Endomorphic metalanguage and abstract planning for real-time intent recognition

Antoine Gréa

Table of Content

List of Figures

Acronyms

ADL Action Description Language 97

AGI Artificial General Intelligence 55

AI Artificial Intelligence 14, 20, 53, 54, 55

AIPS Artificial Intelligence Planning Systems 97

ASCII American Standard Code for Information Interchange 63

BFS Breadth-First Search 22

BNF Backus-Naur Form 49, 50, 61, 102

CBP Case-Based Planning 94

CFG Context-Free Grammar 50, 51, 123

DL Description Logic 49, 52, 53, 54, 59

DSL Domain Specific Language 53

FD Fast Downward 125

FF Fast Forward 116

FOL First Order Logic 34, 36, 43, 45, 47, 49

HEART HiErarchical Abstraction for Real-Time partial order planner 141, 142, 144, 145

HOL Higher Order Logic 36, 45, 47

HTN Hierarchical Task Network 95, 100, 104, 105, 106, 109, 110, 116, 122, 123, 124, 139, 144

HTN-TI Hierarchical Task Network with Task Insersion 95

IPC International Planning Competitions 97, 99, 101, 102, 106

LOLLIPOP aLternative Optimization with partiaL pLan Injection Partial Ordered Planner 124, 127, 132, 133, 135, 136, 137, 138

OWL Ontology Web Language 49, 54, 56

PDDL Planning Domain Description Language 22, 97, 98, 99, 100, 101, 102, 103, 104, 108, 109, 111, 112, 113

PEG Parsing Expression Grammar 52

POCL Partial Order Causal Links 94, 116, 118, 119, 124, 125, 127, 128, 130, 133, 136, 137, 139, 140, 143

PSP Plan Space Planning 94, 95, 116, 117, 120, 122, 124, 125

RDF Resource Description Framework 49, 54, 55, 56, 57, 58, 60, 68, 77, 78, 108

SELF Structurally Expressive Language Framework 22, 57, 58, 59, 60, 63, 67, 70, 71, 72, 73, 75, 77, 78, 87, 96, 110, 111, 112, 113

STN Simple Temporal Network 110

STRIPS Stanford Research Institute Problem Solver 97, 123

UCD Unicode Character Database 62, 63, 64

UCPOP soUnd Complete Partial Order Planner 97

URI Uniform Resource Identier 54, 56, 60, 108

URL Uniform Resource Locator 72

UTF-8 Unicode Transformation Format on 8-bits 62, 63

W3C World Wide Web Consortium 49, 54

XML eXtensible Markup Language 54, 60

ZFC Zermelo–Fraenkel set theory with the axiom of Choice 37, 40, 44, 45, 58

Symbols

$\chi_{\mathbb{S}}$ Search iterator 89

$[]$ Iverson's brakets 140

$\{\!|\}$ Mapping 91, 135

μ Meta 59, 65, 73

ν Name 73

$\gg$ Null relation 30, 31, 34, 85

k Number of solutions 90, 92

t^* Optimization time 91, 94

$\otimes^{\dashrightarrow}$ Orphan 129

ρ Parameter 73

$\mathbb{p}$ Problem 89

$\mathbb{\pi}$ Plan 135

$\mathbb{\Pi}$ Set of plans 85, 86, 87, 95

pre Precondition 85

$\mathbb{P}$ Probability 85, 91

$\mu^\bullet$ Reification 59, 62, 70, 73

$\odot$ Resolver 136

ω Root operator 87, 135

$\gamma_{\mathbb{S}}$ Solution constraints 89, 90, 91, 92, 93

$\mathbb{S}$ Set of searches 89, 92

$t_{\mathbb{S}}$ Search time 91, 94

$\S$ Solution quantifier 36

S Statements 57, 60

$\square$ State 82

$\square$ States 87, 93

$\triangledown$ Functionnal subposition 32, 34

$\subseteq$ Subsumption 73

$:$ Specification 37, 59, 73

$\triangle$ Functionnal superposition 32, 34

Preface

Abstract

Artificial Intelligence

Human-machine interaction is among the most complex problems in the field of AI. Indeed, software that cooperates with dependent people must have incompatible qualities such as speed and expressiveness, or even precision and generality. The objective is then to design models and mechanisms capable of making a compromise between efficiency and generality. These models make it possible to expand the possibilities of adaptation in a fluid and continuous way. Thus, the search for a complete and optimal response has overshadowed the usefulness of these models. Indeed, explainability and interactivity are at the heart of popular concerns of modern AI systems. The main issue with such requirements are that semantic information is hard to convey to a program. Part of the solution to this problem lies in how to represent knowledge.

Formalization is the best way to rigorously define the problem. Mathematics is the best set of tools to express formal notions. However, since our approach requires non-classical mathematics, it is easier to define a coherent theory that simply fits our needs. That theory is a weak instance of category theory. We propose a functional algebra inspired by lambda calculus. It is then possible to reconstruct classical mathematical concepts as well as other tools and structures useful for our usage.

By using this formalism, it becomes possible to axiomatize an endomorphic metalanguage. This one manipulates a dynamic grammar capable of adjusting its semantics to exploitation. The recognition of basic structures allows this language to avoid using keywords. This, combined with a new model of knowledge representation, supports the construction of an expressive knowledge description model.

With this language and this formalism, it becomes possible to create frameworks in fields that were previously heterogeneous. For example, in automatic planning, the classic state-based model makes it impossible to unify the representation of planning domains. This results in a general planning framework that allows all types of planning domains to be expressed.

Concrete algorithms are then created that show the principle of intermediate solutions. Two new approaches to real-time planning are developed and evaluated. The first is based on a usefulness heuristic of planning operators to repair existing plans. The second uses hierarchical task networks to generate valid plans that are abstract and intermediate solutions. These

plans allow for a shorter execution time for any use that does not require a detailed plan.

We then illustrate the use of these plans on intent recognition by reverse planning. Indeed, in this field, the fact that no plan libraries are required makes it easier to design recognition models. By exploiting abstract plans, it becomes possible to create a system theoretically more efficient than those using complete planning.

Format

In this section we present a quick guide to the presentation of the information in this document. This will give its global structure and each of the type of formatting and its meaning.

Text format

The text can be emphasized **more** or *less* to make a key word more noticeable.

Citations

In text citations will be in this format: Author *et al.* (year) to make the author part of the text and (Author *et al.* year) when simply referencing the work.

Quotes

Sometimes, important quotes needs emphasis. They are presented as:

> *"Don't quote me on that !"* Gréa (2019)

Acknowledgements

Ehlo, this is the author of this thesis. I would like to thank a few people without whom none of this would have been possible.

It was a beautiful morning of June that I first walked the Doua wooded driveways to meet my supervisor team: Samir Aknine and Laetitia Matignon. They have given me their trust in the success of this thesis project. Without their advice and support none of this would have been possible. I would like to thank Samir in particular whom, as a mentor and friend, has listened to me

Actual picture of the time

and supported me even in the most difficult moments with compassion and understanding.

I would also like to thank Christine Gertosio, who offered me unfailing support in my teaching department and granted me the opportunity to pursue an ATER offer at POLYTECH Lyon. It was a real pleasure to teach classes of passionate and enthusiastic students.

I would also like to offer my gratitude to the researchers who helped me in this adventure, in no particular order: Fabrice Jumel, Jacques Saraydaryan, Shirin Sohrabi, Damien Pelier. I would also like to thank Linas Vepstas, who greatly helped me in the mathematical part of this thesis.

Without funding, this work wouldn't have been possible. I would like to thank the arc 2 (Academic Research Community) research allocation courtesy of the Auvergne-Rhône-Alpes region in France. This organism is acting toward an improvement of the quality of life and of aging.

This work was done with the **Lyon 1** university in the LIRIS (Laboratory of computeR science in Image and Information Systems). I was part of vibrant and wonderful **sma** team (Multi-Agent System). Part of the work was funded by teaching at the university Lumière Lyon 2 and the POLYTECH engineering school.

Finally, I would like to thank my family for supporting me despite my periods of avolition. I would like to thank in particular my mother Charlotte and my sister Sarah for having done their best to keep me company and pretend to listen to me talk about the technicalities of my thesis in order to make me progress.

Thank you again.

Sarah Gréa helping me do the rubber duck method.

Préface

Résumé

L'interaction personne-machine fait partie des problèmes les plus complexes dans le domaine de l'intelligence artificielle (IA). En effet, les logiciels qui coopèrent avec des personnes dépendantes doivent avoir des qualités incompatibles telles que la rapidité et l'expressivité, voire la précision et la généralité. L'objectif est alors de concevoir des modèles et des mécanismes capables de faire un compromis entre efficacité et généralité. Ces modèles permettent d'élargir les possibilités d'adaptation de manière fluide et continue. Ainsi, la recherche d'une réponse complète et optimale a éclipsé l'utilité de ces modèles. En effet, l'explicabilité et l'interactivité sont au cœur des préoccupations populaires des systèmes modernes d'IA. Le principal problème avec de telles exigences est que l'information sémantique est difficile à transmettre à un programme. Une partie de la solution à ce problème réside dans la manière de représenter les connaissances.

La formalisation est le meilleur moyen de définir rigoureusement un problème. Aussi, les mathématiques sont le meilleur ensemble d'outils pour exprimer des notions formelles. Cependant, comme notre approche exige des mathématiques non classiques, il est plus facile de définir une théorie cohérente qui correspond simplement à nos besoins. Cette théorie est une instance partielle de la théorie des catégories. On propose une algèbre fonctionnelle inspirée du lambda calcul. Il est alors possible de reconstruire des concepts mathématiques classiques ainsi que d'autres outils et structures utiles à notre usage.

En se servant de ce formalisme, il devient possible d'axiomatiser un métalangage endomorphique. Celui-ci manipule une grammaire dynamique capable d'ajuster sa sémantique à l'usage. La reconnaissance des structures de base permet à ce langage de ne pas utiliser de mot-clés. Ceci, combiné à un nouveau modèle de représentation des connaissances, supporte la construction d'un modèle de représentation des connaissances expressive.

Avec ce langage et ce formalisme, il devient envisageable de créer des cadriciels dans des champs jusqu'alors hétéroclites. Par exemple, en planification automatique, le modèle classique à état rend l'unification de la représentation des domaines de planification impossible. Il en résulte un cadriciel général de la planification permettant d'exprimer tout type de domaines en vigueur.

On crée alors des algorithmes concrets qui montrent le principe des solutions intermédiaires. Deux nouvelles approches à la planification en temps réel

sont présentées et évaluées. La première se base sur une euristique d'utilité des opérateurs de planification afin de réparer des plans existants. La seconde utilise la planification hiérarchique pour générer des plans valides qui sont des solutions abstraites et intermédiaires. Ces plans rendent possible un temps d'exécution plus court pour tout usage ne nécessitant pas le plan détaillé.

On illustre alors l'usage de ces plans sur la reconnaissance d'intention par planification inversée. En effet, dans ce domaine, le fait de ne pas nécessiter de bibliothèques de plans rend la création de modèles de reconnaissance plus aisée. En exploitant les plans abstraits, il devient possible de réaliser un système théoriquement plus performant que ceux utilisant de la planification complète.

Formatage

Dans cette section, nous présentons un guide rapide de la présentation de l'information contenue dans ce document. Cela donnera sa structure globale et chacun du type de formatage et de sa signification.

Format texte

Le texte peut être souligné **plus** ou *moins* pour rendre un mot-clé plus visible.

Références

Les citations textuelles seront dans ce format : Auteur *et al.* (année)) pour que l'auteur fasse partie du texte et (Auteur *et al.*année) lorsqu'on se contente de faire référence à l'œuvre.

Citations

Parfois, les citations importantes doivent être mises en avant. Ils sont présentés comme :

Gréa (2019) *"Ne me citez pas là-dessus !"*

Remerciements

Je voudrais remercier quelques personnes sans qui rien de cela n'aurait été possible.

C'était une belle matinée de juin que j'ai parcouru pour la première fois les allées boisées de la Doua afin de rencontrer mon équipe encadrante: Samir Aknine et Laetitia Matignon. Ces derniers m'ont accordé leur confiance dans la réussite de ce projet de thèse. Sans leur conseil et soutien rien de cela n'aurait été possible. Je remercie particulièrement Samir, qui en tant qu'encadrant, et ami, a su m'écouter et me soutenir même dans les moments les plus pénibles avec compassion et compréhension.

Photo de l'époque

Je souhaite également remercier Christine Gertosio, qui m'a offert une aide sans faille dans mon service d'enseignement et qui m'a donné l'opportunité d'une offre d'ATER à POLYTECH Lyon. C'était un véritable plaisir de donner des cours à des promotions d'élèves passionnés et enthousiastes.

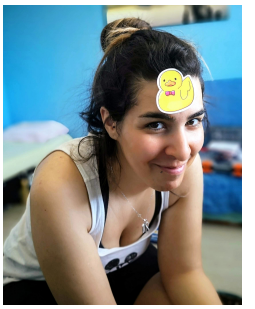
Je souhaite également offrir ma gratitude aux chercheurs qui m'ont aidé dans cette aventure, sans ordre particulier: Fabrice Jumel, Jacques Saraydaryan, Shirin Sohrabi, Damien Pelier. J'aimerais également remercier Linas Vepstas, qui m'a grandement aidé dans la partie mathématique de cette thèse.

Sans financement, ce travail n'aurait pas été possible. Je tiens à remercier l'allocation de recherche du arc 2 (Academic Research Community) gracieuseté de la région de Auvergne-Rhône-Alpes en France. Cet organisme agit pour l'amélioration de la qualité de vie et du vieillissement.

Ce travail a été réalisé avec l'université Lyon 1 dans le LIRIS (Laboratoire de calcul scientifique en Image et Systèmes d'Information). Je faisais partie d'une équipe dynamique et conviviale sma (Multi-Agent System). Une partie du travail a été financée par l'enseignement à l'université Lumière Lyon 2 et à l'école d'ingénieurs Polytech.

Enfin, je tiens à remercier ma famille de m'avoir supporté malgré mes périodes d'avolition. Je voudrais remercier tout particulièrement ma mère Charlotte et ma sœur Sarah pour avoir fait de leur mieux pour me tenir compagnie et faire semblant de m'écouter déblatérer des technicités de ma thèse afin de me faire avancer.

Encore merci.



Sarah Gréa m'aidant en utilisant la méthode du canard en plastique.

1 Introduction

In antiquity, philosophy, mathematics and logic were considered as a single discipline. Since Aristotle we have realized that the world is not just black and white but full of nuances and colors. The inspiration for this thesis comes from one of the most influential philosophers and scientists of his time: Alfred Korzibsky. He founded a discipline he called *general semantics* to deal with problems of knowledge representation in humans. Korzibsky then found that complete knowledge of reality being inaccessible, we had to abstract. This abstraction is then only similar to reality in its structure. In these pioneering works, we find notions similar to that of modern descriptive languages.

It is from this inspiration that this document is built. We start off the beaten track and away from computer science by a brief excursion into the world of mathematical and logical formalism. This makes it possible to formalize a language that allows to describe itself partially by structure and that evolves with its use. The rest of the work illustrates the possible applications through specific fields such as automatic planning and intention recognition.

1.1 Motivations

The social skills of modern robots are rather poor. Often, that lack inhibits human-robot communication and cooperation. Humans being a social species, they require the use of implicit social cues in order to interact comfortably with an interlocutor.

In order to enhance assistance to dependent people, we need to account for any deficiency they might have. The main issue is that the patient is often unable or unwilling to express their needs. That is a problem even with human caregivers as the information about the patient's intents needs to be inferred from their past actions.

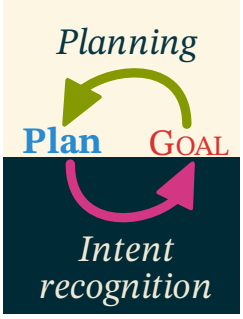
This aspect of social communications often eludes the understanding of AI systems. This is the reason why intent recognition is such a complicated problem. The primary goal of this thesis is to address this issue and create the formal foundations of a system able to help dependent people.

1.2 Problem

First, *what exactly is intent recognition ?* The problem is simple to express: finding out what other agents want to do before they do. It is important to distinguish between several notions. *Plans* are the sequence of actions that the agent is doing to achieve a *goal*. This goal is a concrete explanation of the wanted result. However, the *intent* is more of a set of abstract goals, some of which may be vague or impossible (e.g. drink something, survive forever, etc.).

Some approaches use trivial machine learning methods, along with a hand-made plan library to match observations to their most likely plan using statistics. The issue with these common approaches is that they require an extensive amount of training data and need to be trained on each agent. This makes the practicality of such system quite limited. To address this issue, some works proposed hybrid approaches using logical constraints on probabilistic methods. These constraints are made to guide the resolution toward a more coherent solution. However, all probabilistic methods require an existing plan library that can be quite expensive to create. Also, plan libraries cannot take into account unforeseen or unlikely plans.

A work from Ramırez and Geffner (2009) added an interesting method to solve this issue. Indeed, they noticed an interesting parallel between that problem and the field of automated planning. This analogy was made by using the Theory of mind (Baker *et al.* 2011), which states that any agent will infer the intent of other agents using a projection of their own expectations on the observed behaviors of the other agents.



This made the use of planning techniques possible to infer intents without the need for extensive and well-crafted plan libraries. Now only the domain of the possible actions, their effects and prerequisites are needed to infer the logical intent of an agent.

The main issue of planning for that particular use is computation time and search space size. This prevents most planners to make any decision before the intent is already realized and therefore being useless for assistance. This time constraint leads to the search of a real-time planner algorithm that is also expressive and flexible.

1.3 Contributions

In order to achieve such a planner, the first step was to formalize what is exactly needed to express a domain. Hierarchical and partially ordered plans gave the most expressivity and flexibility but at the cost of time and performance. This is why, a new formalism of knowledge representation was needed in order to increase the speed of the search space exploration while restricting it using semantic inference rules.

While searching for a knowledge representation model, we developed some prototypes using standard ontology tools but all proved to be too slow and inexpressive for that application. This made the design of a lighter but more flexible knowledge representation model, a requirement for planning domain representation.

Then the planning formalism has to be encoded using our general knowledge representation tool. Since automated planning has a very diverse ecosystem of approaches and paradigms, its standard, the PDDL needs use of various extensions. However, no general formalism has been given for PDDL and some approaches often lack proper extensions (hierarchical planning, plan representation, etc.). This is why a new formalism is proposed and compared to the one used as the standard of the planning community.

