't improve the minimalist aspect of things much. Also we should define ways to escape characters and also to input escape sequences that are often needed in such case. And since SELF isn't meant for programming that can become very verbose and complex.

1.3.2 Advanced Inference

The inference in SELF is very basic. It could be improved a lot more by simply checking the consistency of the database on most aspects. However, such a task seems to be very difficult or very slow. Since that kind of inference is undecidable in SELF, it would be all a research problem just to find a performant inference algorithm.

Another kind of inference is more about convenience. For example, one can erase singlets (containers with a single value) to make the database lighter and easier to maintain and query.

1.3.3 Queries

We haven't really discussed quarries in SELF. They can be made using the main syntax and the solution quantifiers but efficiency of such queries is unknown. Making an efficient query engine is a big research project on its own.

For now a very simplified query API exists in the prototype and seems to perform well but further tests are needed to assess its scalability capacities.

2 General Planning Framework

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

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

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

2.1 Illustration

To illustrate how automated planners works, we introduce a typical planning problem called **block world**. In this example, a robot with one grabing arm tries to stack blocks on a table in a specific order. The arm is only capable of handling one block at a time. We suppose that the table is large enough so that all the blocks can be put on it without any stacks. Figure 2.1 illustrates how the operation works.

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

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

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

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

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

Figure 2.1: The block world setup.

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

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

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

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

However, In the next section we aim to create such a general planning formalism.

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

2.2 Formalism

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

Figure 2.2: An example of a solution to a planning problem.

Table 2.1: List of classical symbols and syntax for planning.

Symbol	Description
$F, \square, A$	Sets of fluents, states and actions.
$\otimes, \odot^{\pm}$	Sets of flaws and signed resolvers. Flaws have variants:
$\otimes^{\ddagger}$	• unsupported subgoal.
$\otimes^{\dagger}$	• causal threat to an existing causal link.
$\otimes^{\circlearrowleft}$	• cycle in the plan.
$\otimes^{*}$	• decomposition of a compound action.
$\otimes^{\curvearrowright}$	• alternative to an existing action.
$\otimes^{\bar{o}}$	• orphan action in the plan.
$\Pi, \mathbb{S}$	Sets of plans and search space.
$l \downarrow a$	Partial support of action a by the causal link l.
$\pi \downarrow a$	Full support of action a by plan π
$\prec, \succ$	Precedence and succession relation used for order.
$\mapsto^{*}$	General shortest path algorithm.
h	Search heuristic.
$\mathbb{p}$	Planning problem.
c	Constraints on the action.
¢	Cost of an action.
d	Duration of an action.
ω	Root operator.

2.2.1 Planning domain

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

2.2.1.1 Fluents

First, we need to define the smallest unit of knowledge in planning, the fluents.
Definition 10 (Fluent)**.** A planning fluent is a predicate $f(arg_1, arg_2, ..., arg_n) \in F$ where:

- f is a relation/function.
- $arg_{i\in[1,n]} \in \mathbb{U}$ are the arguments (possibly quantified).
- $n = |f|$ is the arity of f.

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

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

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

When expressing states, we need a formalism to express sets of fluents as formulaes.
Definition 11 (State)**.** A state is a set of fluent. It is provided with a truth value like a fluent and can behave like one. The truth value is the *conjunction* of all fluents within the state $\square \vdash \bigwedge_{f\in\square} f$ denoted by a small square $\square$. States can contain other states in which case their truth value is the *disjunction* of the member's truth value: $\square \vdash \bigvee_{\square'\in\square} \square'$. This creates an and/or tree with the branches being all *or vertices* except for the ones connecting to fluents that becomes *and vertices*. All leaves of the tree are fluents.

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

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

In such a case, both state $\square_1$ and $\square_2$ have their truth value being the conjunction of all their fluents. In order to express a disjunction, one has to combine states in the following way: $\square_3 = \{\square_1, \square_2\}$. In that case $\square_3$ is the root of the and/or tree and all its direct children are or vertices. The states $\square_1$ and $\square_2$ have their children as *and vertices* since they are fluents.

Another important part of the behavior of fluents is their ability to match and to being unified.

Matching is the relation $f_1 :_\square f_2$ that affects any pair of fluents $\langle f_1, f_2\rangle$ along with a context state $\square$ to another state containing the context augmented with unification constraints or $\emptyset$ if the two fluents are contradicting one another given the context. If the two fluents are equal or have a different function, they do not influence the context. Opposite fluents will always contradict. The quantified or variable arguments may cause contradiction or add a constraint into the context.

The actual unification relation $f_1 \vdash_\square f_2$ bind any matching fluents to another grounded fluent giving the constraints of the given context.

Both relations are generalized to states by merging the result of the following way:

Figure 2.3: Example of a state encoded as an and/or tree.

$$\vdash: (\square_r, \square_f) = \bigcup_{f_r \in \square_f, f_f \in \square_f} \vdash: (f_r, f_f, \square_a)$$

with, $\square_r$ being the reference state and $\square_f$ being the formula state. The state $\square_a$ is an accumulator that is result of the partial union of the previous matching iterations. This formula is the same for $:$ and $\vdash$. It is interesting to note that the relations doesn't need an additional context and share the same definition domain $\square^2 \to \square$ when taking states as arguments.