Planning Domain Description Language

Then finally, a couple of planners were designed to attempt answering the speed and flexibility requirements of human intent recognition. The first one is a prototype that aims to evaluate the advantages of repairing plans to use several heuristics. The second is a more complete prototype derived from the first (without plan repairs), which also implements a BFS approach to hierarchical decomposition of composite actions. This allows the planner to provide intermediary plans that, while incomplete, are an abstraction of the result plans. This allows for anytime intent recognition probability computation using existing techniques of inverted planning.

Breadth-First Search

1.4 Plan

In this document we will describe a few contributions from the new SELF for intent recognition. Each chapter builds on the previous one. Usually a chapter will be the application of the one before, all going toward intent recognition using inverted planning.

Structurally Expressive Language Framework

Chapter 2

First we will present a new mathematical model that suits our needs. This axiomatic theory is used to create a model capable of describing all the mathematical notions required for our work.

Chapter 3

In the third chapter, a new knowledge description system is presented as well as the associated grammar and inference framework. This system is based on triple representation to allow for structurally defined semantic.

Chapter 4

The chapter 4 is an introduction to automated planning along with a formal description of a general planner using appropriate search spaces and solution constraints.

Chapter 5

The fifth chapter is an application of knowledge description to automated planning. This allows us to design a general planning framework that can express any existing type of domain. Existing languages are compared to our proposed approach.

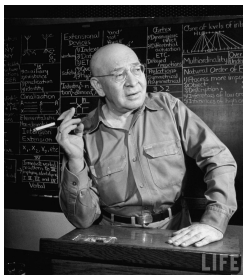
Chapter 6

Using this framework, two online planning algorithms are presented in chapter six: one that uses repairs on existing plans and one that uses hierarchical domains to create intermediary abstract plans.

Chapter 7

The final chapter is about intent recognition and its link to planning. Existing works are presented as well as a technique called *inverted planning*.

2 Foundation and Tools



Alfred Korzybski (1933, ch. 4 pp. 58)

> *"A map* is not *the territory it represents, but, if correct, it has a* similar structure *to the territory, which accounts for its usefulness."*

Mathematics and logic are at the heart of all formal sciences, including computer science. The boundary between mathematics and computer science is quite blurry. Indeed, computer science is applied mathematics and mathematics are abstract computer science. Both cannot be separated when needing a formal description of a new model.

In mathematics, a *foundation* is an *axiomatic theory* that is consistent and well-defined. It can also be called a **model**. For a foundation to be generative of a subset of mathematics, it must define all the supported notions.

In this chapter, we define a new formalism as well as a proposed foundation that lies on the bases of the type theory and lambda calculus. With this formalism, we define the classical set theory (which is the foundation for classical mathematics). The contribution is mainly in the axiomatic system, and functional algebra. The rest is simply an explanation, using our formalism, of existing mathematical notions and structures commonly used in computer science. **This formalism is used for all the formulas later on this document.**

2.1 Existing Model Properties

Any knowledge must be expressed using an encoding support (medium) like a language. Natural languages are quite expressive and allow for complex abstract ideas to be communicated between individuals. However, in science we encounter the first issues with such a language. It is culturally biased and improperly conveys formal notions and proof constructs. Indeed, natural languages are not meant to be used for rigorous mathematical proofs. This is one of the main conclusions of the works of Korzybski (1958) on "general semantics". The original goal of Korzybski was to pinpoint the errors that led humans to fight each other in World War I. He affirmed that the language is unadapted to convey information reliably about objective facts or scientific notions. There is a discrepancy between the natural language and the underlying structure of the reality.

In the following sections we describe a few inherent properties of formalism and mathematical reasoning that are useful to consider when defining a theory.

2.1.1 Abstraction

> **abstraction** (n.d.): *The process of formulating generalized ideas or concepts by extracting common qualities from specific examples*

Collins English Dictionary (2014)

The idea behind abstraction is to simplify the representation of complex instances. This mechanism is at the base of any knowledge representation system. Indeed, it is unnecessarily expensive to try to represent all properties of an object. An efficient way to reduce that knowledge representation is to prune away all irrelevant properties while also only keeping the ones that will be used in the context. *This means that abstraction is a losy process* since information is lost when abstracting from an object.

Since this is done using a language as a medium, this language is a *host language*. Abstraction will refer to an instance using a *term* (also called symbol) of the host language. If the host language is expressive enough, it is possible to do abstraction on an object that is already abstract. The number of layers abstraction needed for a term is called its *abstraction level*. Very general notions have a higher abstraction level and we represent reality using the null abstraction level. In practice abstraction uses terms of the host language to bind to a referenced instance in a lower abstraction level. This forms a structure that is strongly hierarchical with higher abstraction level terms on top.

Example 1. We can describe an individual organism with a name that is associated to this specific individual. If we name a dog "Rex" we abstract a lot of information about a complex, dynamic living being. We can also abstract from a set of qualities of the specimen to build higher abstraction. For example, its species would be *Canis lupus familiaris* from the *Canidae* family. Sometimes several terms can be used at the same abstraction level like the commonly used denomination "dog" in this case.

Terms are only a part of that structure. It is possible to combine several terms into a *formula* (also called proposition, expression or statements).

2.1.2 Formalization

> **formal** (adj.): *Relating to or involving outward form or structure, often in contrast to content or meaning.*

American Heritage Dictionary (2011a)

A formalization is the act to make formal. The word "formal" comes from Latin *fōrmālis*, from *fōrma*, meaning form, shape or structure. This is the same base as for the word "formula". In mathematics and *formal sciences*

the act of formalization is to reduce knowledge down to formulas. Like stated previously, a formula combines several terms. But a formula must follow rules at different levels:

- *Lexical* by using terms belonging in the host language.
- *Syntactic* as it must follow the grammar of the host language.
- *Semantic* as it must be internally consistent and meaningful.

The information conveyed from a formula can be reduced to one element: its semantic structure. Like its etymology suggests, a formula is simply a structured statement about terms. This structure holds its meaning. Along with using abstraction, it becomes possible to abstract a formula and therefore, to make a formula about other formulas should the host language allowing it.

Example 2. The formula using English "dog is man's best friend" combines terms to hold a structure between words. It is lexically correct since it uses English words and grammatically correct since it can be grammatically decomposed as (n. v. n. p. adj. n.). In that the (n.) stands for nouns (v.) for verbs (adj.) for adjectives and (p.) for possessives. Since the verb "is" is the third person singular present indicative of "be" and the adjective is the superlative of "good", this form is correct in the English language. From there the semantic aspect is correct too but that is too subjective and extensive to formalize here. We can also build a formula about a formula like "this is a common phrase" using the referential pronoun "this" to refer to the previous formula.

Any language is comprised of formulas. Each formula holds knowledge about their subject and state facts or belief. A formula can describe other formulas and even *define* them. However, there is a strong limitation of a formalization. Indeed, a complete formalization cannot occur about the host language. It is possible to express formulas about the host language but *it is impossible to completely describe the host language using itself* (Klein 1975). This comes from two principal reasons. As abstraction is a loose process one cannot completely describe a language while abstracting its definition. If the language is complex enough, its description requires an even more complex *metalanguage* to describe it. And even for simpler language, the issue stands still while making it harder to express knowledge about the language itself. For this we need knowledge of the language *a priori* and this is contradictory for a definition and therefore impossible to achieve.

When abstracting a term, it may be useful to add information about the term to define it properly. That is why most formal system requires a *definition* of each term using a formula. This definition is the main piece of semantic information on a term and is used when needing to evaluate a term in a different abstraction level. However, this is causing yet another problem.

2.1.3 Circularity

> **circularity** (n.d.): *Defining one word in terms of another that is itself defined in terms of the first word.*

American Heritage Dictionary (2011b)

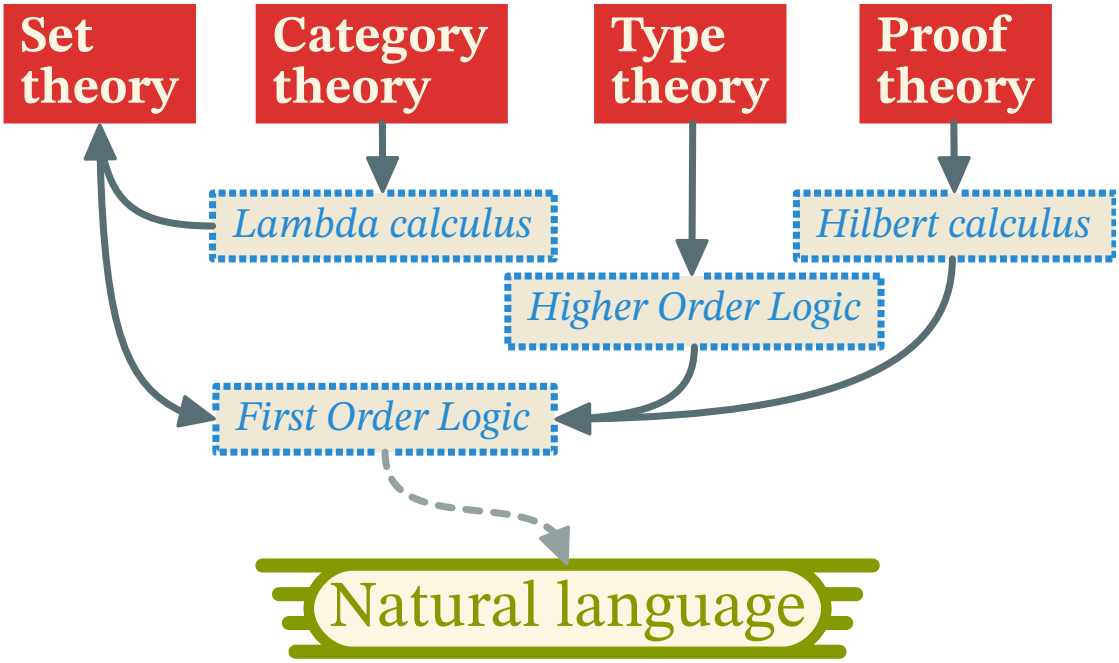


Figure 2.1: Dependency graph of the most common foundational mathematical theories and their underlying implicit formalism.

Circularity is one of the issues we explore in this section about the limits of formalization languages. Indeed, defining a term requires using a formula in the host language to express the abstracted properties of the generalization (Korzybski 1933). The problem is that most terms will have *circular* definitions.

Example 3. Using definitions from the American Heritage Dictionary (2011b), we can find that the term "word" is defined using the word "meaning" that is in turn defined using the term "word". Such circularity can happen to use an arbitrarily long chain of definition that will form a cycle in the dependencies.

For example, we illustrate dependencies between some existing theories and their formalism in figure 2.1. Since a formalization cannot fully be self defined, another host language is generally used, sometimes without being acknowledged.

The only practical way to make some of this circularity disappear is to base on a natural language as host language for defining the most basic terms. This allows to acknowledge the problem in an instinctive way while being aware of it while building the theory.

2.2 Functional theory

We aim to reduce the set of axioms allowing to describe a model. The following theory is a proposition for a possible model that takes into account the previously described constraints. It is inspired by category theory (Awodey 2010), and typed lambda calculus (Barendregt 1984).

2.2.1 Category theory

This theory is based, as its name implies, on *categories*. A category is a mathematical structure that consists of two components:

- A set of **objects** that can be any arbitrary mathematical entities.
- A set of **morphisms** that are functional monomes. They are often represented as *arrows*.

Many definitions of categories exist (Barr and Wells 1990, vol. 49) but they are all in essence similar to this explanation. The best way to see the category theory is as a general theory of functions. Even if we can use any mathematical entity for the types of the components, the structure heavily implies a functional connotation.

2.2.2 Axioms

In this part, we propose a model based on functions. The unique advantage of it lays in its explicit structure that allows it to be fully defined. It also holds a coherent algebra that is well suited for our usage. This approach can be described as a special case of category theory. However, it differs in its priorities and formulation. For example, since our goal is to build a standalone foundation, it is impossible to fully specify the domain or co-domain of the functions and they are therefore weakly specified (Godel and Brown 1940).

Our theory is axiomatic, meaning it is based on fundamental logical proposition called axioms. Those form the base of the logical system and therefore are accepted without any need for proof. In a nutshell, axioms are true prior hypotheses.

The following axioms are the explicit base of the formalism. It is mandatory to properly state those axioms as all the theory is built on top of it.

$== x \mapsto x$

Axiom (Identity)**.** Let's the identity function be $=$ that associates every function to itself. This function is therefore transparent as by definition $=(x)$ is the same as x. It can be described by using it as **affectation** or aliases to make some expressions shorter or to define any function.

In the rest of the document, classical equality and the identity function will refer to the same notion.

That axiom implies that the formalism is based on *functions*. Any function can be used in any way as long as it has a single argument and returns only one value at a time (named image, which is also a function).

It is important to know that **everything** in our formalism *is* a function. Even notions such as literals, variables or set from classical mathematics are functions. This property is inspired by lambda calculus and some derived functional programming languages.

Axiom (Association)**.** Let's the term $7\rightarrow$ be the function that associates two expressions to another function that associates those expressions. This special function is derived from the notation of *morphisms* of the category theory.

Using definition 1 and definition 2, we can note $(\mapsto) = x \mapsto (f(x) \mapsto f)$

The formal definition uses Currying to decompose the function into two. It associates a parameter to a function that takes an expression and returns a function.

Next we need to lay off the base definitions of the formalism.

2.2.3 Formalism definition

This functional algebra at the base of our foundation is inspired by *operator algebra* (Takesaki 2013, vol. 125) and *relational algebra* (Jónsson 1984). The problem with the operator algebra is that it supposes vectors and real numbers to work properly. Also, relational algebra, like category theory, presupposes set theory.

Here we define the basic notions of our functional algebra that dictates the rules of the formalism we are defining.

Definition 1 (Currying)**.** Currying is the operation named after the mathematician Haskell Brooks Curry (1958) that allows multiple argument functions in a simpler monoidal formalism. A monome is a function that has only one argument and has only one value, as the axiom of Association.

$(\!|f|\!) = (x \mapsto (\!|f(x)|\!))$

The operation of *Currying* is a function $(\!|)$ that associates to each function f another function that recursively partially applies f with one argument at a time.

If we take a function h such that when having x as a parameter, gives the function g that takes an argument y, *unCurrying* is the function $(\!)$ so that $f(x, y)$ behaves the same way as $h(x)(y)$. We note $h = (\!|f|\!)$.

$(\!|f|\!) = (\!|x, y \mapsto f(x)(y)|\!)^+$

Definition 2 (Application)**.** We note the application of f with an argument x as $f(x)$. The application allows to recover the image y of x which is the value that f associates with x.

$y = f(x)$

Various usages of $()$ range from basic arithmetic to action application in planning (see chapter 4)

Along with Currying, function application can be used *partially* to make constant some arguments.

Definition 3 (Partial Application)**.** We call *partial application* the application using an insufficient number of arguments to any function f. This results in a function that has fewer arguments with the first ones being locked by the partial application. It is interesting to note that Currying is simply a recursion of partial applications.

From now on we will note $f(x, y, z, ...)$ any function that has multiple arguments but will suppose that they are implicitly Curryied. If a function only takes two arguments, we can also use the infix notation e.g. xfy for its application.

$\times(3) = x \mapsto 3 \times x$

Example 4. Applying this to basic arithmetic for illustration, it is possible to create a function that will triple its argument by making a partial application of the multiplication function $\times(3)$ so we can write the operation to triple the number 2 as $\times(3)(2)$ or $\times(2, 3)$ or with the infix notation 2×3.

$\gg = \gg \mapsto \gg$

Definition 4 (Null)**.** The *null function* is the function between nothing and nothing. We note this function $\gg$.

The notation $\gg$ was chosen to represent the association arrow $7\!\!\rightarrow$ but with a dot instead of the tail of the arrow. This is meant to represent the fact that it inhibits associations.

2.2.4 Literal and Variables

As everything is a function in our formalism, we use the null function to define notions of variables and literals.

$l = \gg \mapsto l$

Definition 5 (Literal)**.** A literal is a function that associates null to its value. This consists of any function l written as $\gg \mapsto l$. This means that the function has only itself as an immutable value. We call *constants* functions that have no arguments and have as value either another constant or a literal.

Example 5. A good example of that would be the yet to be defined natural numbers. We can define the literal 3 as $3 = \gg \mapsto 3$. This is a function that has no argument and is always valued to 3.

$x = x \mapsto \gg$

Definition 6 (Variable)**.** A variable is a function that associates itself to $\gg$. This consists of any function x written as $x \mapsto \gg$. This means that the function requires an argument and has undefined value. Variables can be seen as a demand of value or expression and mean nothing without being defined properly.

Example 6. The function f defined in figure 2.2 associates its argument to an expression. Since the argument x is also a variable, the value is therefore dependent on the value required by x. In that example, the number 3 is a literal and $3 \div x$ is therefore an expression using the function $\div$.

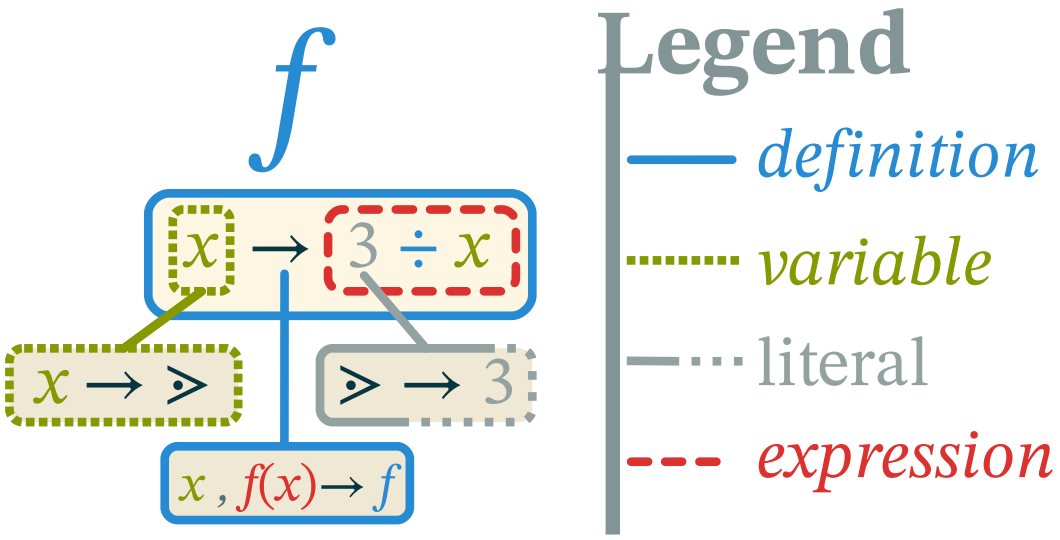


Figure 2.2: Illustration of basic functional operators and their properties.

An interesting property of this notation is that $\geqslant$ is both a variable and a constant. Indeed, by definition, $\geqslant$ is the function that associates $\geqslant \mapsto \geqslant$ and fulfills both definitions.

When defining Currying, we annotated it using the notation $(\!|f|\!) = f \mapsto (x \mapsto (\!|f(x)|\!))$. The obvious issue is the absence of stopping condition in that recursive expression. While the end of the recursion doesn't technically happen, in practice from the way variables and literals are defined, the recursion chain either ends up becoming a variable or a constant because it is undefined when Currying a nullary function.

2.2.5 Functional algebra

Inspired by relational algebra and by category theory, we present a functional algebra that fits our needs. The first operator of this algebra allows to combine several functions into one. This is very useful to merge the definition of two functions in order to specify more complex functions.

Definition 7 (Combination)**.** The *combination function* $\bowtie$ associates any two functions to a new function that is the union of the definition of **either** functions. If both functions are defined for any given argument, then the combination is undefined ($\geqslant$).

$\bowtie = (f, \geqslant \mapsto f) \bowtie (\geqslant, f \mapsto f)$

It is interesting to note that the formal definition of the combination is recursive. This means that it will be evaluated if any of the expression matches, decomposing the functions until one of them isn't defined or until nothing matches and therefore the result is $\geqslant$.

Combination is useful to define boolean constantly in first order logic (section 2.3.1).

Example 7. For two functions f_1 and f_2 that are defined respectively by:

- $f_1 = (1 \mapsto 2) \bowtie (3 \mapsto 4)$
- $f_2 = (2 \mapsto 3) \bowtie (3 \mapsto 5)$

the combination $f_3 = f_1 \bowtie f_2$ will behave as follows:

- $f_3(1) = 2$
- $f_3(2) = 3$
- $f_3(3) = \gg$

$\triangle =$
$(\bowtie) \bowtie (f, f \mapsto f)$

Definition 8 (Superposition)**.** The *superposition function* $\triangle$ associates any two functions to a new function. This function is what the definition of the two functions taken as arguments have in common. The resulting function associates $x \mapsto y$ when **both** functions are superposing. We can say that the superposition is akin to a joint where the resulting function is defined when both functions have the same behavior.

Example 8. Reusing the functions of the previous example, we can note that $f_3 \triangle f_1 = 1 \mapsto 2$ because it is the only association that both functions have in common. We can also say that $f_1 \triangle f_2 = \gg$ because these functions do not share any associations.

Superposition has a "negative" counterpart called a *subposition*. It allows to do the inverse operation of the super position. More intuitively, if the superposition is akin to the set "intersection", the subposition is the "difference" counterpart.

$\nabla = f_1, f_2 \mapsto f_1 \bowtie (f_1 \triangle f_2)$

Definition 9 (Subposition)**.** The *subposition* is the function ∇ that associates any two functions f_1 and f_2 to a new function $f_1 \nabla f_2$. The subposition will allow to "substract" associations from existing functions. The result removes the superposition from the first function definition.

The subposition is akin to a subtraction of function where we remove everything defined by the second function to the definition of the first.

Example 9. From the previous example we can write $f_3 \nabla f_2 = (1 \mapsto 2)$. Since f_2 has the $2 \mapsto 3$ associations in common with f_3 it is removed from the result.

This operation is more useful with the superposition as it can behave like so: $(f_1 \triangle f_2) \nabla f_2 = f_1$. This allows to undo the superposition.

We can also note a few properties of these functions in table 2.1.

Table 2.1: Example properties of superposition and subposition

Formula	Description
$f \triangle f = f$	A function superposed to itself is the same.
$f \triangle \gg = \gg$	Any function superposed by null is null.
$f_1 \triangle f_2 = f_2 \triangle f_1$	Superposition order doesn't affect the result.
$f \nabla f = \gg$	A function subposed by itself is always null.
$f \nabla \gg = f$	Subposing null to any function doesn't change it.

These functions are intuitively the functional equivalent of the union, intersection and difference from set theory. In our formalism we will define the set operations from these.

The following operators are the classical operations on functions.

Composition is useful along especially with the mapping notation to make complex definition more succint.

Definition 10 (Composition)**.** The *composition function* is the function that associates any two functions f_1 and f_2 to a new function such that: $f_1 \circ f_2 = x \mapsto f_1(f_2(x))$.

Definition 11 (Inverse)**.** The *inverse function* is the function that associates any function to its inverse such that if $y = f(x)$ then $x = \bullet(f)(y)$.

We can also use an infix version of it with the composition of functions: $f_1 \bullet f_2 = f_1 \circ \bullet(f_2)$.

These properties are akin to multiplication and division in arithmetic.

Table 2.2: Example of function composition and inverse with their properties.

Formula	Description
$f \circ \gg = \gg$	This means that $\gg$ is the absorbing element of the composition.
$f \circ == f$	Also, $=$ is the neutral element of the composition.
$\bullet(\gg) = \gg \wedge \bullet(=) = (=)$	This means that $\gg$ and $=$ are commutative functions.
$f_1 \circ f_2 \neq f_2 \circ f_1$	However, $\circ$ is not commutative.

From now on, we will use numbers and classical arithmetic as we had defined them. However, we consider defining them from a foundation point of view, later using set theory and Peano's axioms.

In classical mathematics, the inverse of a function f is often written as f^{-1}. Therefore we can define the transitivity of the n^{th} degree as the power of a function such that $f^n = f^{n-1} \circ f$. Figure 2.3 shows how the power of a function is behaving at key values.

By generalizing the formula, we can define the *transitive cover* of a function f and its inverse respectively as $f^+ = f^{+\infty}$ and $f^- = f^{-\infty}$. This cover is the application of the function to its result infinitely. This is useful especially for graphs as the transitive cover of the adjacency function of a graph gives the connectivity function (see section 2.5).

We also call *arity* the number of arguments (or the Currying order) of a function noted $|f|$.

2.2.6 Properties

A modern approach of mathematics is called *reverse mathematics* as instead of building theorems from axioms, we search the minimal set of axioms required by a theorem. Inspired by this, we aim to minimize the formal basis

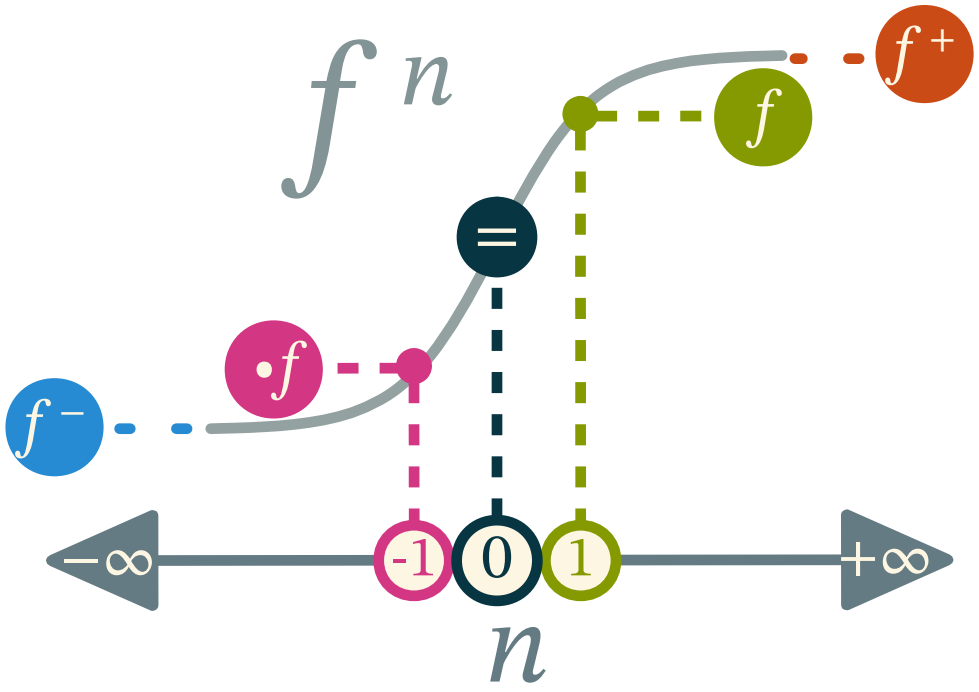


Figure 2.3: Illustration of how the functional equivalent of the power function is behaving with notable values (in filled circles)

of our system as well as identifying the circularity issues, we provide a dependency graph in figure 2.4. We start with the axiom of Association at the bottom and the axiom of Identity at the top. Everything depends on those two axioms but drawing all the arrows makes the figure way less legible.

Then we define the basic application function $()$ that has as complement the Currying $(\!|$ and unCurrying $|\!)$ functions. Similarly, the combination $\bowtie$ has the superposition $\triangle$ and the subpostion $\triangledown$ functions as complements. The bottom bound of the algebra is the null function $\gg$ and the top is the identity function $=$. Composition $\circ$ is the main operator of the algebra and allows it to have an inverse element as the inverse function $•$. The composition function needs the application function in order to be constructed.

The algebra formed by the previously defined operations on functions is a semiring $(\mathbb{F}, \bowtie, \circ)$ with $\mathbb{F}$ being the set of all functions.

Indeed, $\bowtie$ is a commutative monoid having $\gg$ as its identity element and $\circ$ is a monoid with $=$ as its identity element.

Also the composition of the combination is the same as the combination of the composition. Therefore, $\circ$ distributes over $\bowtie$.

At last, using partial application composing with null gives null: $\circ(\gg) = ((\gg) \mapsto \gg) = \gg$.

This foundation is now ready to define other fields of mathematics. We start with logic as it is a very basic formalism in mathematics.

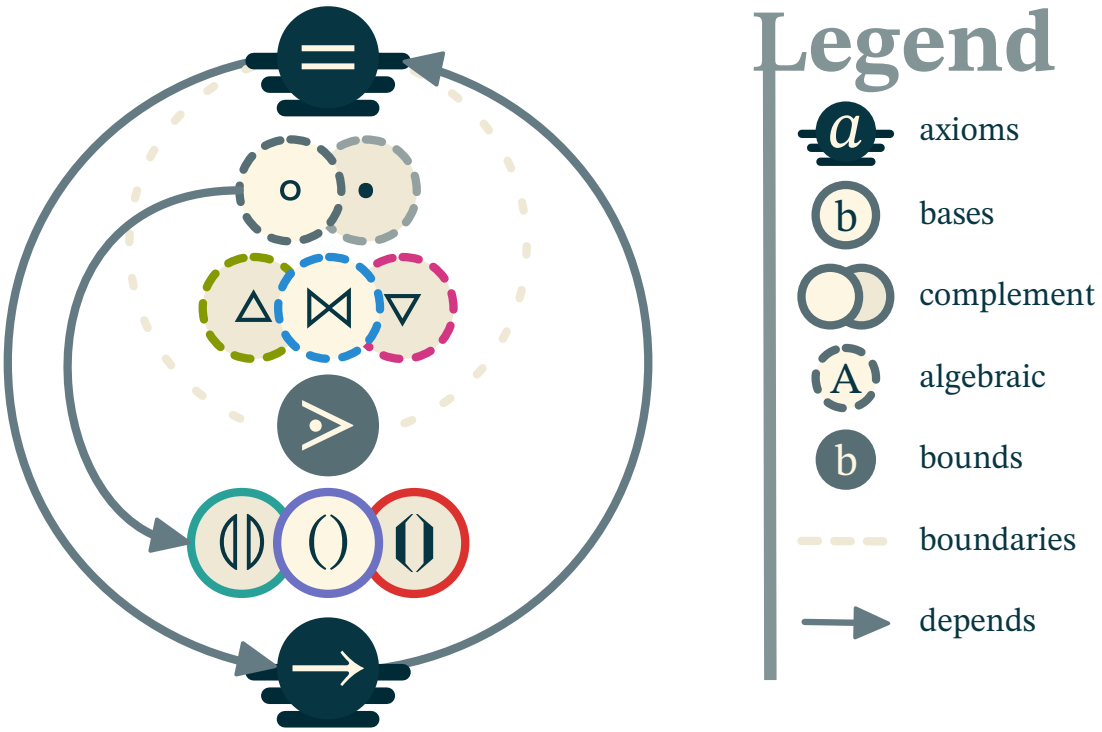


Figure 2.4: Dependency graph of notions in the functional theory

2.3 Logic and reasoning

2.3.1 First Order Logic

First Order Logic

In this section, we present FOL . FOL is based on boolean logic with the two literals *true* (noted $\top$) and *false* (noted $\bot$).

A function noted q that has as only values either $\top$ or $\bot$ is called a **predicate**.

$$\mathcal{D}(\bullet q) = \{\bot, \top\}$$

We define the classical logic *entailment*, the predicate that holds true when a predicate (the conclusion) is true if and only if the first predicate (the premise) is true.

$$\vdash = (\bot, x \mapsto \top) \bowtie (\top, x \mapsto x)$$

Then we define the classical boolean operators $\neg$ *not*, $\wedge$ *and* and $\vee$ *or* as:

- $\neg = (\bot \mapsto \top) \bowtie (\top \mapsto \bot)$, the negation associates true to false and false to true.
- $\wedge = x \mapsto ((\top \mapsto x) \bowtie (\bot \mapsto \bot))$, the conjunction is true when all its arguments are simultaneously true.
- $\vee = x \mapsto ((\top \mapsto \top) \bowtie (\bot \mapsto x))$, the disjunction is true if all its arguments are not false.

The last two operators are curried function and can take any number of arguments as necessary and recursively apply their definition.

Another basic predicate is the **equation**. It is the identity function $=$ but as a binary predicate that is true whenever the two arguments are the same.

Functions that take an expression as parameters are called *modifiers*. FOL introduces a useful kind of modifier used to moralize expressions: *quantifiers*. The quantifiers take an expression and a variable as arguments. Classical quantifiers are also predicates: they restrict the values that the variable can take.

In the realm of FOL, quantifiers are restricted to individual variable (booleans) as follows:

$\forall = \S(\wedge)$
$\exists = \S(\vee)$

- The *universal quantifier* $\forall$ meaning *"for all"*.
- The *existential quantifier* $\exists$ meaning *"it exists"*.

2.3.2 Higher Order Logic

Higher Order Logic

HOL is a class of logic formalism that supersedes FOL. It is, however, less well-behaved than FOL and is not as popular as a consequence. Indeed, HOL allows quantifiers to be applied to sets and even set of sets (see section 2.4). This makes the expressivity of this kind of logic higher but also makes it harder to use and compute.

2.3.3 Modal logic

Section 3.6 will illustrate on an example.

Even bigger than HOL is modal logic. In that logic, quantifiers can be applied to *anything*. The most interesting feature of modal logic is quantifying expressions themselves. This allows for *modality* of statements such as their likelihood, context or to even ask for information.

Using that kind of logic, we can also add some less used quantifiers such as:

$\exists! = \S(= (1) \circ +)$
$\nexists = \S(\neg \circ \wedge)$

- The *uniqueness quantifier* $\exists!$ meaning *"it exists a unique"*.
- The *exclusive quantifier* $\nexists$ meaning *"it doesn't exist"*.

It is also possible to change the nature of quantifiers by using a variable instead of restriction to retrieve a set of values (Hehner 2012):

$\S = f, x, q \mapsto \{\!|f(x) : q|\!\}$

- The *solution quantifier* $\S$ meaning *"those"*.

It is interesting to note that most quantified expression can be expressed using the set builder notation discussed in the following section.

2.4 Set Theory

Since we need to represent knowledge, we will handle more complex data than simple booleans. One such way to describe more complex knowledge is by using set theory. It is used as the classical foundation of mathematics. Most other proposed foundations of mathematics invoke the concept of sets even before their first formula to describe the kind of notions they are introducing. The issue is then to define the sets themselves. At the beginning of his founding 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."*

For Cantor, a set is a collection of concepts and percepts. In our case both notions are grouped in what we call *objects*, *entities* that are all ultimately *functions* in our formalism.

Cantor (1895)

2.4.1 Base Definitions

This part is based on the work of Cantor (1895) and the set theory. The goal is to define the notions of set theory using our formalism.

Definition 12 (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 intention by specifying the rule that all elements follow: $\{n : q(n)\}$

Using our functional foundation, we can define any set as a predicate $\mathcal{S} = e \mapsto \top$ with e being a member of $\mathcal{S}$. This allows us to define the member function noted $e \in \mathcal{S}$ to indicate that e is an element of $\mathcal{S}$.

$$\in = e, \mathcal{S} \mapsto \mathcal{S}(e)$$

Another, useful definition using sets is the *domain* of a function f as the set of all arguments for which the function is defined. We call *co-domain* the domain of the inverse of a function. We can note them $f : \mathcal{D}(f) \to \mathcal{D}(\bullet f)$. In the case of our functional version of sets, they are their own domain.

Definition 13 (Specification)**.** The *function of specification* (noted $:$) is a function that restricts the validity of an expression given a predicate. It can intuitively be read as *"such that"*.

$$:= f, q \mapsto f \triangledown (\mathcal{D}(q = \bot) \to \mathcal{D}(\bullet f))$$

The specification operator is extensively used in classical mathematics but informally, it is often seen as an extension of natural language and can be quite ambiguous. In the present document any usage of $:$ in any mathematical formula will follow the previously discussed definition.

2.4.2 Set Operations

Zermelo–Fraenkel set theory with the axiom of Choice

Along with defining the domains of functions using sets, we can use function on sets. This is very important in order to define ZFC and is extensively used in the rest of the document.

In this section, basic set operations are presented. The first one is the subset.

Definition 14 (Subset)**.** A subset is a part of a set that is integrally contained within it. We note $\mathcal{S} \subset \mathcal{T} \vdash ((e \in \mathcal{S} \vdash e \in \mathcal{T}) \wedge \mathcal{S} \neq \mathcal{T})$, as a set $\mathcal{S}$ is a proper subset of a more general set $\mathcal{T}$.