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

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

We can express a few example of fluent matching:

- $held(noblock) : held(x) = \{x = noblock\}$
- $\neg held(x) : held(x) = \emptyset$
- $held(blockA) :_{\square_1} held(x) = \emptyset$
- $down(blockD) :_{\square_1} down(x) = \square_1 \cup \{x = blockD\}$
- $down(blockC) :_{\square_1} down(blockC) = \square_1$

Unification of fluents goes quite simply:

-

We also can present matching on states:

- $\square_1 : \square_2 = \emptyset$
- $\square_1 : \square_4(x) = \square_1 \cup \{x = blockC\}$

All these relations are used to check and apply actions.

2.2.1.2 Actions

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

Definition 12 (Action)**.** An action is a parametrized tuple $a(args) = \langle :, \vdash, c, ¢, d, \mathbb{P}, \Pi \rangle$ where:

- $:$ and $\vdash$ are states that are respectively the **preconditions and the effects** of the action.
- c is the state representing the **constraints**.
- ¢ is the intrisic **cost** of the action.
- δ is the intrinsic **duration** of the action.
- $\mathbb{P}$ is the prior **probability** of the action succeeding.
- Π is a set of **methods** that decompose the action into smaller simpler ones.

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

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

Actions are often represented as state operators that can be applied in given state to alter it. The application of actions is done by using the actions as relations $a : \square \to \square$ defined as follow: $a(\square) = \square \vdash_{\square : a} a$

$$a(\square) = \begin{cases} \varnothing, & \text{if } \square : a = \varnothing \\ \square \vdash a, & \text{otherwise} \end{cases}$$

2.2.1.3 Domain

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

Definition 13 (Domain)**.** A domain $\mathcal{D}$ is a set of **operators** which are fully lifted *actions*, along with all the relations and entities needed to describe their preconditions and effects.

2.2.2 Planning problem

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

2.2.2.1 Solution

Definition 14 (Partial Plan / Method)**.** A partially ordered plan is an *acyclic* directed graph $\pi = (A_\pi, E)$, with:

- A_π the set of **steps** of the plan as vertices. A step is an action belonging in the plan. A_π must contain an initial step a_π^0 and goal step a_π^* as convinience for certain planning paradigms.
- E the set of **causal links** of the plan as edges. We note $l = a_s \xrightarrow{\square} a_t$ the link between its source a_s and its target a_t caused by the set of fluents $\square$. If $\square = \varnothing$ then the link is used as an ordering constraint.

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

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

2.2.2.2 Problem

With this formalism, the problem is very simplified but still general.
Definition 15 (Problem)**.** The planning problem is defined as the **root operator** ω which methods are potential solutions of the problem. Its preconditions and effects are respectively used as initial state and goal description.

As actions are very general, it is interesting to make the problem and domain space homogenous by using an action to describe any problem.

Most of the specific notions of this framework are optionnal. Any planner using it will probably define what features it supports when compiling input domains and problems.

2.2.3 Planning algorithm

The general planning algorithm can be described as a guided exploration of a search space. The detailed structure of the search space as well as search iterators are dependant on the planning paradigm used.

2.2.3.1 Search space

Definition 16 (Planner)**.** A planning algorithm, often called planner, is an exploration of a search space $\mathbb{S}$ partially ordered by an iterator $\phi^+_{\mathbb{S}}$ guided by a heuristic h. From any problem $\mathbb{p}$ every planner can derive two informations imediately:

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

Formally the problem can be exprimed as a pathfinding problem in the dirrected graph $g_{\mathbb{S}}$ formed by the vertex set $\mathbb{S}$ and the adjacence function $\phi^+_{\mathbb{S}}$. The set of solutions is therefore expressed as:

$$\mathbb{S}^* = \left\{ \mathbb{s}^* : \langle \mathbb{s}_0, \mathbb{s}^* \rangle \in E_{\chi^+(g_{\mathbb{S}})} \land ?_{\mathbb{s}^*}) \right\}$$

In automated there are also other considerations about the search.

2.2.3.2 Diversity

Sometimes, it is necessary to find alternatives. Since re-planning from scratch is computationally demanding, it is better to find several solutions at once. This approach is called *diverse planning*. It aims to find k solutions that deviates from one another significantly. This simply make the process return when either it found k solutions or when it determined that $k > |\mathbb{S}^*|$.

2.2.3.3 Temporality

Another aspect of planning lies in its timming. Indeed sometimes acting needs to be done before a dealine and planning is useful only durring a finite timeframe. We add a predicate that specifies time constraints over algorithms $t : \mathbb{A} \to \mathbb{A}$. This constraint has three main type of application:

- t_{∞}: Optimal search without time limitation, finding the best solution everytime.
- $t_{\text{⏱}}$: Anytime search, finding a solution and improving its quality until stopped.
- $t_{\text{◔}}$: Real-time search, being able to give a solution in a given time even if it is an approximation.