Definition 15 (Union)**.** The union of two or more sets $\mathcal{S}$ and $\mathcal{T}$ is the set that contains all elements in *either* set. We can note it:

$$\mathcal{S} \cup \mathcal{T} = \{e : e \in \mathcal{S} \vee a \in \mathcal{T}\}$$

Definition 16 (Intersection)**.** The intersection of two or more sets $\mathcal{S}$ and $\mathcal{T}$ is the set that contains only the elements member of *both* set. We can note it:

$$\mathcal{S} \cap \mathcal{T} = \{e : e \in \mathcal{S} \wedge e \in \mathcal{T}\}$$

Definition 17 (Difference)**.** The difference of one set $\mathcal{S}$ to another set $\mathcal{T}$ is the set that contains only the elements contained in the first but not the last. We can note it:

$$\mathcal{S} \setminus \mathcal{T} = \{e : e \in \mathcal{S} \wedge e \notin \mathcal{T}\}$$

An interesting way to visualize relationships with sets is by using Venn diagrams (Venn 1880). In figure 2.5 we present the classical union, intersection and difference operations. It also introduces a new way to represent more complicated notions such as the Cartesian product by using a representation for powerset and higher dimensionality inclusion that a 2D Venn diagram cannot represent.

Example 10. Figure 2.5 is the graphical representation of the statements in table 2.3.

Table 2.3: Caption

Formula	Description
$e_1 \in \mathcal{S}_1$	e_1 is an element of the set $\mathcal{S}_1$.
$e_2 \in \mathcal{S}_1 \cap \mathcal{S}_2$	e_2 is an element of the intersection of $\mathcal{S}_1$ and $\mathcal{S}_2$.
$e_3 \in \mathcal{S}_1 \cap \mathcal{S}_2 \cap \mathcal{S}_3$	e_3 is an element of the intersection of $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$.
$\mathcal{S}_5 \subset \mathcal{S}_2$	$\mathcal{S}_5$ is a subset of $\mathcal{S}_2$.
$\mathcal{S}_6 \subset \mathcal{S}_2 \cup \mathcal{S}_3$	$\mathcal{S}_6$ is a subset of the union of $\mathcal{S}_2$ and $\mathcal{S}_3$.
$f = \mathcal{S}_5 \to \mathcal{S}_6$ f	is a function which domain is $\mathcal{S}_5$ and co-domain is $\mathcal{S}_6$.
$\mathcal{S}_4 \subset \wp(\mathcal{S}_1)$	$\mathcal{S}_4$ is a combination of elements of $\mathcal{S}_1$.

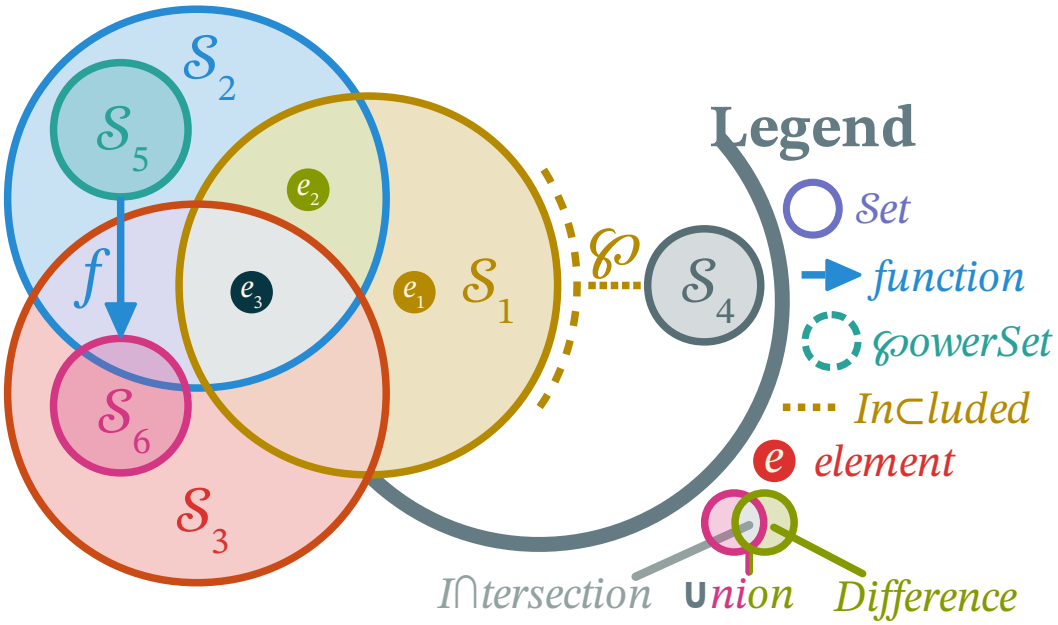


Figure 2.5: Example of an upgraded Venn diagram to illustrate operations on sets.

These Venn diagrams, originally have a lack of expressivity regarding complex operations on sets. Indeed, from their dimensionality it is complicated to express numerous sets having intersection and disjunctions. For example, it is difficult to represent the following notion.

Definition 18 (Cartesian product)**.** The Cartesian product of two sets $\mathcal{S}$ and $\mathcal{T}$ is the set that contains all possible combinations of an element of both sets. These combinations are a kind of ordered set called *tuples*. We note this product:

$$\mathcal{S} \times \mathcal{T} = \{\langle e_\mathcal{S}, e_\mathcal{T} \rangle : e_\mathcal{S} \in \mathcal{S} \wedge e_\mathcal{T} \in \mathcal{T}\}$$

From this 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 Cartesian product can be seen as the set equivalent of Currying. The angles $\langle\rangle$ notation is used for tuples, those are another view on Currying by replacing several arguments using a single one as an ordered list. A tuple of two elements is called a *pair*, of three elements a *triple*, etc. We can access elements in tuples using their index in the following way $e_2 = \langle e_1, e_2, e_3 \rangle_2$. By decomposing the tuples as sets we can write:

$$\mathcal{S} \times \mathcal{T} = e_\mathcal{S}, e_\mathcal{T} \mapsto \mathcal{S}(e_\mathcal{S}) \wedge \mathcal{T}(e_\mathcal{T})$$

Definition 19 (Mapping)**.** The mapping notation $\{\!|\}$ is a function such that $\{\!|f(x) : x \in \mathcal{S}|\!\}$ will give the result of applying all elements in set $\mathcal{S}$ as arguments of the function using the unCurrying operation recursively. If the function isn't specified, the mapping will select a member of the set non deterministically. The function isn't defined on empty sets or on sets with fewer members than arguments of the provided function.

Example 11. The classical sum operation on numbers can be noted:

$$\sum_{i=1}^{3} 2i = \{\!|+(2 * i) : i \in [1,3]|\!\} = +(2 * 1)(+(2 * 2)(2 * 3))$$

2.4.3 The ZFC Theory

The most common axiomatic set theory is ZFC (Kunen 1980, vol. 102). In that definition of sets there are a few notions that come 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)**.** $\forall \mathcal{S} \forall \mathcal{T} : \forall e((e \in \mathcal{S}) = (e \in \mathcal{T})) \vdash \mathcal{S} = \mathcal{T}$

This means that two sets are equal if and only if they have all their members in common.

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 equal if and only if they have precisely the same elements, the empty set is unique.

The definition by intention uses the set builder notation to define a set. It is composed of an expression and a predicate q that will make any element e in a set $\mathcal{T}$ satisfying its part of the resulting set $\mathcal{S}$, or as formulated in ZFC:

Axiom (Specification)**.** $\forall q \forall \mathcal{T} \exists \mathcal{S} : (\forall e \in \mathcal{S} : (e \in \mathcal{T} \wedge q(e)))$

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.

2.5 Graphs

With set theory, it is possible to introduce all of standard mathematics. A field of interest for this thesis is the study of the structure of data. This interest arises from the need to encode semantic information in a knowledge base using a very simple language (see chapter 3). Most of these structures use graphs and isomorphic derivatives.

Definition 20 (Graph)**.** A graph is a mathematical structure g which is defined by its *connectivity function* χ that links two sets into a structure: the edges E and the vertices V.

2.5.1 Adjacency, Incidence and Connectivity

Definition 21 (Connectivity)**.** The connectivity function is a combination of the classical adjacency and incidence functions of the graph. It is defined using a circular definition in the following way:

- *Adjacency*: $\chi_{\circ\!-\!\circ} = v \mapsto \{e : v \in \chi_{-\!\bullet}(e)\}$
- *Incidence*: $\chi_{-\!\bullet} = e \mapsto \{v : e \in \chi_{\circ\!-\!\circ}(v)\}$

Also: $\chi_{\circ\!-\!\circ} = \bullet\chi_{-\!\bullet}$

Defining either function defines the graph. For convenience, the connectivity function combines the adjacency and incidence:

$$\chi = \chi_{\circ\!-\!\circ} \bowtie \chi_{-\!\bullet}$$

Usually, graphs are noted $g = (V, E)$ with the set of vertices V (also called nodes) and edges E (arcs) that links two vertices together. Each edge is classically a pair of vertices ordered or not depending on whether the graph is directed or not. It is possible to go from the set based definition to the functional relation using the following equation: $\mathcal{D}(\chi_{-\!\bullet}) = E$

$E \subseteq V^2$

Example 12. A graph is often represented with lines or arrows linking points together like illustrated in figure 2.6. 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.

2.5.2 Digraphs

The digraphs or *directional graphs* are a specific case of graphs where *all* edges have a direction. This means that we can have two vertices v_1 and v_2 linked by an edge and while it is possible to go from v_1 to v_2, the inverse is impossible. For such case the edges are ordered pairs and the incidence function can be decomposed into:

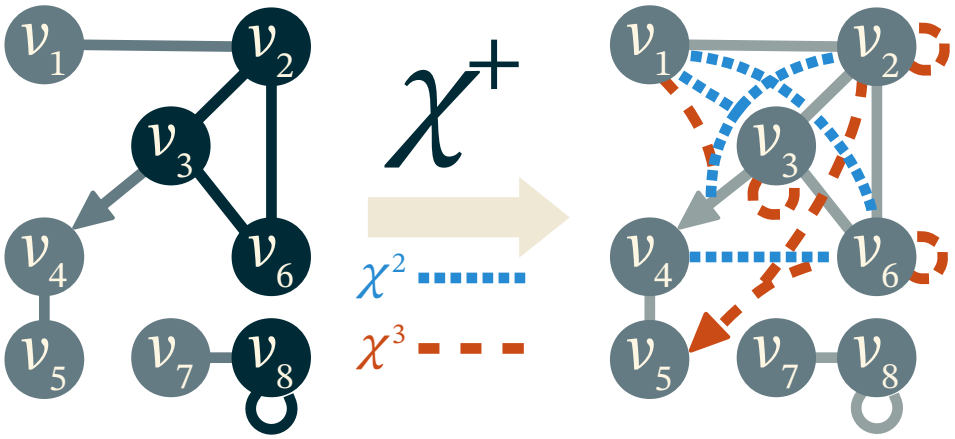


Figure 2.6: Example of the recursive application of the transitive cover to a graph.

$$\chi_{\multimap} = \chi_{\rightarrowtail} \bowtie \chi_{\multimap}$$

We note $\chi_{\rightarrowtail}$ the **incoming relation** and $\chi_{\multimap}$ the **outgoing relation**.

In digraphs, classical edges can exist if allowed and will simply be bidirectional edges.

2.5.3 Path, cycles and transitivity

Most of the intrinsic information of a graph is contained within its structure. Exploring its properties requires 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 $g = (V, E)$.

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 2.6. Note how there is no edge in $\chi^2(g)$ between v_5 and v_6 and the one in $\chi^3(g)$ is directed toward v_5 because there is no path back to v_6 since the edge between v_3 and v_4 is directed. Intuitively, the different powers of the connectivity graph χ^n, are representation of all destinations within a distance n from any vertices. Extending that notion to $n = \infty$ we can define the following:

Definition 22 (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 $\langle v_1, v_2 \rangle \in \mathcal{D}(\chi^+(g))$.

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

Similarly we define *cycles* as the existence of a path from a given vertex to itself. For example, in figure 2.6, the cycles of the original graph are colored in blue. Some graphs can be strictly acyclical, enforcing the absence of cycles.

2.5.4 Trees

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 edges 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 2.7 we illustrate how and/or trees are often depicted.

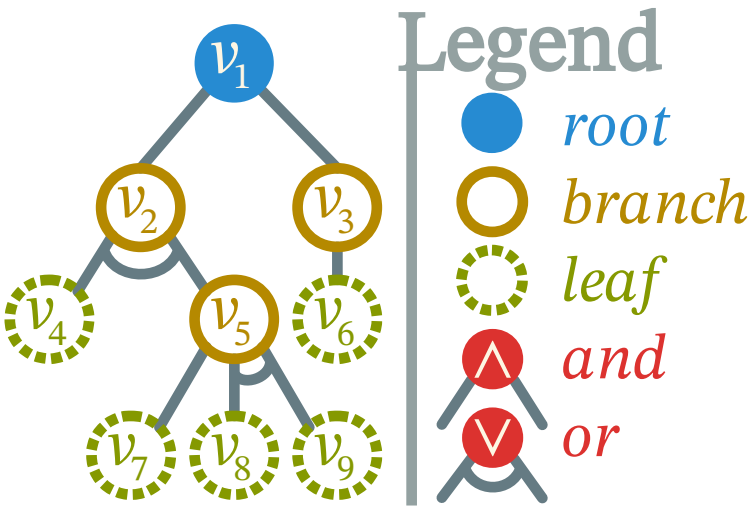


Figure 2.7: Example of and/or tree.

2.5.5 Quotient

Another notion often used for reducing big graphs is the quotiening as illustrated in figure 2.8.

Definition 23 (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 $\div_f(g) = (\{f(v) : v \in V\}, \{f(e) : e \in E\})$ with f being a function that maps any vertex either toward itself or toward its quotiened vertex.

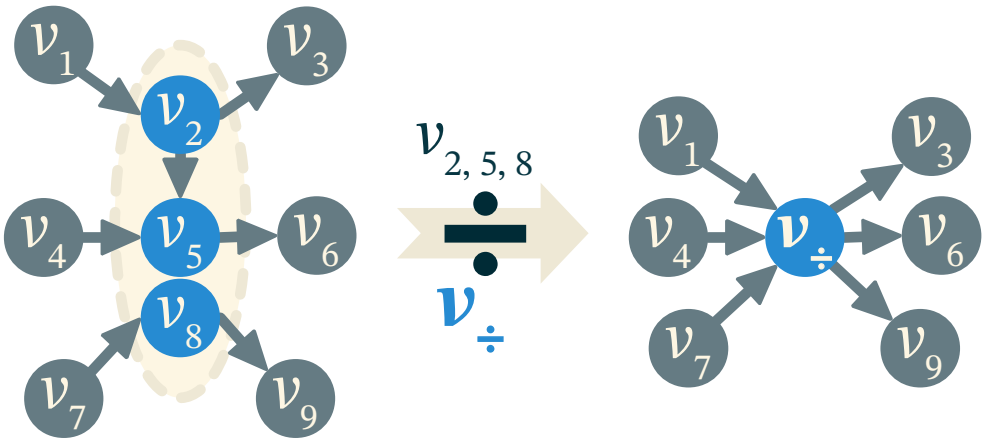


Figure 2.8: Example of graph quotient.

A quotient can be thought of as the operation of merging several vertices into one while keeping their connections with other vertices.

Example 13. Figure 2.8 explains how to do the quotient of a graph by merging the vertices v_2, v_5 and v_8 into $v_\div$. The edge between v_2 and v_5 is lost since it is inside the quotienned part of the graph. All other edges are now connected to the new vertex $v_\div$.

2.5.6 Hypergraphs

A generalization of graphs are **hypergraphs** where the edges are allowed to connect to more than two vertices (Ray-Chaudhuri and Berge 1972). They are often represented using Venn-like representations but can also be represented with edges "gluing" several vertex like in figure 2.9.

Example 14. In figure 2.9, vertices are the discs and edges are either lines or gluing surfaces. In hypergraph, classical edges can exist like e_4, e_6 or e_7. Taking for example e_1, we can see that it connects 3 vertices: v_1, v_2 and v_3. It is also possible to have an edge connecting edges like e_8 that connects e_3 to itself. Edges can also "glue" more than two edges like e_2 connects e_1, e_3 and e_4. The most exotic structures are edge-loops as seen with e_9 and e_{10} which allow graphs that are only made of edges without any vertices.

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.

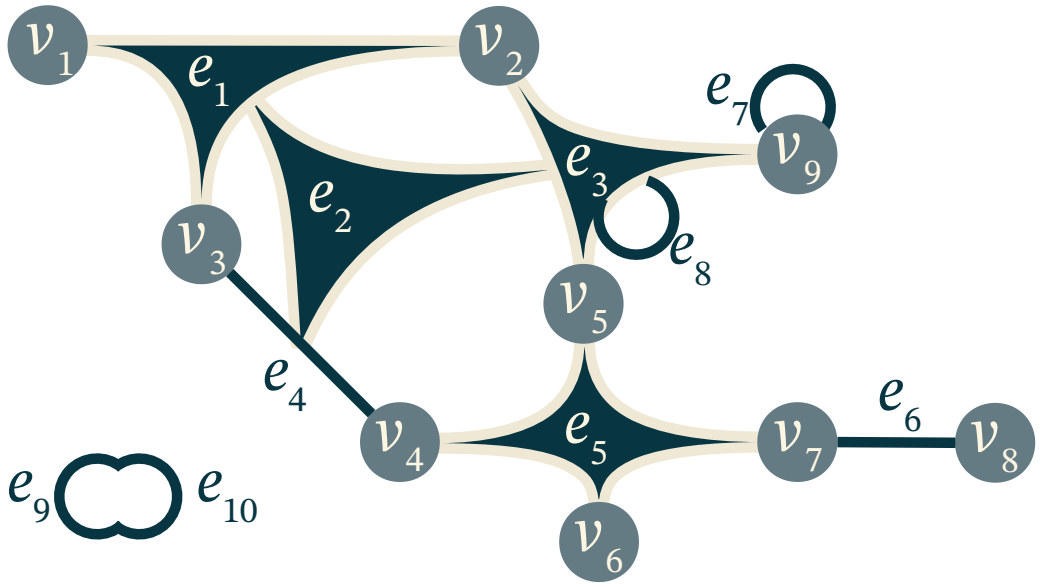


Figure 2.9: Example of hypergraph with total freedom on the edges specification.

Hypergraphs have a special case where $E \subset V$. This means that edges are allowed to connect to other edges. In figure 2.9, this is illustrated by the edge e_2 connecting to three other edges. That type of edge-graphs are akin to port graphs (Silberschatz 1981). An interesting discussion about the compatibility of hypergraphs with ZFC is presented by Vepstas (2008). He said 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 2.9, 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 limitations of FOL and ZFC based models, particularly in the field of knowledge representation. Some structures require higher dimensions as proposed by HOL , modal logic and hypergraphs. However, it is important to note that these models are more general than those based on FOL and ZFC . Indeed, these models contain what is possible to represent in a classical way but remove restrictions specific to these models.