2.2.3.4 General planner

A general planner $\mathbb{C}^*[\mathbb{S}, \phi^+_{\mathbb{S}}, h, \to](\mathcal{D}, \mathbb{p})$ is an algorithm that can plan any formalism of automated planning. It takes two set of parameters:

- **Formalism dependant parameters**
  - $\mathbb{S}$ the search space
  - $\phi^+_{\mathbb{S}}$ the search iterator
  - h the heuristic
  - $\to$ the problem transformation function
- **Domain depedant parameters**
  - $\mathcal{D}$ the *planning domain*
  - $\mathbb{p}$ the *planning problem*

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

Other informations must be added to any problem $\mathbb{p}$ in the form of constraints:

$$\big(t(\mathbb{C}) ⩙ |ℿ(\omega)| = k\big) \in c(\omega)$$

The value for k is extracted from the problem and the temporality is expressed using either $\wedge$, $\vee$ or $\wedge\bot\vee$ instead of $⩙$.

The transformation function $\mathbb{p} \to \langle \mathbb{s}_0, ?_{\mathbb{s}^*} \rangle$ gives the starting point $\mathbb{s}_0$ and the solution predicate $?_{\mathbb{s}^*}$. This predicate is derived from the problem description and its contraints.

For the algorithm itself, we simply use a parameterized instance of the K* algorithm (Aljazzar and Leue 2011, alg. 1). This algorithm uses the classical algorithm A* to explore the graph while using Dijkstra on some sections to find the k shortest paths. The parameters are as follow: $K^*(g_{\mathbb{S}}, \mathbb{s}_0, ?_{\mathbb{s}^*}, h)$. The solution predicate contains the expression of the restriction of k solutions, therefore, it superseeds the need for the k parameter. We also add the heuristic h to guide the A* part of the algorithm.

Of course this algorithm is merely an example of a general planner algorithm. Its efficiency hasn't been tested.

Figure 2.4: Schema of the full formalism.

2.3 Classical Formalisms

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

2.3.1 State-transition planning

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

This makes this kind of planning quite simple to instantiate from the general planner:

$$\mathbb{C}_{state} = \mathbb{C}^*\left[\square, \bigcup_{a\in A} a(\mathbb{s}), h, :(\omega), \vdash(\omega)\right]$$

For this formalism, we often set $k = 1$ and $t(\mathbb{C}_{state}) = t_{\otimes}$ as is customary in classical planning. It can also be specified as a backward search by inverting the application of a with a^{-1} and having the starting state as $:(\omega)$ and solution predicate as $\vdash(\omega)$.

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

Table 2.2: List of classical symbols and syntax for probabilities.

Symbol	Description
$\mathbb{P}(e)$	Probability of event e.
$\mathcal{O}$	Set of observations.
$\mid$	Reward function.

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

2.3.2 Plan space planning

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

$$\mathbb{C}_{psp} = \mathbb{C}^* \left[\Pi, \bigcup_{f \in \otimes(\mathbb{s})}^{r \in \odot_f^{\pm}} r(\mathbb{s}), h, \left(\{a_\pi^0, a_\pi^*\}, \{a_\pi^0 \to a_\pi^*\}\right), \otimes(\mathbb{s}) = \varnothing \right]$$

with a_π^0 and a_π^* being the intial and goal steps of the plan $\mathbb{s}_0$ such that $\vdash (a_\pi^0) =: (\omega)$ and $: (a_\pi^*) =\vdash (\omega)$. The iterator is all the possible resolutions of all flaws on any plan $\mathbb{s}$ and the solution predicate is true when the plan has no more flaws.

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

This approach usually can give a partial plan if we set $t(\mathbb{C}_{psp}) < t(\exists \pi \in \Pi \wedge ?_{\mathbb{s}}(\pi))$. This plan is not a solution but can eventually be engineered into having approximative properties relative to a solution.

2.3.3 Case based planning

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

$$\mathbb{C}_{cbp} = \mathbb{C}^* \left[\mathcal{L}^\pi, \odot(\mathbb{s}), h, \sigma^*(\mathcal{L}^\pi, \omega), \mathbb{s}(: (\omega)) \Downarrow\vdash (\omega)\right]$$

with $\mathcal{L}^\pi$ being the plan library. The planner selects efficiently a plan that fit the best with the intial and goal state of the problem. This plan is then repaired and validated iteratively. The problem with this approach is that it may be unable to find a valid plan or might need to populate and maintain a good plan library. For such case an auxiliary planner is used (preferably diverse with $k > 1$).

2.3.4 Probabilistic planning

FIXME Reward using rupee symbol |

Probabilistic planning tries to deal with uncertainty by generating a policy instead of a plan. The initial problem holds probability laws that govern the execution of any actions. It is sometimes accompagnated with a reward function instead of a deterministic goal.

$$\mathbb{C}_{prob} = \mathbb{C}^* \left[\Box \times A, \mathbb{s}_{+1} = \bigcup_{a \in A}^{a(\square) \neq \emptyset} \langle \pi_{\langle \square', a' \rangle \to a'(\square')} \sigma(\mathbb{s}_{+1}), a \rangle, h_{|}, \bigcup_{a \in A}^{a(:(\omega)) \neq \emptyset} \langle :\, (\omega), a \rangle, \square \vdash \omega \right]$$

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

2.3.5 Hierarchical planning

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

$$\mathbb{C}_{htn} = \mathbb{C}^* = \left[\Pi, \left(\sigma\{a \in A_{\mathbb{s}} : \Pi_a \neq \emptyset\} \to \sigma(\Pi_a) \right) (\mathbb{s}), h, \omega, \forall a \in A_{\mathbb{s}} : \Pi(a) = \emptyset \right]$$