2.6 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.

Definition 24 (Seed)**.** A seed corresponds to a vertex along with the set of adjacent edges. Formally we note a seed $\star = (v, \chi_g(v))$ that means that a

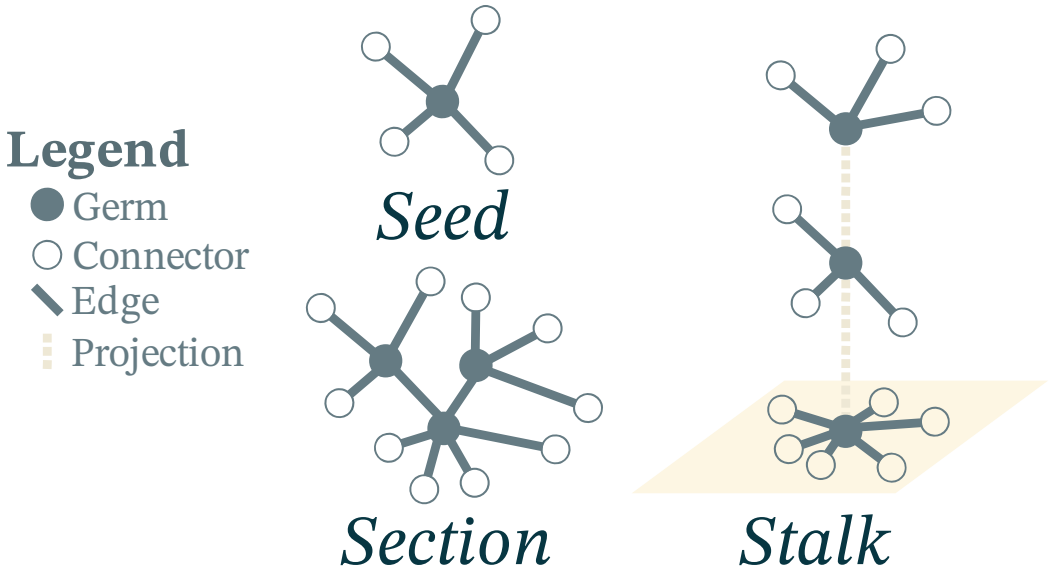


Figure 2.10: Example of a seed, a section and a stalk.

seed built from the vertex v in the graph g contains a set of adjacent edges $\chi_g(v)$. We call the vertex v the *germ* of the seed. All edges in a seed do 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 contain all information about a vertex. Illustrated in the figure 2.10, 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 25 (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 $g_\star = (V, \{\!\{\cup : E_{section}\}\!\})$ the graph formed by the section.

In figure 2.10, 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 the graph quotient is often used in that domain, it was transposed to sections. Quotients allow us to define stalks.

Definition 26 (Stalk)**.** Given a projection function $f : V \to V'$ over the germs of a section $\star$, the stalk above the vertex $v' \in V'$ is the quotient of all seeds that have their germ follow $f(v) = v'$.

The quotienning is used in stalks for their projection. Indeed, as shown in figure 2.10, 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.

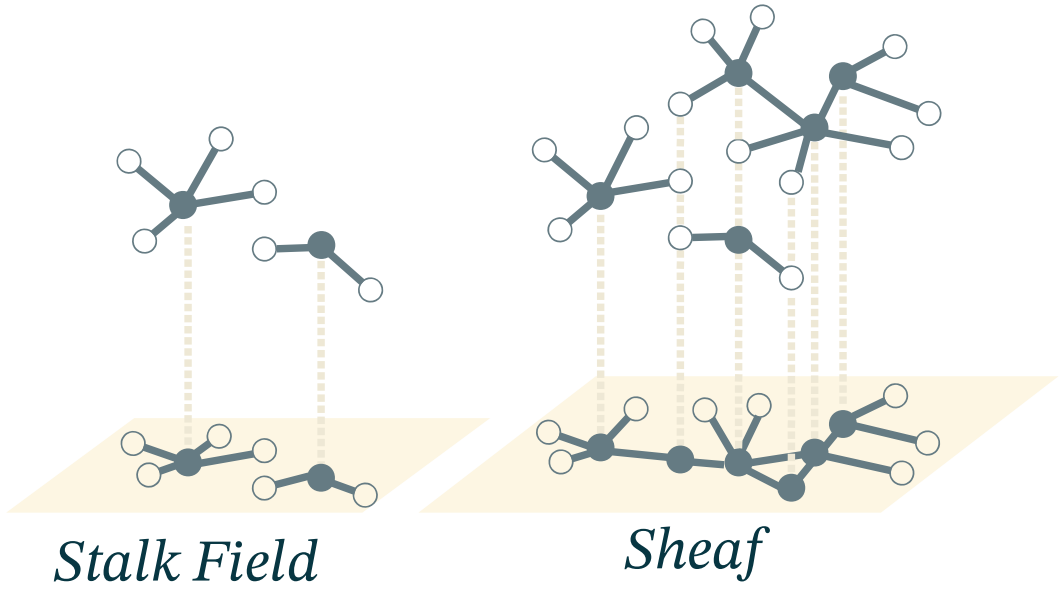


Figure 2.11: Example of sheaves.

Definition 27 (Sheaf)**.** A sheaf is a collection of sections, together with a projection. We note it $\mathcal{F} = \langle g_\star, glue \rangle$ with the function $glue$ being the gluing axioms that the projection should respect depending on the application. The projected sheaf graph is noted as the fusion of all quotiened sections:

$$glue_{\mathcal{F}} = \{\div_{glue_\star} : \{glue_\star \in g_\star\}$$

By putting several sections into one projection, we can build stack fields. These fields are simply a subcategory of sheaves. Illustrated in figure 2.11, a sheaf is a set of sections with a projection relation that usually merges similarly typed connectors.

2.7 Conclusion

In this chapter, we presented the tools we will use for the rest of the document along with its underlying formalism. First we presented a functional

theory that allows for a concise expression of the formula for our usage. We also described classical mathematical tools like FOL , set theory and graphs. In parallel, we introduced non-classical tools of higher order such as HOL , modal logic, hypergraphs and sheaves. Those notions are mostly data structures and allow to express any model needed for our usage.

The first of these models is about a partial self described language for knowledge representation.

3 Knowledge Representation

Figure 3.1: Noam Chomsky 2017

Knowledge representation is at the intersection of maths, logic, language and computer sciences. 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 production rules aiming to describe all accepted input strings. Usually, the rules are hierarchical and deconstruct the input using simpler rules until it matches a terminal symbol. 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 BNF metalanguage was created on top of Chomsky's formalization.

Backus-Naur Form

A similar process happened in the 1970s, when logic based knowledge representation gained popularity among computer scientists (Baader *et al.* 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 FOL. Then, around the 1990s, the research began to merge in search of common semantics in what led to the development of DL. This domain is expressing knowledge as a hierarchy of classes containing individuals.

Description Logic

From there and with the advent of the world wide web, actors of the internet 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 21st century, several languages were created, all based on the W3C specifications called RDFs (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 OWL (Horrocks *et al.* 2003). The 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.

World Wide Web Consortium

Ontology Web Language

Nowadays, when designing a knowledge representation, one usually starts with existing framework. The most popular in practice is certainly the classical relational database, followed closely by more novel methods for either big data or more expressive solutions like ontologies.

In this chapter, we present a new tool that is more expressive than ontologies while remaining efficient. This model is based on dynamic grammar and

basically is defined mostly by the structure of knowledge. Our model is inspired from RDF triplets, especially in its Turtle syntax (W3C 2014). Of course, this will lead to compromises, but can also have some interesting properties. This knowledge representation system will allow us to express hierarchical planning domains in chapter 6.

Resource Description Framework

3.1 Grammar and Parsing

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

Context-Free Grammar

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.

3.1.1 Backus-Naur Form

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

```
1 <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 section, we introduce 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.

Example 15. We can make a grammar for all sequence of `A` using the rule `<scream> ::= "A"+`. If we want to make a rule that prevent the use of the letter `z` we can write `<no-sleep> ::= ~"z"`.

3.1.2 Tools for text analysis

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 and explain our choice of grammatical framework.

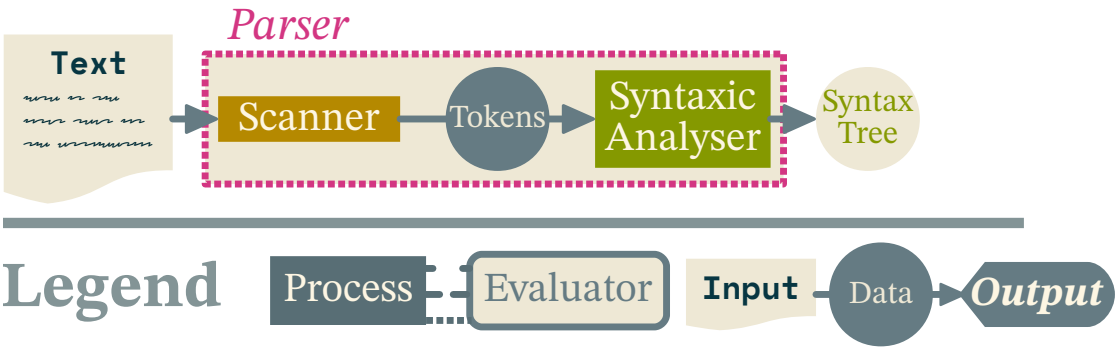


Figure 3.2: Process of a parser while analyzing text

One of the main tools for both static and dynamic grammar is a parser. It is the program that will decode the input into a *syntax tree*. This process is detailed in figure 3.2. To do that it first scans the input text for matching *tokens*. Tokens are akin to words and are the data unit at the lexical level. Then the tokens are matched against production rules of the parser (usually in the form of a grammar). The matching of a rule will add an entry into the resulting tree that is akin to the hierarchical grammatical description of a sentence (e.g. proposition, complement, verb, subject, etc.).

Most of the time, a parser will be used with an *evaluator*. This component transforms the syntax tree into another similarly structured result. 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 compiler simpler. We call those tools compiler-compilers or parser generators (Paulson 1982). They take a grammar description as input and gives the program of a compiler of the generated language as an output. Figure 3.3 explains how both the generation and resulting program work. Each of them uses a parser linked to an *evaluator*. In the case of a compiler-compiler, the evaluator is actually a compiler process. It will transform the syntax tree of the grammar into executable code. This code is the generated compiler and is subject to our interest in this case.

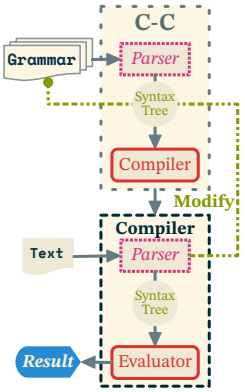


Figure 3.4: Dynamic grammar modification

3.1.3 Dynamic Grammar

One of the issues with classical grammars is that they are set in stone once compiled. One cannot change the definition of a grammar without editing the grammar definition and compiling it. Although this might be more than sufficient for most usages, it can hinder the adaptability of a general-purpose tool. In this section we present the existing types of dynamic grammar and their advantages and limitations.

For dynamic grammar, compilers can get more complicated. The most straightforward way to make a parser able to handle a dynamic grammar is to introduce code in the rule handling that will tweak variables affecting the parser

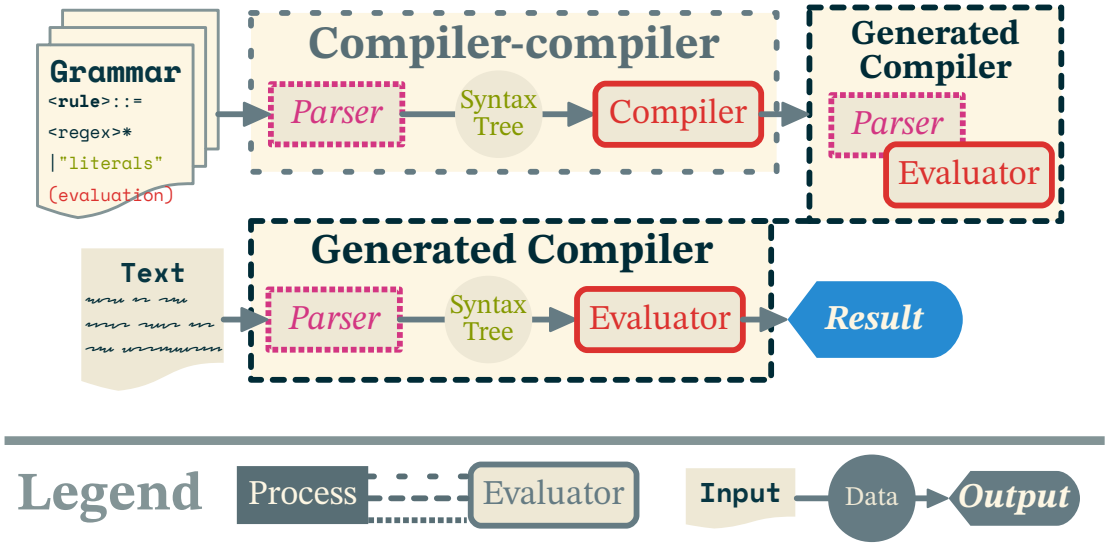


Figure 3.3: Illustration of the meta-process of compiler generation

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). Reusing our prior illustration, we can show in figure 3.4, the particularity of this type of grammar. This approach uses PEG (Ford 2004) with Packrat parsing that 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 offsets the computation load in memory making it actually less efficient in general (Becket and Somogyi 2008).

Parsing Expression Grammar

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.

My system uses a grammar, composed of classical rules and is extended using meta-rules that activate once the classical grammar fails.

3.2 Description Logics

One of the most standard and flexible way of representing knowledge is by using ontologies. They are based mostly on the formalism of DL (Krötzsch *et al.* 2013). 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 have 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 DL to store statements into three boxes (Baader *et al.* 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.

Example 16. In the classical genealogy example (Baader *et al.* 2003), the TBox can be a statement similar to $\text{Woman} = \text{Person} \cap \text{Female}$. This is reasoning about the concept hierarchy and is usually modeled at design time.

The RBox is often not present in DL systems but have interesting expressivity properties. For example, it is possible to define the atomic role gender so that $\text{Person} \cap \forall \text{gender} \in \{\text{Male}, \text{Female}, \text{NonBinary}\}$. This will enforce that every person should have one of the three genders exposed in the set.

The ABox is about instances like $\text{Female} \cup \text{Person}(\text{ALICE})$ stating that Alice is a female and a person. This statement allows the system to infer that $\text{Woman}(\text{ALICE})$ by applying the rules of the TBox and RBox.

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

3.3 Ontologies and their Languages

Most AI problem needs a way to represent knowledge. The classical way to do so has been more and more specialized for each AI community. Every domain uses its DSL that neatly fits the specific use it is intended to do.

Domain Specific Language

There was a time when the branch of AI wanted to unify knowledge description under the banner of the "semantic web". This domain has given numerous works on service composition that is very close to hierarchical planning (Rao *et al.* 2004).

From numerous works, a repeated limitation of the "semantic web" seems to come from the languages used (Dornhege *et al.* 2012; Hirankitti and Xuan 2011). 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.

eXtensible Markup Language

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 relation that associates a unique string called a URI to most nodes. 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.

Uniform Resource Identier

Example 17. To illustrate how RDF is used, we present in listing 3.1, an example from the W3C (2004a). This example is from the famous "Library" example commonly used to explain relational databases. The short triple representation in that listing is possible thanks to the Turtle variant of RDF (Beckett and Berners-Lee 2011). This example also shows the use for properties from the `rdf:` namespace. These properties are fundamental to RDF and allow standard description of basic properties such as the type.

```
ex:ontology rdf:type owl:Ontology .
ex:name rdf:type owl:DatatypeProperty .
ex:author rdf:type owl:ObjectProperty .
ex:Book rdf:type owl:Class .
ex:Person rdf:type owl:Class .