2.4 Existing Languages and Frameworks

2.4.1 Classical

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

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

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

With time, the planning competitions became known under the name of International Planning Competitions (IPC) regularly hosted by the ICAPS conference. With each installment, the language evolved to address issues encountered the previous years. The current version of PDDL is 3.1 (Kovacs 2011). Its syntax, goes similarly as described in listing 2.1.

```
(define (domain <domain-name>)
  (:requirements :<requirement-name>)
  (:types <type-name>)
  (:constants <constant-name> - <constant-type>)
  (:predicates (<predicate-name> ?<var> - <var-type>))
  (:functions   (<function-name> ?<var> - <var-type>) - <function-type>)

  (:action <action-name>
```

```
        :parameters (?<var> - <var-type>)
        :precondition (and (= (<function-name> ?<var>) <value>) (<predicate-name> ?<var>))
        :effect
        (and (not (<predicate-name> ?<var>))
          (assign (<function-name> ?<var>) ?<var>)))
```

Listing 2.1: Simplified explanation of the syntax of PDDL.

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

For example, consider the classic block world domain expressed in listing 2.2. It uses a predicate to express whether a block is on the table because several blocks can be on the table at once. However it uses a 0-ary function to describe the one block allowed to be held at a time. The description of the stack of blocks is done with an unary function to give the block that is on top of another one. To be able to express the absence of blocks it uses a constant named `no-block`. All the actions described are pretty straightforward: `stack` and `unstack` make sure it is possible to add or remove a block before doing it and `pick-up` and `put-down` manages the handling operations.

```
(define (domain BLOCKS-object-fluents)
  (:requirements :typing :equality :object-fluents)
  (:types block)
  (:constants no-block - block)
  (:predicates (on-table ?x - block))
  (:functions   (in-hand) - block
        (on-block ?x - block) - block) ;;what is in top of block ?x

  (:action pick-up
        :parameters (?x - block)
        :precondition (and (= (on-block ?x) no-block) (on-table ?x) (= (in-hand) no-block))
        :effect
        (and (not (on-table ?x))
          (assign (in-hand) ?x)))

  (:action put-down
        :parameters (?x - block)
        :precondition (= (in-hand) ?x)
        :effect
        (and (assign (in-hand) no-block)
          (on-table ?x)))

  (:action stack
        :parameters (?x - block ?y - block)
        :precondition (and (= (in-hand) ?x) (= (on-block ?y) no-block))
        :effect
        (and (assign (in-hand) no-block)
         (assign (on-block ?y) ?x)))

  (:action unstack
        :parameters (?x - block ?y - block)
        :precondition (and (= (on-block ?y) ?x) (= (on-block ?x) no-block) (= (in-hand) no-block))
        :effect
```

```
        (and (assign (in-hand) ?x)
         (assign (on-block ?y) no-block))))
```

Listing 2.2: Classical PDDL 3.0 definition of the domain Block world

However, PDDL is far from an universal standard. Some efforts have been made to try and standardize the domain of automated planning in the form of optional requirements. The latest of the PDDL standard is the version 3.1 (Kovacs 2011). It has 18 atomic requirements as represented in figure 2.5. Most requirements are parts of PDDL that either increase the complexity of planning significantly or that require extra implementation effort to meet.

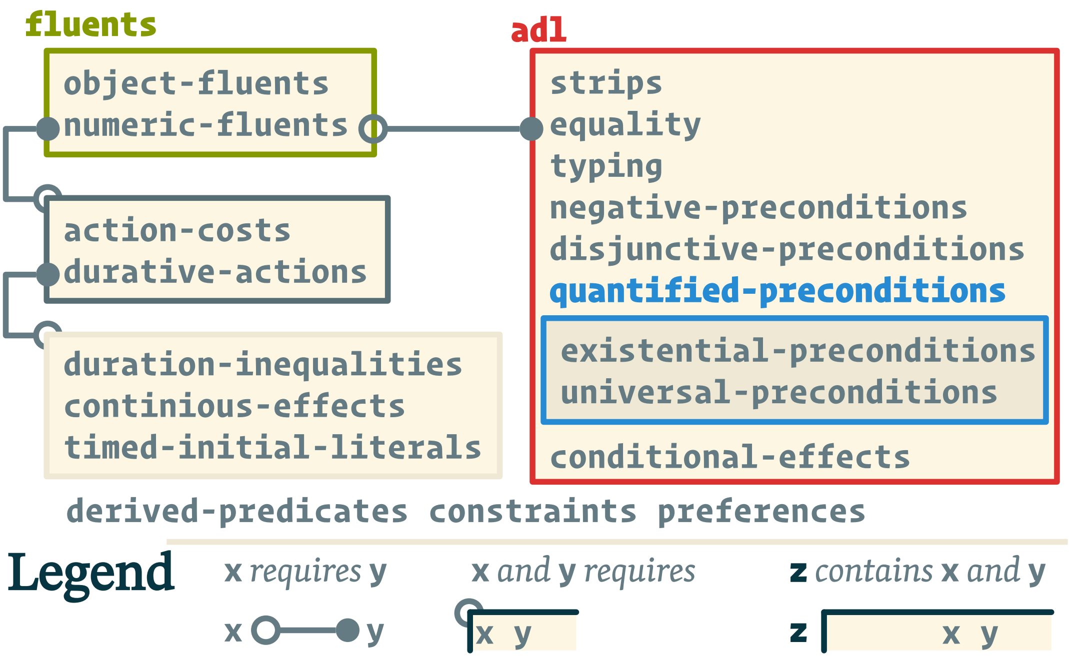

Figure 2.5: Dependancies and grouping of PDDL requirements.

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

Another problem is that PDDL isn't made to be used by planners to help with their planning process. Most planners will totally separate the compilation of PDDL before doing any planning, so much so that most planners of the latest IPC used a framework that translates PDDL into a useful form before planning, adding computation time to the planning process. The list of participating planners and their use of language is presented in table 2.3.

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

Table 2.3: Planners participating in the Classic track of the 2018 International Planning Competition (IPC). The table states whether the planner used a translation and a preprocessing system to handle PDDL. Most of the planners are based on FastDownward directly.

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

2.4.2 Temporality oriented

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

2.4.2.1 PDDL+

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

In listing 2.3, we reproduce an example from Fox and Long (2002). It shows the syntax of durative actions in PDDL+. The timed precondition are also available in PDDL 3.1, but the `increase` and `decrease` rate of fluents is an exclusive feature of PDDL+.

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

Listing 2.3: Example of PDDL+ durative action from Fox's paper.

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

2.4.3 Probabilistic

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

2.4.3.1 PPDDL

PPDDL is such a language. It was used during the 4th and 5^th IPC for its probabilistic track (Younes and Littman 2004). It allows for probabilistic effects as demonstrated in listing 2.4. The planner must take into account the probability when chosing an action. The plan must be the most likely to succeed. But even with the best plan, failure can occur. This is why probabilistic planning often gives policies instead of a plan. A policy dictates the best choice in any given state, failure or not. While this allows for much more resilient execution, computation of policies are exponentially harder than classical planning. Indeed the planner needs to take into account every outcome of every action in the plan and react accordingly.

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

Listing 2.4: Example of PPDDL use of probabilistic effects from Younes' paper.

2.4.3.2 RDDL

Another language used by the 7th IPC's uncertainty track is RDDL (Sanner 2010). This language has been choosen because of its ability to express problems that are hard to encode in PDDL or PPDDL. Indeed, RDDL is capable of expressing Partially Observable Markovian Decision Process (POMDP) and Dynamic Bayesian Networks (DBN) in planning domains. This along with complex probability laws allows for easy implementation of most probabilistic planning problems. Its syntax differs greatly from PDDL, and seems closer to scala or C++. An example is provided in listing 2.5 from Sanner (2010). In it, we can see that actions in RDDL doesn't need preconditions or effects. In that case the reward is the closest information to the classical goal and the action is simply a parameter that will influence the probability distribution of the events that conditioned the reward.

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

    requirements = { reward-deterministic };
```

```

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

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

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

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

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

instance inst_dbn {

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

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

Listing 2.5: Example of RDDL syntax by Sanner.

2.4.4 Multi-agent

Planning can also be a collective effort. In some cases, a system must account for other agents trying to either cooperate or compete in achieving similar goals. The problem that arise is coordination. How to make a plan meant to be executed with several agents concurently ? Several multi-agent action languages have been proposed to answer that question.

2.4.4.1 MAPL

Another extension of PDDL 2.1, MAPL was introduced to handle synchronization of actions (Brenner 2003). This is done using modal operators over fluents. In that regard, MAPL is closer to the PDDL+ extension proposed earlier. It introduce durative actions that will later be integrated into the PDDL 3.0 standard. MAPL also introduce a synchronization mechanism using speech as a comunication vector.

This seems very specific as explicit comunication isn't a requirement of collaborative work. Listing 2.6 is an example of the syntax of MAPL domains. PDDL 3.0 seems to share a similar syntax.

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

Listing 2.6: Example of MAPL syntax by Brenner.

2.4.4.2 MA-PDDL

Another aspect of multi-agent planning is the ability to affect tasks and to manage interactions between agents efficiently. For this MA-PDDL seems more adapted than MAPL. It is an extension of PDDL 3.1, that makes easier to plan for a team of heteroneous agents (Kovács 2012). In the example in listing 2.7, we can see how action can be affected to agents. While it makes the representation easier, it is possible to obtain similar effect by passing an agent object as parameter of an action in PDDL 3.1. More complex expressions are possible in MA-PDDL, like referencing the action of other agents in the preconditions of actions or the ability to affect different goals to different agents. Later on, MA-PDDL was extended with probabilistic capabilities inspired by PPDDL (Kovács and Dobrowiecki 2013).

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

Listing 2.7: Example of MA-PDDL syntax by Kovacs.

2.4.5 Hierarchical

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

2.4.5.1 UMCP

One of the first planner to support HTN domains was UCMP by Erol *et al.* (1994). It uses Lisp like most of the early planning systems. Apparently PDDL was in part inspired by UCMP's syntax. Like for PDDL, the domain file describes action (called operators here) and their preconditions and effects (called postconditions). The syntax is demonstrated in listing 2.8. The interesting part of that language is the way decomposition is handled. Each task is expressed as a set of methods. Each method has an expansion expression that specifies how the plan should be constructed. It also has a pseudo precondition with modal operators on the temporality of the validity of the literals.