_:x rdf:type ex:Book .
_:x ex:author _:x1 .
_:x1 rdf:type ex:Person .
_:x1 ex:name "Fred"^^xsd:string .
```

Listing 3.1: Example of RDF turtle ontology

Built on top of RDF, the W3C recommended another standard called OWL (W3C 2012). 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, 1,p825). OWLcomes 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 4^{th} field in RDF. This field is used for information about any statement represented as a triple (or 3 fields as the subject property and object). These include context, annotations, access rights, 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). The reification is a compound process. It needs two steps:

- *abstraction* or generalization of the relational structure of a concept. It can be seen as an imperfect description of an object.
- *symbolization* or referring to another structure as being equivalent to a single object. It can be seen as "compressing" the information into one symbol.

In RDF, reification is the act of describing a statement using special relations such as `rdf:subject`, `rdf:property` and `rdf:object`. Then the node describing the statement is typed as a statement and can be used in high order knowledge. Consequently, by identifying each statement, it becomes possible to efficiently form statements about any statements.

Reifying isn't the only way to express reflexivity in ontologies. In the work of Toro *et al.* (2008), the solution explored is to encode queries into the ontology. This allows for query caching and certainly adds to the expressivity. However, encoding queries will only be relevant once queries are already executed.

3.4 Limits

The issue with using these classical tools is that they are very hard to combine. Indeed, making ontologies with a dynamic grammar is out of the question when using the main ontology frameworks. This difficulty is only slightly alleviated when trying to build an ontology framework on top of a dynamic grammar. This lack of adaptability or expressivity is the reason why other approaches must be considered.

Hart and Goertzel (2008) uses a different approach in their framework for AGI called OpenCog. The structure of the knowledge is based on a rhizome, a collection of trees, linked to each other. This structure is called Atomspace. Each vertex in the tree is an atom, leaf vertices are nodes, the others are links. Atoms are immutable, indexed objects. Values 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.

Artificial General Intelligence

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.

In this chapter, we present a similar knowledge structure as AtomSpace that is used along with notions inspired by ontology. The next section presents our contribution toward a knowledge description framework that allows native higher order representation needed for hierarchical planning.

3.5 Structurally Expressive Language Framework

As we have seen, the most used knowledge description systems (e.g. RDF, ontologies and relational databases) have a common drawback: they are static. This means that they are created to be optimized for a specific use case, or gets general at the cost of efficiency.

This issue is mainly due to the lack of flexibility of the language. Since the grammars used for ontologies are static, the language cannot be modified unless manually and by recompiling the tools.

The main issue is that such systems are unable to adapt to the use case by themselves. To fix this issue, a new knowledge representation model is presented. We propose to base our framework on dynamic grammar and exploit the properties of the grammar to make the knowledge description evolve to fit its usage.

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, it 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:

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

Table 3.1: Comparison of different approaches using our criteria.

Approaches	**Neutral**	**Permissive**	**Minimalist**	**Adaptative**
Relational	- -	- - -	- -	-
Triple	+	++	+	+
Ontology	-	+	-	-
AtomSpace	++	+++	- -	++
SELF	+++	+++	++	+++

Table 3.1 presents the fitness of each approach for each criterion. The first approach is the relational database. While widely used, this approach re-

quires an extensive definition of the database schema and, while using simple syntax, is quite verbose in comparison of modern languages. The second approach is the triple representation of RDF. While it allows for more possibilities of expression, it still requires some specific node URIs in order to express higher order knowledge. The ontologies don't have many advantages. They can be more expressive but the added restrictions on interpretation makes it less appealing for our use. Indeed, it needs library worth of specific core URIs and its typing system restricts the ability to abstract even more, especially on OWL Lite. What it gains in speed and desirability, it loses on flexibility. The last existing approach is the AtomSpace of OpenCog. This knowledge base allows for very abstract structures. It, however, is quite heavy and not meant to be directly understood by human readers. This along with the time needed for inference makes it an unfit approach for our objectives.

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

3.5.1 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 supposes that every statement s must be true if it belongs to the domain $\mathcal{D}$. This graph is a directed 3-uniform hypergraph.

See section 2.5.6.

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 by an incoming connector (its subject), an outgoing connector (its object) and a non-directed connector

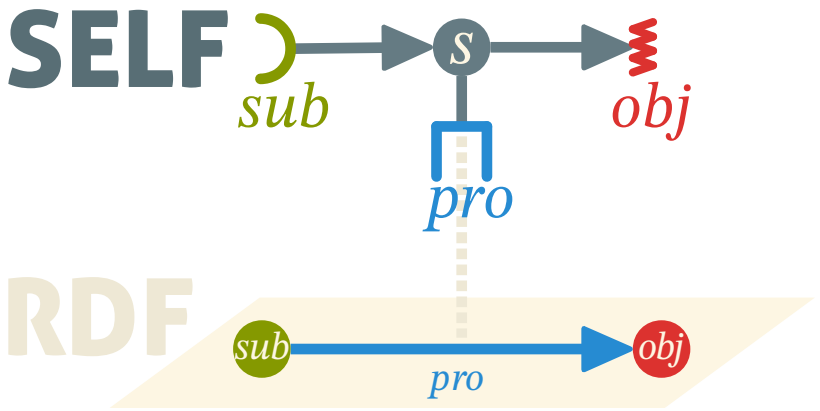


Figure 3.5: Projection of a statement from the SELFs to RDFs space.

(its property). The sections are domains and must be coherent. Each statement, along with its property, makes a stalk as illustrated in figure 3.5.

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.

3.5.1.1 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 require three regular statements just to be defined. 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. From this axiom it is possible to derive theorems that are logical propositions directly deductible from the axioms of a 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 an 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 system and is often a requirement of such constructs. This theorem is also 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.

3.5.1.2 Native Properties

In the following, we suppose all notions from previous chapter. The difference is that we define and use only a subset of the functions defined in the SELF formalism. In relation to the theory of SELF, we use the functional theory previously defined as the underlying formalism.

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

The **equality relation** $=: \mathbb{U} \mapsto \mathbb{U}$, 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** $\nu : \mathbb{U} \mapsto L_{Str}$ that affects a string literal to some entities. This leads us to introduce literals into SELF that are 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 :. It is noted $: : \mathbb{U} \mapsto T$. That relation is complete as all entities have a type. Theorem 3 causes the set of entities to be universal. Type theory, along with DL, introduces a **subsumption relation** $\subseteq: T \mapsto T$ as a partial ordering relation to the types. Since types can be seen as sets of instances, we simply use 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** $\mu : \mathbb{U} \mapsto \mathcal{D}$ with $\mu^{\bullet} = \bullet\mu$. This allows to create domain from certain entities and to encapsulate domains into entities. $\mu^{\bullet}$ 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** $\rho : \mathbb{U} \mapsto \mathbb{U}$. 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. We introduce the **statement relation** $\chi : S \mapsto \mathbb{U}$ 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.

Example 18. Since statements are triplets and edges, s_0 gives the subject of a statement s. Respectively, s_1 and s_2 give the property and object of any statement. For adjacencies, χ can give the set of statements any entity is the object or subject of. For any property pro, the notation $\chi(pro)$ gives the set of statements using this property.

These definitions allow us to build the hypergraph structure by using basic graph formalism.

All of this structure along with the native relations are presented in the table of symbols. Figure 3.6 illustrates the way those sets and relations interact with one another. The Venn diagram of SELF is contained within $\mathbb{U}$ since it is endomorphic. It is interesting to notice that $\mathcal{D}$ is a subset of the powerset of S.

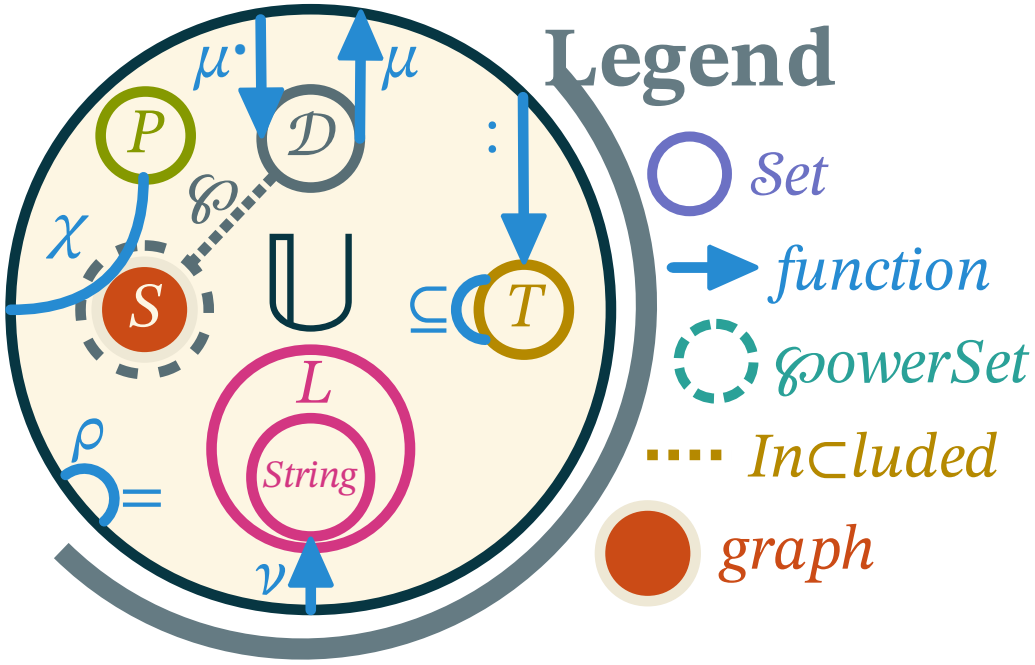


Figure 3.6: Venn diagram of subsets of $\mathbb{U}$ along with their relations. Dotted lines mean that the sets are defined a subset of the wider set.

3.5.2 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 2004b; W3C 2004c). The XML syntax is also quite restrictive and cannot evolve dynamically to adapt to the usage.

The problem is that our principles can be contradictive. Indeed, a general language that is very permissive is often far from minimalist or adaptive. A potential solution would be to use two languages:

- A first one that is general and minimalistic.
- A second one that is permissive and adaptive.

The issue is that we certainly don't want users to have to learn two separate languages to use our framework. Also the second language would be complicated to describe since it will be specific and will need to fit any particular use cases.

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

3.5.2.1 Grammar

The description of $\mathbb{g}$ 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 3.2, we give a simplified version of $\mathbb{g}$. It is written in a pseudo-BNF fashion, which is extended with the classical repetition operators $*$ and $+$ along with the negation operator $\sim$. 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 3.5.3.

```
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 <self> ::= <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 3.2: Simplified pseudo-<+bnf> description for basic <+self>.

In $\mathbb{g}$, 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.

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

The rule at line 10 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 the solution quantifier is presented in section 3.5.5.1. In the following examples we will consider that `<EQUAL> ::= "="`, `<SOLVE> ::= "?"` and `<EOS> ::= ";"`.

At line 11, we can see one of the most defining features of $\mathbb{g}$: statements. The input is nothing but a set of statements. Each component of the statements is an entity. 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.

3.5.2.2 Neutrality and encoding

In order to respect the principle of neutrality, the language must not suppose 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 literals that use the same grammar as in classical Java. This means that the decimal separator is the dot (.). This concession is made for reasons of simplicity and efficiency, but it is possible to define literals dynamically in theory (see section 8.1.1.1).

Unicode Transformation Format on 8-bits

The principle of neutrality makes mandatory to use an extensive character encoding standard in order to support non-roman languages. The best candidate for such an encoding is the UTF-8 (see Unicode Consortium 2018a, chap. 2). This standard is certainly the most used in the world and have a significant impact on the way text is expressed in any written language used nowadays.

The Unicode consortium caters to a database that names, categorizes and describes all characters. That database is called the UCD (Unicode Consortium 2018b).

Unicode Character Database

SELF can work with UTF-8 and exploits the UCD for its token definitions. 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). For efficiency's sake it is still recommended to restrain our use to the ASCII (One of UTF-8's predecessor) subset of characters.

American Standard Code for Information Interchange

White-spaces are defined against UCD's definition of the separator category `Z&` (see Unicode Consortium 2018a, chap. 4).

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 uppercase can be used in that context. For performance and simplicity reasons we will only use ASCII in our examples and application.

3.5.3 Dynamic Grammar

The syntax we described is only valid for $\mathbb{g}$. 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 hande 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 $\mathbb{g}$: 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 $\mathbb{g}$ 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.

3.5.3.1 Containers

The first meta-rule is to infer a container. A container is delimited by, at least, a left and a right delimiter (they can be the same symbol). An optional middle delimiter can also be used but must be different from any other delimiters. We infer the delimiters using the algorithm 1.

Algorithm 1 Container meta-rule

```
1:  function container(Token current)
2:      lookahead(current, EOS)                    ▷ Populate all tokens of the statement
3:      for all token in horizon do
4:          if token is a new symbol then delimiters.append(token)
5:      if length(delimiters) <2 then
6:          if coherentDelimiters(horizon, delimiters[0]) then
7:              inferMiddle(delimiters[0])         ▷ New middle delimiter in existing containers
8:              return Success
9:          return Failure
10:     while length(delimiters) > 0 do
11:         for all (left, middle, right) in sortedDelimiters(delimiters) do
12:             if coherentDelimiters(horizon, left, middle, right) then
13:                 inferDelimiter(left, right)
14:                 inferMiddle(middle)                                  ▷ Ignored if null
15:                 delimiters.remove(left, middle, right)
16:                 break
17:         if length(delimiters) stayed the same then return Success
18:     return Success
```

The algorithm will start at line 4, by searching all new symbols and store them as delimiter candidates in `delimiters`. The function `sortedDelimiters` at line 11 will generate all possible orders to put the delimiters into. It will also sort those combinations from most likely to unlikely. This is done by using the UCD `Bidi_Mirrored` property of paired delimiters (category `Z`) and checking if the order is coherent with the `Bidi_Class`.

Checking the result of the choice is very important. At line 12 the function `coherentDelimiters` checks if the delimiters allow for triple reduction and enforce restrictions.

Example 19. 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.

Once the inference is done, the resulting calls to inferDelimiter will add the rules listed in listing 3.3 to $\mathbb{g}$. 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 (using the property `Name` starting with "left" and the property `Bidi_Class`). 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 <parenthesis> ::= "(" [<naked_entity>] (<?MIDDLE> <naked_entity>)* ")"
4 <naked_entity> ::= <statement> | <entity>
```

Listing 3.3: 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.

Example 20. If we parse the expression `a = (b,c);`, we start by tokenizing it as `<ENTITY> <EQUAL> <SYMBOL><ENTITY><SYMBOL><ENTITY><SYMBOL> <EOS>` (ignoring whitespaces and comments). This means that the statement is 4 tokens too long to form a triple. This triggers a parsing error and then an evaluation using meta-rules. All the `<ID>` tokens are new symbols, but they don't have the same subtype. This means that candidate delimiters are `(`, `(,` and `)`. To infer the correct combination, the left and right delimiters are found via their Unicode description. The comma is left to be inferred as the middle delimiter. The grammar is rewritten and the statement becomes `<ENTITY> <EQUAL> <container> <EOS>` which is a valid triple statement.

3.5.3.2 Parameters

If no viable container has been found, we proceed with the next meta-rule. This rule needs the first one to have been used at least once before being able to work. This meta-rule allows for parameterized entities using containers. A parameter is an ordered list of arguments, just like for functions. Algorithm 2 presents how we infer parameters from invalid statements.

Algorithm 2 Parameter meta-rule

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

This algorithm will search for extra containers in the statement. For each container, the function `coherentParameters` at line 7 will check if the container

can be turned into a parameter for the preceding entity. In order to remove ambiguities, we disallow parameters on containers as well as using containers as properties.

Once a coherent parameter has been found, the container is removed from the statement and added as the preceding entity's parameter. To do so quicker the next time, the function `inferParameter` at line 8 adds that syntax as new rules as illustrated in listing 3.4, replacing `<?container>` with the name of the container used.

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

Listing 3.4: Rules added to the current grammar for handling parameters

Example 21. In this case we have to parse `f(x) = x;`. If we already have the parenthesis delimiter defined, it becomes `<ENTITY> <container> <EQUAL> <ENTITY> <EOS>`. Since the statement isn't a triple, we execute the meta-rules. The container rule finds no new symbols and fails. Then the parameter meta-rule will reduce the statement by bounding the first entity to the following container and mark it as a parameterized entity. This gives `<meta_entity> <EQUAL> <ENTITY> <EOS>`.

3.5.3.3 Modifiers

In some cases, using containers for parameters can become verbose. In order to make that task more concise, it is useful to add *syntactic sugar*. This term refers to a writing convenience that is actually equivalent to a longer or more complex notation. In our case, making containers optional comes mainly from the use of modifiers. So this will be our last meta-rule since it requires parameters to have been used at least once before. In algorithm 3 we explain the process of how the modifier notation is inferred.

Algorithm 3 Modifier meta-rule

```
1:  function modifer(Entity[] statement)
2:      reduced = statement
3:      while length(reduced) >3 do
4:          for i from 0 to length(reduced) - 1 do
5:              if ν(reduced[i]) not null and
6:  ν(reduced[i+1]) not null and
7:  (ν(reduced[i]) is a new symbol or
8:  reduced[i] has been parameterized before) and
9:  coherentModifier(reduced, i) then
10:                 mod = inferModifier(reduced[i], reduced[i+1])
11:                 reduced.remove(reduced[i], reduced[i+1])
12:                 reduced.insert(mod, i)          ▷ Replace parameterized entity
13:                 break
14:         if length(statement) stayed the same  then return Success
15:     return Failure
```

In a very similar way as with algorithm 2, this algorithm will first determine if the proposed modifier is coherent. If the entity has been parameterized before, and the statement is valid after reduction, the syntax will be accepted. From the call of `inferModifier`, comes new rules explicated in listing 3.5. The call also adds the modifier entity to an inferred token `<MOD>`.

```
1 <meta_entity> ::= <?MOD> <ID>
2 <meta_property> ::= <?MOD> <ID>
```

Listing 3.5: Rules added to the current grammar for handling modifiers

Since it is most used for special entities like quantifiers, once used, the parent entity will take a polymorphic type. This means that type inference will not issue errors for any usage of them.

Example 22. With the input `!x = 0;` we parse `<SYMBOL> <ENTITY> <EQUAL> <LITERAL> <EOS>`. This cannot be a container since the new symbol is at the beginning without any mirroring possible. It cannot be a parameter since no container is present. But this will conclude with the modifier meta-rule as the new symbol precedes an entity. This becomes `<meta_entity> <EQUAL> <LITERAL> <EOS>` and becomes a valid statement.

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

3.5.4 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 knowledge description system with the right entities or retrieve an existing one that corresponds to what is being parsed.

When parsing, the rules `<entity>` and `<property>` will trigger the creation or retrieval of an entity. This mechanism will use the name of the entity to retrieve an entity with the same name in a given scope. If no such entity exists it is created and added to the current scope.

3.5.4.1 Naming and Scope

When parsing an entity, 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.

Example 23. In order to make this notion easier to understand, we start with its expected behavior. In figure 3.7 the process of variable qualification is illustrated. In order to become a variable, an entity must fulfill two conditions:

Figure 3.7: Example of scope resolution.

- Be used as a *parameter* in the statement.
- Be mentioned *twice* in the statement.

In our example, `f` has `x` as parameters and `x` is used in the statement inside the contained statement on the right hand of the first statement. Of course we suppose in this example that parentheses are delimiters and that (`;`) is the end of statement tokens.

An important nuance can be shown in the second statement. Indeed, `x` is also a variable but since it is a different global level statement, *it is not resolved as the same variable as the previous* `x`. This means that both variables are **independent**.

In the third statement, `x` is only mentioned once and therefore is now a global symbol. It is still independent from the two previous statements.

A scope is the reach of an entity's direct influence. It affects the naming relation by removing variable 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 variables possible, akin to RDF blank nodes.

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 is temporarily bound to a given entity during the parsing. This scope is limited to the statement being interpreted.
- *Global*: This scope means that 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

Algorithm 4 Determination of the scope of an entity

```
1:  function inferScope(Entity e)
2:      Entity[] reach = []
3:      if : (e) = S then
4:          for all i ∈ χ(e) do reach.append(inferVariable(i))          ▷ Adding scopes nested in statement e
5:      for all i ∈ μ•(e) do reach.append(inferVariable(i))             ▷ Adding scopes nested in container e
6:      if ∃ρ(e) then
7:          Entity[] param = inferScope(ρ(e))
8:          for all i ∈ param do param.remove(inferScope(i))   ▷ Remove duplicate scopes from parameters
9:          for all i ∈ param do reach.append(inferVariable(i))          ▷ Adding scopes from paramters of e
10:     scope(e) ← reach
11:     if GLOBAL ∉ scope(e) then scope(e) ← scope(e) ∪{LOCAL}
    return reach
12: function inferVariable(Entity e)
13:     Entity[] reach = []
14:     if LOCAL ∈ scope(e) then
15:         for all i ∈ scope(e) do
16:             if ∃e_p ∈ 𝕌 : ρ(p) = i then        ▷ e is already a parameter of another entity e_p
17:                 scope(e) ← scope(e) \{LOCAL}
18:                 scope(e_p) ← scope(e_p) ∪ scope(e)
19:                 scope(e) ← scope(e) ∪{VARIABLE, p}
20:         reach.append(e)
21:         reach.append(scope(e))
    return reach
```

scope. When an entity is enclosed in any entity while already being a parameter of another entity, it becomes a variable since it is referenced twice in the same statement.

3.5.4.2 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.

3.5.5 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. An *inference engine* is the term that describes an algorithm used in ontologies to infer new statements from an explicit set of statements known as an ontological database.

In our case, 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 classical 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.

3.5.5.1 Quantifiers

In SELF quantifiers 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 $\mathbb{g}$. It is the language equivalent of μ^* and is used to extract and evaluate reified knowledge (see section 3.5.1.2).

Example 24. 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 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, turns the expression into the possible value range of its variable. It is like replacing it by the natural expression "those x that".
- The **universal quantifier** `<ALL>` behaves like ∀ and forces the expression to take every possible value of its variable.
- The **existential quantifier** `<SOME>` behaves like ∃ and forces the expression to match *at least one* arbitrary value for its variable.
- The **uniqueness quantifier** `<ONE>` behaves like ∃! and forces the expression to match *exactly one* arbitrary value for its variable.
- The **exclusion quantifier** `<NULL>` behaves like ∄ and forces the expression not to match the value of its variable.

The last four quantifiers are inspired from Aristotle's square of opposition (D'Alfonso 2011) as illustrated in figure 3.8.

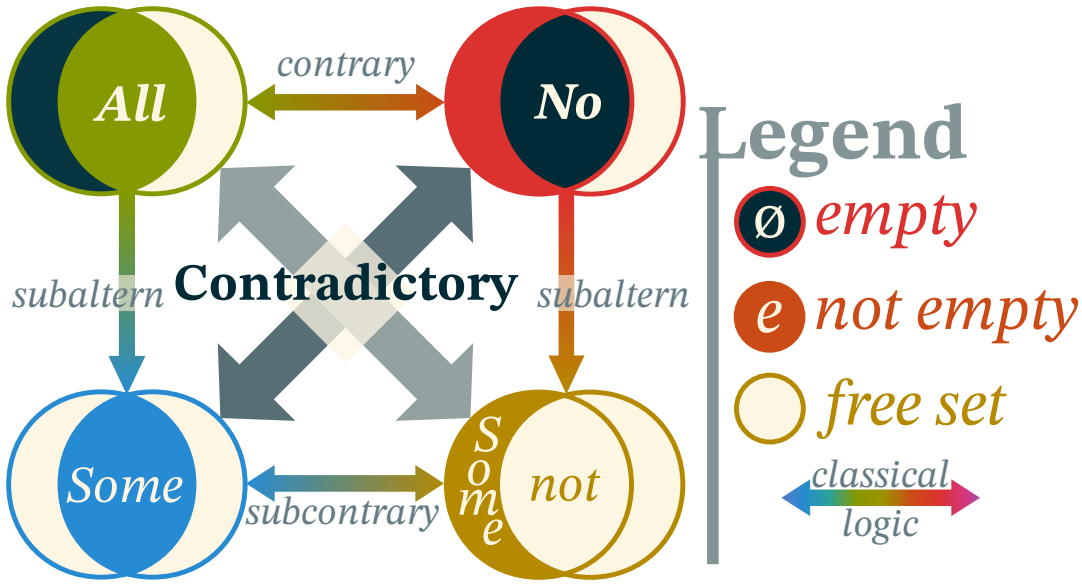


Figure 3.8: Aristotle's square of opposition

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.

Example 25. If we want to express the fact that a glass of water is not empty we can write either `glass contains ~(~);` or `glass ~(contains) ~` with `<NULL> = ~`. This shows that `<NULL>` is used for negation and to indicate the absence of value.

This property is quite handy as it requires only one symbol and allows for complex constructs that are difficult to explain using available paradigms.

In listing 3.6, we present an example file that is meant to define most of the useful native properties along with default quantifiers.

```
1 * =? ;
2 ?(x) = x; //Optional definition
3 ?~ = { };
4 ?_ ~(=) ~;
5 ?!_ = {_};
6
7 (*e, !T) : (e :: T); *T : (T :: Type);
8 *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 3.6: 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. Since domains are sets of statements, this means that anything equaling the solution quantifier at this level will be evaluated as a domain. This is because the entity is a domain **by structure**. 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.

Uniform Resource Locator

If it is a string literal, then it must be either a file path or URL or a valid SELF expression.

Example 26. Using the first statement, we can include external domains akin to the `import` directive in Java. Writing `"path/lang.w" = ? ;` as a first statement will make the process parse the file located at `path/lang.w` and insert it at this spot.

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 do not match other anonymous variables necessarily in the same statement).

In listing 3.6 the type hierarchy can be illustrated by the figure 3.9. It shows how the type `Entity` is the parent of all types. This figure is separated by two axes of symmetry. The vertical separation concern abstraction. Types on the left are used for the Meta relation. The horizontal line distinguishes valuation. Terms on top are externally valued (valued by the context) and terms on the bottom are intrinsically valued (valued by their definition).

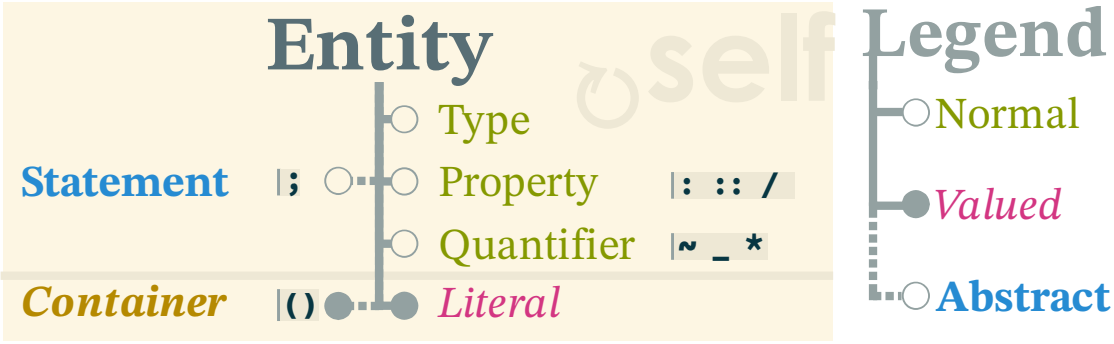


Figure 3.9: Hierarchy of types in SELFs

3.5.5.2 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 of 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 affect all entities to a possible parameter list.
- $:$ (at line 7) is the first property that matches every entity to a type.
- χ (at line 24) is the first property to match for all statements.

We limit the inference to one symbol to eliminate ambiguities and prevent accidental redefinition of native properties. This also improves performance as the inference is stopped after finding a first matching entity that can be used programmatically using a single constant.

3.5.6 Extended Inference Mechanisms

In this section we present the default inference engine. It is quite limited since it is 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 user provided 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 real time.

3.5.6.1 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.

3.5.6.2 Instantiation

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

Since entities are immutable, updating their instance can be quite tricky. 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 realizes 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 to be 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.

3.6 Example

In the following section, a use case of the framework will be presented. First we have to explain a few notions.

3.6.1 Modality of Statements

In the field of logic there exists one special flavor of it called *modal logic*. It lays the emphasis upon the qualifications of statements, and especially the way they are interpreted. This is a very appropriate example for SELF. The modality of a statement acts like a modifier, it specifies a property regarding its plausibility, origin or validity.

Example 27. In the figure 3.10, we present a case of three persons gossiping, Alice, Becky and Carol. The presentation is inspired by the work of Schwarzentruber (2018). Here is a list of the statements in this example:

- Alice said to Becky that Carol should *probably* change her style from C_1 to C_2.
- Becky said to Alice that she finds the Carol's style *usually* good.
- Alice told Carol that Becky told her that she should *sometimes* change her style to C_2.

The following statement can be inferred:

- Carol *possibly* thinks that Becky thinks that the style C_2 is *often* good.

In the example, all modalities are *emphasized*. One can notice an interesting property of these statements in that they are about other statements. This kind of description is called *higher order knowledge*.

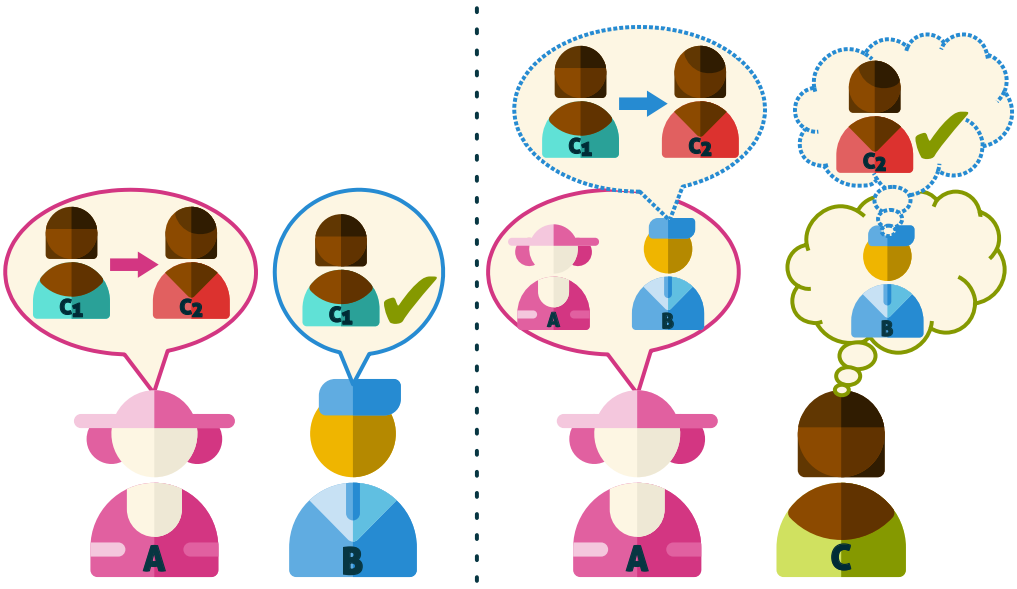


Figure 3.10: Example of modal logic propositions: Alice gossips about what Beatrice said about Claire

3.6.2 Higher order knowledge

SELF is based on the ability to easily process higher order knowledge. In that case the term *order* refers to the level of abstraction of a statement (Schwarzentruber 2018). For such usages, a hypergraph structure is a clear advantage in terms of expressivity and ease of manipulation of those statements. This is due to the higher dimensionality of sheaves (and by extension hypergraphs) that makes meta-statement as simple to express as any other statement. This chain of abstractions using meta-statements is where the higher order knowledge is encoded.

Example 28. We present the previous example using RDF (in listing 3.7) and SELF (in listing 3.8) to describe knowledge of the gossip.

```
@prefix : <http://genn.io/self/gossip#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://genn.io/self/gossip> .

<http://genn.io/self/gossip> rdf:type owl:Ontology ;
                             owl:imports rdf: .

:modality rdf:type owl:AnnotationProperty ;
          rdfs:range :Modality .

:told_a rdf:type owl:ObjectProperty .
:told_b rdf:type owl:ObjectProperty .
:told_c rdf:type owl:ObjectProperty .
:Modality rdf:type owl:Class .
:good rdf:type owl:NamedIndividual .
:is rdf:type owl:NamedIndividual ,
            rdf:Property .
:probably rdf:type owl:NamedIndividual ,
                   :Modality .
:s1 rdf:type owl:NamedIndividual ,
            rdf:Statement ;
    rdf:object :c2 ;
    rdf:predicate :worsethan ;
    rdf:subject :c ;
    :modality :probably .
:s2 rdf:type owl:NamedIndividual ;
    rdf:object :good ;
    rdf:predicate :is ;
    rdf:subject :c ;
    :modality :usually .
:s3 rdf:type owl:NamedIndividual ,
            rdf:Statement ;
    rdf:object :s4 ;
    rdf:predicate :told_a ;
    rdf:subject :b .
:s4 rdf:type owl:NamedIndividual ,
            rdf:Statement ;
    rdf:object :c2 ;
    rdf:predicate :should ;
    rdf:subject :c ;
```

```
    :modality :sometimes .
:should rdf:type owl:NamedIndividual ,
                rdf:Property .
:sometimes rdf:type owl:NamedIndividual ,
                    :Modality .
:told_a rdf:type owl:NamedIndividual ,
                rdf:Property .
:usually rdf:type owl:NamedIndividual ,
                 :Modality .
:worsethan rdf:type owl:NamedIndividual ,
                   rdf:Property .
:a rdf:type owl:NamedIndividual ;
   :told_b :s1 ;
   :told_c :s3 .
:b rdf:type owl:NamedIndividual ;
   :told_a :s2 .
:c rdf:type owl:NamedIndividual .
:c2 rdf:type owl:NamedIndividual .
```

Listing 3.7: RDF representation of the gossip example

```
"lang.s" = ? ;
a told(b) probably(c worsethan ctwo);
b told(a) usually(c is good);
a told(c) (b told(a) sometimes(c should ctwo));
```

Listing 3.8: <+self> representation of the gossip example

It is obvious that the SELF version is an order of magnitude more concise than RDF to express modal logic. The 4 lines of SELF are **equivalent** to the 62 lines of RDF. In the RDF version we use the reified statements `:s1`, `:s2`, `:s3` and `:s4` along with a `:modality` annotation to express high order knowledge and modalities. In SELF, everything is inferred by structure and one can start exploiting their database right away.

3.7 Conclusion

In this chapter, we presented a new endomorphic metalanguage for knowledge description. This language along with its framework allows for extended expressivity and higher order knowledge. This framework was needed to overcome the limitations of classical knowledge representation tools, mainly in order to encode hierarchical planning domains into human and computer-readable text.

In the following chapter we will show an application of this framework to encode planning domains. This application allows to transpose a general planning framework into a specialized language using SELF.

4 General Planning Formalism

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

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

In this chapter, we will start with a popular planning problem as an example of what planning is about. Then, we will formalize general notions of planning using the functional formalism of chapter 2. This leads to our contribution: by factorizing how planners search for a result, it becomes possible to formalize all planning paradigms into one unified approach. To finish we will show how this formalism can be applied to every single planning paradigm.

4.1 Illustration

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

Example 29. In this example, a robotic grabbing arm tries to stack blocks on a table in a specific order. The arm is only capable of handling one block at a time. We suppose that the table is large enough so that all the blocks can be put on it without any stacks. Figure 4.1 illustrates the setup of this domain.

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

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

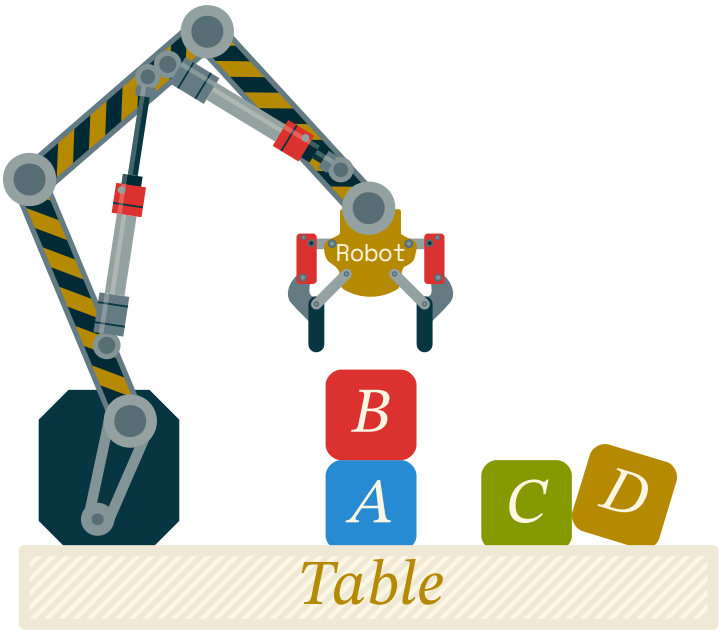


Figure 4.1: The block world domain setup.

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

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

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

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

Example 30. For example, in the block world domain we can have an initial state with the $blockB$ on top of $blockA$ and the $blockC$ being on the table. In figure 4.2, we give a plan solution to the problem consisting of having the stack $\langle blockA, blockB, blockC \rangle$ from that initial state.

All automated planners aim to find such a solution to their planning problem. The main issue is that planning problem quickly becomes very expensive to solve. This is why planners are often evaluated on time and solution quality. The quality of a plan is often measured by how hard it is to execute, whether by its execution time or by the resources needed to accomplish it. This metric

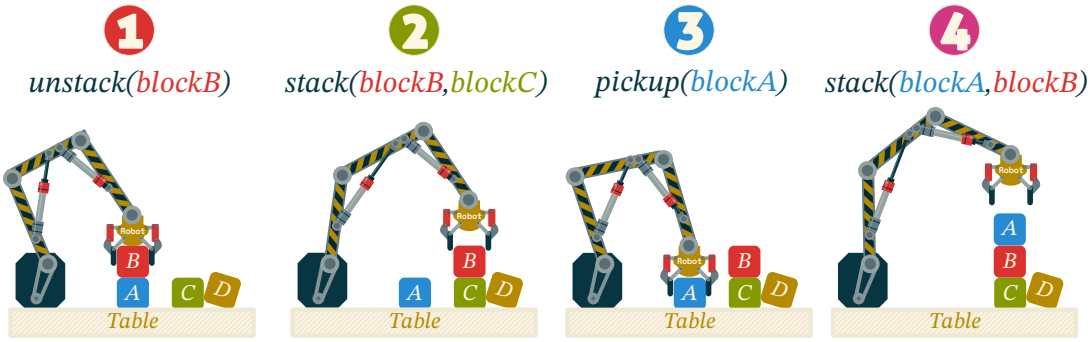


Figure 4.2: An example of a solution to a planning problem with a goal that requires three blocks stacked in alphabetical order.

is often called the *cost* of a plan and is often simply the sum of the costs of its actions.

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

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

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

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

However, in the next section we aim to define such a general planning formalism. The main goal is to provide the automated planning community with a general unifying framework.