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

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

(declare-method move(x y z)
                :expansion ((n restack x y z))
                :formula (and (not (veq y table))
                              (not (veq x table))
                              (not (veq z table))
                              (before (clear x) n)
                              (before (clear z) n)
                              (before (on x y) n)))

(declare-method move(x y z)
                :expansion ((n dostack x z))
                :formula (and (veq y table)
                              (before (clear x) n)
                              (before (on x y) n)))
```

Listing 2.8: Example of the syntax used by UCMP.

2.4.5.2 SHOP2

The next HTN planner is SHOP2 by Nau *et al.* (2003). It remains to this day, one of the reference implenentation of an HTN planner. The SHOP2 formalism is quite similar to UCMP's: each method has a signature, a precondition formula and eventually a decomposition description. This decomposition is a set of methods like in UCMP. The methods can be also partially ordered allowing more expressive plans. An example of the syntax of a method is given in listing 2.9.

```
(:method
  ; head
```

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

Listing 2.9: Example of method in the SHOP2 language.

2.4.5.3 HDDL

A more recent example of HTN formalism comes from the PANDA framework by Bercher *et al.* (2014). This framework is considered the current standard of HTN planning and allows for great flexibility in domain description. PANDA takes previous formalism and generalize them into a new language exposed in listing 2.10. That language was called HDDL after its most used file extention.

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

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

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

```

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

Listing 2.10: Example of HDDL syntax as used in the PANDA framework.

2.4.5.4 HPDDL

A very recent language proposition was done by RAMOUL (2018). He proposes HPDDL with a simple syntax similar to the one of UCMP. In listing 2.11 we give an example of HPDDL method. Its expressive power seems similar to that of UCMP and SHOP. Except for a possible commercial integration with PDDL4J (Pellier and Fiorino 2017), there doesn't seem to have any advantages compared to earlier works.

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

Listing 2.11: Example of HPDDL syntax as described by Ramoul.

2.4.6 Ontological

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

2.4.6.1 WebPDDL

This question finds it first answer in 2002 with WebPDDL. This language, explicited in listing 2.12, is meant to be compatible with RDF by using URI identifiers for domains (McDermott and Dou 2002). The syntax is inspired by PDDL, but axioms are added as constraints on the knowledge domain. Actions also have a return value and can have variables that aren't dependant on their parameters. This allows for greater expressivity than regular PDDL, but can be partially emulated using PDDL 3.1 constraints and object fluents.

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

Listing 2.12: Example of WebPDDL syntax by Mc Dermott.

2.4.6.2 OPT

This previous work was updated by McDermott (2005). The new version is called OPT and allows for some further expressivity. It can express hierarchical domains with links between actions and even advanced data structure. The syntax is mostly an update of WebPDDL. In listing 2.13, we can see that the URI were replaced by simpler names, the action notation was simplified to make the parameter and return value more natural. Axioms were replaced by facts with a different notation.

```
(define (domain www-agents)
  (:extends knowing regression-planning commerce)
  (:requirements :existential-preconditions :conditional-effects)
  (:types Message - Obj Message-id - String )
  (:type-fun (Key t) (Feature-type (keytype t)))
  (:type-fun (Key-pair t) (Tup (Key t) t))
  (:functions (price-quote ?m - Money)
              (query-in-stock ?pid - Product-id)
```

```
               (reply-in-stock ?b - Boolean) - Message)
  (:predicates (web-agent ?x - Agent)
               (reply-pending a - Agent id - Message-id msg - Message)
               (message-exchange ?interlocutor - Agent
                                 ?sent ?received - Message
                                 ?eff - Prop)
               (expected-reply a - Agent sent expect-back - Message))
  (:facts
    (freevars (?agt - Agent ?msg-id - Message-id
               ?sent ?reply - Message)
      (<- (and (web-agent ?agt)
               (reply-pending ?agt ?msg-id ?sent)
               (expected-reply ?agt ?sent ?reply))
               (normal-value (receive ?agt ?msg-id) ?reply))))
  (:action (send ?agt - Agent ?sent - Message) - (?sid - Message-id)
    :precondition (web-agent ?agt)
    :effect (reply-pending ?agt ?sid ?sent))
  (:action (receive ?agt - Agent ?sid - Message-id) - (?received -
    Message)
    :vars (?sent - Message ?eff - Prop)
    :precondition (and (web-agent ?agt)
                       (reply-pending ?agt ?sid ?sent))
    :effect (when (message-exchange ?agt ?sent ?received ?eff) ?eff)))
```

Listing 2.13: Example of the updated OPT syntax as described by Mc Dermott.

2.5 Color and general planning representation

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

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

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

Listing 2.14: Content of the file “planning.w“

The file presented in listing 2.14, gives the definition of the syntax of fluents and actions in SELF. The first line includes the default syntax file using the first statement syntax. The fluents are simply typed as entities. This allows them to be either parameterized entities or statements. States are either a set of fluent or a logical statement between states or fluents. When a state is represented as a set, it represent the conjonction of all fluents in the set.

Then at line 5, we define the preconditions, effects and constraints formalism. They are represented as simple properties between actions and states. This allows for simple expression of commonly expressed formalism like the ones found in PDDL. Line 6 expresses the other attributes of actions like cost, duration and prior probability of success.

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

3 Online and Flexible Planning Algorithms

3.1 Existing Algorithms

État de l'art

3.2 Lollipop

3.2.1 Operator Graph

3.2.2 Negative Refinements

3.2.3 Usefullness Heuristic

3.2.4 Algorithm

3.2.5 Theoretical and Empirical Results

3.3 HEART

3.3.1 Domain Compilation

3.3.2 Abstraction in POP

3.3.3 Planning in cycle

3.3.4 Properties of Abstract Planning

3.3.5 Computational Profile

3.4 Planning Improvements

3.4.1 Heuristics using Semantics

3.4.2 Macro-Action learning

3.5 Recognition

3.5.1 Existing approcahes

3.5.2 Rico

3.5.2.1 Probabilities and approximations

We define a probability distribution over dated states of the world. That distribution is in part given and fixed and the rest needs computation. **TODO : that's super bad...**

Here is the list of given prior probabilities and asumptions :

- $P(O) = \prod_{o \in O} P(o)$
- $P(\mathcal{G}) = \sum_{G \in \mathcal{G}} P(G) = 1$ since we assume that the agent must be pursuing one of the goals.
- $P(G|\pi) = 1$ for a plan π appliable in I that achieves G.

From dirrect application of Bayes theorem and the previous assomptions, we have :

$$P(\pi|O) = \frac{P(O|\pi)P(\pi)}{P(O)} = \frac{P(O|\pi)P(\pi|G)P(G)}{P(O)} \tag{3.1}$$

$$P(G|O) = \frac{P(O|G)P(G)}{P(O)} \tag{3.2}$$

From the total probability formula :

$$P(O|G) = \sum_{\pi \in \Pi_G} P(O|\pi)P(\pi|G) \tag{3.3}$$

$$P(O|G) = \sum_{\pi \in \Pi_G} P(O|\pi)P(\pi|G) \tag{3.4}$$

4 Conclusion

Apendix

References

Alessandro, W., and I. Piumarta
OMeta: An object-oriented language for pattern matching, *Proceedings of the 2007 symposium on Dynamic languages*, 2007.

Aljazzar, H., and S. Leue
K: A heuristic search algorithm for finding the k shortest paths, *Artificial Intelligence*, 175 (18), 2129–2154, 2011.

Asimov, I.
The gods themselves, Greenwich, Connecticut: Fawcett Crest, 1973.

Baader, F.
The description logic handbook: Theory, implementation and applications, Cambridge university press, 2003.

Backus, J. W.
The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference, *Proceedings of the International Comference on Information Processing, 1959*, 1959.

Becket, R., and Z. Somogyi
DCGs+ memoing= packrat parsing but is it worth it?, *International Symposium on Practical Aspects of Declarative Languages*, Springer, 2008, 182–196.

Beckett, D., and T. Berners-Lee
Turtle - Terse RDF Triple Language, W3C Team Submission W3C, March 2011.

Bercher, P., S. Keen, and S. Biundo
Hybrid planning heuristics based on task decomposition graphs, *Seventh Annual Symposium on Combinatorial Search*, 2014.

Brenner, M.
A multiagent planning language, *Proc. Of the Workshop on PDDL, ICAPS*, 2003, 3,33–38.

Bürckert, H.-J.
Terminologies and rules, *Workshop on Information Systems and Artificial Intelligence*, Springer, 1994, 44–63.

Cantor, G.
On a Property of the Class of all Real Algebraic Numbers., *Crelle's Journal for Mathematics*, 77, 258–262, 1874.

Cantor, G.
Beiträge zur Begründung der transfiniten Mengenlehre, *Mathematische Annalen*, 46 (4), 481–512, 1895.

Chomsky, N.
Three models for the description of language, *IRE Transactions on information theory*, 2 (3), 113–124,

1956.

Ciesielski, K.
Set Theory for the Working Mathematician, Cambridge University Press, August 1997.

D'Alfonso, D.
Generalized Quantifiers: Logic and Language, 2011.

Erol, K., J. A. Hendler, and D. S. Nau
UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning, *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, University of Chicago, Chicago, Illinois, USA: AAAI Press, June 1994, 2,249–254.

Fikes, R. E., and N. J. Nilsson
STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial intelligence*, 2 (3-4), 189–208, 1971.

Ford, B.
Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl, *ACM SIGPLAN Notices*, ACM, 2002, 37,36–47.

Ford, B.
Parsing expression grammars: A recognition-based syntactic foundation, *ACM SIGPLAN Notices*, ACM, 2004, 39,111–122.

Fox, M., and D. Long
PDDL+: Modeling continuous time dependent effects, *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002, 4,34.

Fraenkel, A. A., Y. Bar-Hillel, and A. Levy
Foundations of set theory, vol. 67Elsevier, 1973.

Ghallab, M., C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, et al.
PDDL – The planning domain definition language, 1998.

Ghallab, M., D. Nau, and P. Traverso
Automated planning: Theory & practice, Elsevier, 2004.

Ghallab, M., D. Nau, and P. Traverso
Automated Planning and Acting, Cambridge University Press, 2016.

Gradel, E., M. Otto, and E. Rosen
Two-variable logic with counting is decidable, *Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on*, IEEE, 1997, 306–317.

Grünwald, P.
A minimum description length approach to grammar inference, in S. Wermter, E. Riloff, and G. Scheler (eds.), *Connectionist, Statistical and Symbolic Approaches to Learning for Natural Language Processing*, Lecture Notes in Computer Science; Springer Berlin Heidelberg, 1996, 203–216.

Hart, D., and B. Goertzel
Opencog: A software framework for integrative artificial general intelligence, *AGI*, 2008, 468–472.

Hehner, E. C.
A practical theory of programming, Springer Science & Business Media, 2012.

Henglein, F., and U. T. Rasmussen
PEG parsing in less space using progressive tabling and dynamic analysis, *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ACM, 2017, 35–46.

Hernández, D., A. Hogan, and M. Krötzsch
Reifying RDF: What works well with wikidata?, *SSWS@ ISWC*, 1457, 32–47, 2015.

Horrocks, I., P. F. Patel-Schneider, and F. V. Harmelen
From SHIQ and RDF to OWL: The Making of a Web Ontology Language, *Journal of Web Semantics*, 1, 2003, 2003.

Höschele, M., and A. Zeller
Mining input grammars with AUTOGRAM, *Proceedings of the 39th International Conference on Software Engineering Companion*, IEEE Press, 2017, 31–34.

Hutton, G., and E. Meijer
Monadic parser combinators, 1996.

Klyne, G., and J. J. Carroll
Resource Description Framework (RDF): Concepts and Abstract Syntax, Language Specification. Ed. B. McBride W3C, 2004.

Kovacs, D. L.
BNF Description of PDDL 3.1, IPC, 2011.

Kovács, D. L.
A multi-agent extension of PDDL3. 1, 2012.

Kovács, D. L., and T. P. Dobrowiecki
Converting MA-PDDL to extensive-form games, *Acta Polytechnica Hungarica*, 10 (8), 27–47, 2013.

Kovitz, B.
Terminology - What do you call graphs that allow edges to edges?, *Mathematics Stack Exchange*, January 2018.

Lindström, P.
First Order Predicate Logic with Generalized Quantifiers, 1966.

Loff, B., N. Moreira, and R. Reis
The Computational Power of Parsing Expression Grammars, *International Conference on Developments in Language Theory*, Springer, 2018, 491–502.

McDermott, D.
OPT Manual Version 1.7. 3 (Reflects Opt Version 1.6. 11) DRAFT*, 2005.

McDermott, D., and D. Dou
Representing disjunction and quantifiers in RDF, *International Semantic Web Conference*, Springer, 2002, 250–263.

Motik, B.
On the properties of metamodeling in OWL, *Journal of Logic and Computation*, 17 (4), 617–637, 2007.

Nau, D. S., T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, et al.
SHOP2: An HTN planning system, *J. Artif. Intell. Res.(JAIR)*, 20, 379–404, 2003.

Paulson, L.
A semantics-directed compiler generator, *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82*, Albuquerque, Mexico: ACM Press, 1982, 224–233. doi:10.1145/582153.582178.

Peano, G.
Arithmetices principia: Nova methodo exposita, Fratres Bocca, 1889.

Pednault, E. P.
ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus., *Kr*, 89, 324–332, 1989.

Pellier, D., and H. Fiorino
PDDL4J: A planning domain description library for java, December 2017. doi:10.1080/0952813X.2017.1409278.

Penberthy, J. S., D. S. Weld, and others
UCPOP: A Sound, Complete, Partial Order Planner for ADL, *Kr*, 92, 103–114, 1992.

RAMOUL, A.
Mixed-initiative planning system to assist the management of complex IT systems PhD thesis, 2018.

Renggli, L., S. Ducasse, T. Gîrba, and O. Nierstrasz
Practical Dynamic Grammars for Dynamic Languages, *Workshop on Dynamic Languages and Applications*, Malaga, Spain, 2010, 4.

Sanner, S.
Relational dynamic influence diagram language (rddl): Language description Unpublished Ms. Australian National University 2010.

Silberschatz, A.
Port directed communication, *The Computer Journal*, 24 (1), 78–82, January 1981. doi:10.1093/comjnl/24.1.78.

Souto, D. C., M. V. Ferro, and M. A. Pardo
Dynamic Programming as Frame for Efficient Parsing, *Proceedings SCCC'98. 18th International Conference of the Chilean Society of Computer Science (Cat. No.98EX212)(SCCC)*, November 1998, 68. doi:10.1109/SCCC.1998.730784.

Tolksdorf, R., L. Nixon, F. Liebsch, Minh NguyenD., and Paslaru BontasE.
Semantic web spaces, 2004.

Unicode Consortium
Unicode Character Database, *About the Unicode Character Database*, https://www.unicode.org/ucd/#Latest, June 2018a.

Unicode Consortium
The Unicode Standard, Version 11.0, Core Specification 11.0Mountain View, CA, June 2018b.

Van Harmelen, F., V. Lifschitz, and B. Porter
Handbook of knowledge representation, vol. 1Elsevier, 2008.

Vepstas, L.
Hypergraph edge-to-edge, *Wikipedia*, May 2008.

Vepštas, L.
Sheaves: A Topological Approach to Big Data, 2008.

W3C
RDF Semantics, W3C, 2004a.

W3C
RDF Vocabulary Description Language 1.0: RDF Schema http://www.w3.org/TR/rdf-schema/, February 2004b.

Younes, H. akan L., and M. L. Littman
PPDDL 1.0: An extension to PDDL for expressing planning domains with probabilistic effects, *Techn. Rep. CMU-CS-04-162*, 2004.