4.2 Formalism

In this section, we will present how automated planning works by using the formalism we first described in chapter 2. This leads to a general formalism of automated planning. The goal is to explain what is planning and how it works. First we must express the knowledge domain formalism, then we describe how problems are represented and lastly how a general planning algorithm can be envisioned.

4.2.1 Planning domain

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

4.2.1.1 Fluents

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

Definition 28 (Fluent)**.** A planning fluent is a predicate $f \in F$.

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

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

Example 31. In our example, back in section 4.1, we have three predicates: `down`, `held` and `on`. They can form many fluents like $held(no - block)$, $on(blockA, blockB)$ or $\neg down(blockA)$. When expressing a fluent we suppose its truth value is $\top$ and denote falsehood using the negation $\neg$.

4.2.1.2 States

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

Definition 29 (State)**.** A state is a logical formula of fluents. Since all logical formulas can be reduced to a simple form using only $\wedge$, $\vee$, and $\neg$, we can represent states as *and/or trees*. This means that the leaves are fluents and the other nodes are states. We note states using small squares $\square$ as it is often the symbol used in the representation of automates and grafcets.

Example 32. In the domain block world, we can express a couple of states as :

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

In such a case, both states $\square_1$ and $\square_2$ have their truth value being the conjunction of all their fluents. We can express a disjunction in the following way: $\square_3 = \square_1 \vee \square_2$. In that case, $\square_3$ is the root of the and/or tree and all its direct children are or vertices. The states $\square_1$ and $\square_2$ have their children as *and vertices*. All the leaves are fluents. This tree is presented in the figure 4.3.

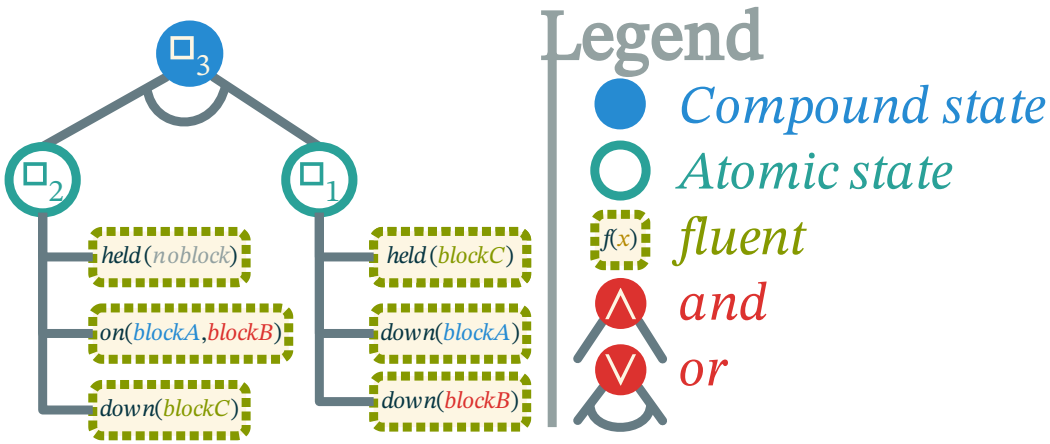


Figure 4.3: Example of a state encoded as an and/or tree.

4.2.1.3 Verification and binding constraints

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

Classical planning has a clear distinction between preconditions and effects. Preconditions are predicates that need to be satisfied before the action can be executed. In classical formalism, effects are separated into positive and negative effects (Ghallab *et al.* 2004). Most of the time, planners will only support simple positive and grounded effects in order to make planning much easier.

In our model we consider preconditions and effects as states. The idea is to make the planning formalism as uniform as possible in order to factor most operations into generic ones.

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

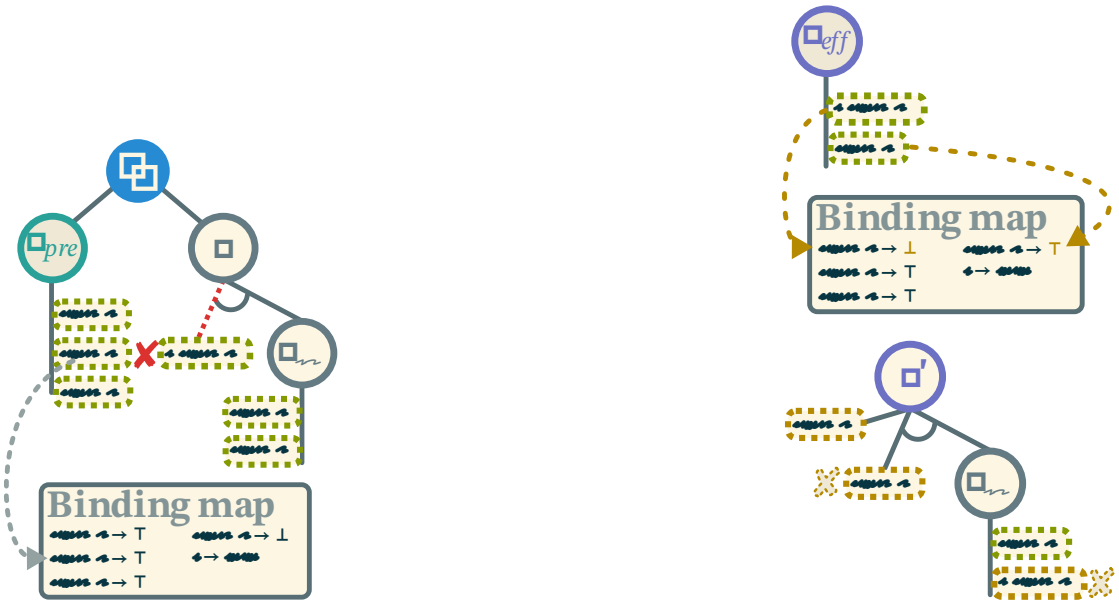
The algorithm is a regular and/or tree exploration and evaluation applied on the state $\square = \square_{pre} \wedge \square$. During the evaluation, if an inconsistency is found then the algorithm returns nothing. Otherwise, at each node of the tree, the algorithm will populate the binding map and verify if the truth value of the node holds under those constraints. All quantified variables are also registered in the binding map to enforce coherence in the root state. If the node is a state, the algorithm recursively applies until it reaches fluents. Once $\square$ is valuated as true, the binding map is returned. Figure 4.4a illustrates this process.

Example 33. Using previously defined example states $\square_{1,2,3}$, and adding the following:

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

We can express a few examples of fluent verification:

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



(a) Verification (b) Application

Figure 4.4: Examples of operations on states.

4.2.2 Effect Application

Once the verification is done, the binding map is kept until the planner needs to apply the action to the state. In our formalism, effects *are* states instead of describing what fluents are added or deleted. The application will therefore be much different than with classical planning. Indeed the goal of the

application will be to enforce the validity of the effect state while maintaining the coherence of the next state.

The application of an effect state is noted $\square_{eff}(\square) = \square'$ and is very similar to the verification. The algorithm will traverse the state $\square$ and use the binding map to force the values inside it. The binding map is previously completed using $\square_{eff}$ to enforce the application of its new value in the current state. This leads to changing the state $\square$ progressively into $\square'$ and the application algorithm will return this state. This process is illustrated in figure 4.4b.

4.2.2.1 Actions

Actions are the main mechanism behind automated planning, they describe what can be done and how it can be done. For clarity's sake, in this document we use the following informal definitions:

- *Action* (or task): Any and all form of function that is closed on the domain of states. Meaning that actions are functions (cf. chapter 2) that take a state and returns another state or $\gg$. Actions are the more general notion of this list.
- *Operator*: Any generalized action that is provided as part of a planning domain. Operators are fully lifted.
- *Plan*: Any action that isn't reducible to a single atomic instanced operator. Often plans are described as sequence of existing actions.
- *Method*: is a plan that is a possible realisation of an action or task. An action can have several possible method to be realized.

Definition 30 (Action)**.** An action is a parameterized tuple $a(args) = \langle pre, eff, \gamma, ¢, d, \mathbb{P}, \mathbb{M} \rangle$ where:

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

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

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

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

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

Example 34. A useful action we can define from previously defined states is the following:

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

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

4.2.2.2 Domain

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

Definition 31 (Domain)**.** A planning domain $\mathcal{D}$ is a set of **operators** which are fully lifted *actions*, along with all the relations and entities needed to describe their preconditions and effects.

Example 35. In the previous examples the domain was named block world. It consists in four actions: $pickup, putdown, stack$ and $unstack$. Usually the domain is self contained, meaning that all fluents, types, constants and operators are contained in it.

4.2.3 Planning Problem & Solution

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

4.2.3.1 Solution to Planning Problems

Definition 32 (Partial Plan / Method)**.** A partially ordered plan is an *acyclic* directed graph $\mathbb{\pi} = (A_{\mathbb{\pi}}, E)$, with:

- $A_{\mathbb{\pi}}$ the set of **steps** of the plan as vertices. A step is an action belonging in the plan. $A_{\mathbb{\pi}}$ must contain an initial step $a^0_{\mathbb{\pi}}$ and goal step $a^*_{\mathbb{\pi}}$ as convenience for certain planning paradigms.

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

With this definition, any kind of plan can be expressed. This includes temporal, fully or partially ordered plans or even hierarchical plans (using the methods of the actions $\mathbb{\Pi}$). It can even express diverse planning results. An exhaustive list of instance is presented in section 4.5.

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

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

Example 36. In the section 4.1, we described a classical fully ordered plan, illustrated in figure 4.2. A partially ordered plan has a tree-like structure except that it also meets in a "sink" vertex (goal step). We explicit this structure in figure 4.5. This figure is an example of how partially ordered plans are structured. The main feature of such a graph is its acyclicity.

4.2.3.2 Planning Problem Definition

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

Definition 33 (Planning 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 for the preconditions and effects, we decided to factorize processes by making the problem homogeneous. Indeed, since the problem itself is an action and that each action also can have methods comprised of actions, it is possible to treat problems and actions the same way (especially useful for hierarchical planning). Most of the specific notions of this framework are optional. The user is free to support any features of the framework according to their use case.

Since we base our planning formalism on the one of SELF, we can express the structure of our formalism using the same extended Venn diagram as in figure 3.6. The figure 4.6 explains how planning uses the previously defined

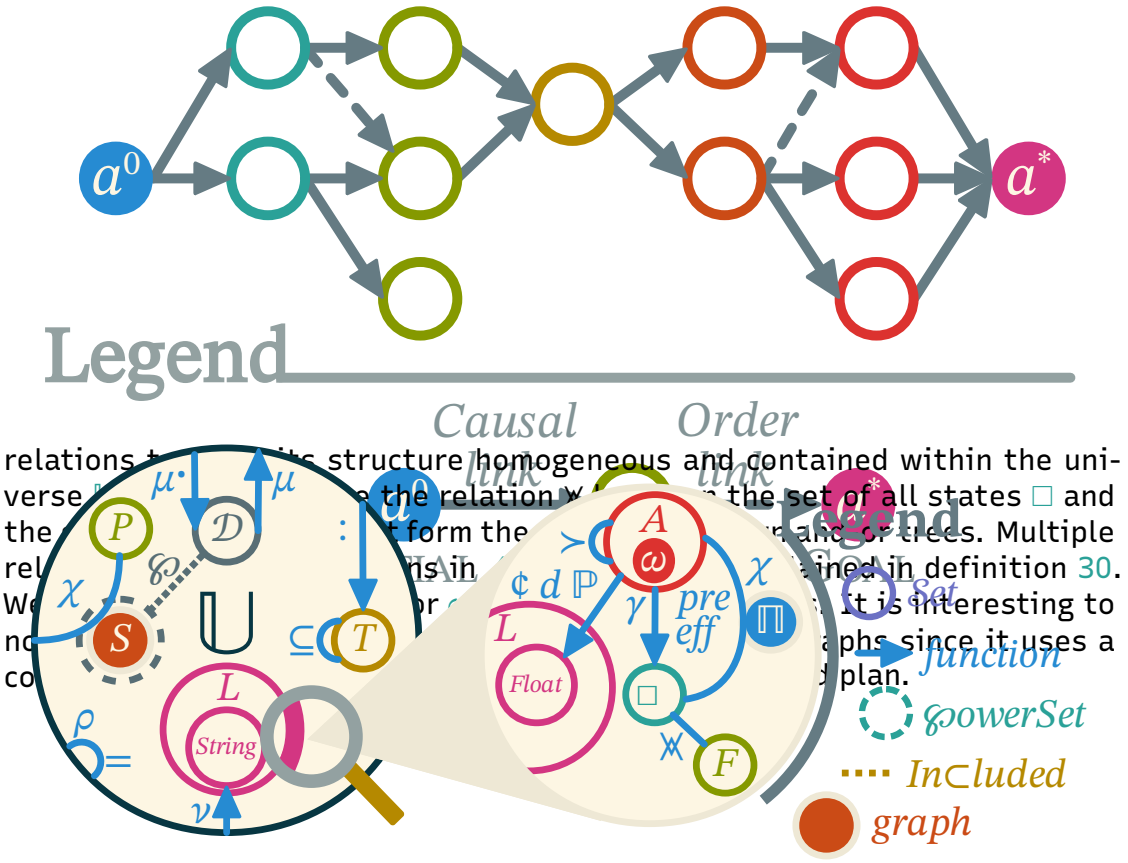


Figure 4.6: Venn diagram extended from the one from SELFs to add all planning knowledge representation.

4.3 Planning search

A general planning algorithm can be described as a guided exploration of a search space. The detailed structure of the search space as well as search iterators are dependent on the planning paradigm used and therefore are parameters of the algorithm. There are also quite a few aspects that need consideration like solution and temporal constraints.

4.3.1 Search space

Definition 34 (Planner)**.** A planning algorithm, often called planner, is an exploration of a search space $\mathbb{S}$ partially ordered by an iterator $\chi_{\mathbb{S}}$ guided by a heuristic h. From any problem $\mathbb{p}$ every planner can derive two pieces of information immediately:

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

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

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

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

In order to accommodate some specific planning paradigm, it is interesting to have a common way to encode search constraints such as the expected number of solutions, or even the allocated time to complete the search.

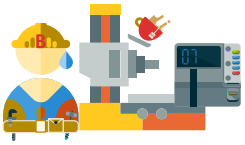
4.3.2 Solution constraints

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

We note $\gamma_{\mathbb{S}}$ the **set of constraints on the solution**. These constraints are represented as states: a set of predicates that evaluates to true once the solution is meeting the expectations.

These constraints can vary widely between planning paradigm. As such, it is interesting to standardize how each planning paradigm will encode extra problem specification and requirements. In order to explain why solution constraints are relevant to planning, we will give it all on a simple example.

Example 37. Bob is a machinist and plans to make a part using a mill. This part is needed for a bigger project and must be ready by a specific deadline.



While taking a tea break, Bob spills over his drink over the mill's electronics making it unusable. Bob originally planned to make the part using computer control. This is a problem because Bob needs an alternative plan now that his original plan failed.

In the following sections we will explain what Bob can do to meet his deadline and how it is relevant for planning.

4.3.2.1 Diversity

A planning paradigm in particular requires explicit constraints on the solution. This paradigm is called *diverse planning*. The goal is to find several plans that are all solutions but all different enough as to be different approaches. It can be taught as a form of hierarchical planning where the composition isn't specified but we expect nonetheless different methods.

In order to quantify the expected diversity of a set of solutions, we introduce to $\gamma_{\mathbb{S}}$ the plan distance metric Δ (Lee 1999) that allows to compare how much two plans are different.

Example 38. Bob could have come up with two different plans to make that part such as using the mill manually or using a subcontractor to make the part.

The main idea behind diverse planning is to come prepared and to present very different plans. For example, Bob couldn't just have used a plan that would have consisted of switching the orientation of the work on the mill or anything too similar to the original plan.

4.3.2.2 Cardinality

In diverse planning, another criteria that is user specified is the cardinality of the set of solutions. We note k the number of expected different solutions. This simply makes the process return when either it found k solutions or when it determined that $k > |\mathbb{S}^*|$.

In classical planning, $k = 1$ since only the best plan is expected. This parameter is added to the set of constraints $\gamma_{\mathbb{S}}$.

Example 39. In our example, Bob clearly came prepared with a unique plan. The replanning he is now doing could have been avoided with more plans prepared in advance.

In practice, finding a good number of plans is quite arbitrary and context dependent and so often left at the user's discretion.

4.3.2.3 Probability

For probabilistic planning, all elements of the probability distributions used are typically included in the domain. The prior probability distribution noted $\mathbb{P}$ is therefore encoded into $\gamma_{\mathbb{S}}$.

Probabilistic planning uses this probability distribution to compute a *policy* that helps reach a goal. A policy is a heuristic guide that returns the action most likely to succeed in any given situation. So a policy is basically a function $\mathbb{pol} = \square \to \{\!|\max \circ \mathbb{P}_{\square} : A|\!\}$ that will always give the action with the maximum success probability knowing the current state.

Using the mapping notation $\{\!|\}$ from chapter 2.

Example 40. Bob can use a policy to solve that issue. In his case it is probably more of a "protocol" or a document instructing of what to do in most situations. That way, Bob doesn't have to improvise and will follow what was decided before.

4.3.2.4 Temporality

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

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

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

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

Example 41. In our running example, Bob has a specified deadline to abide with. This kind of constraints are often implicit and not given to the planner. Classical planners will simply make a best effort to find the best solution and are only evaluated up to a certain time to compare results between approaches.

4.4 General Planning Algorithm

A general planner is an algorithm capable of resolving all types of planning domains and problems. This means that it becomes possible to compare the characteristics of completely different planning approaches. It also makes it possible to use several planning paradigms at once (e.g. durative actions and probabilistic planning combined). The formalization of such a planner allows for a more general explanation and description of planning problems and solutions.

4.4.1 Formalization

We note a general planner $\mathbb{\Pi}^*(\mathbb{s}^*, \mathbb{s}_0, q_{\mathbb{s}^*}, h, \gamma_{\mathbb{S}}, \mathcal{D})$ an algorithm that can find solutions to any planning formalism using the appropriate instance.

The most important point to understand is that a general planner is a shortest path algorithm. These kinds of algorithms are very common in AI and are often used as an example of classical algorithms. One of the first classical shortest path algorithms is called $A*$ (Hart *et al.* 1968). It is a simple heuristic powered shortest path algorithm. It is important to note that any shortest path algorithm can be used to make a general planner. For example, a variant of the $A*$ algorithm can be used for diverse planning. This algorithm is called $K*$ after the term k used to denote the number of expected solutions (Aljazzar and Leue 2011, alg. 1).

In automated planning, this last algorithm is used as the base of some diverse planners (Stentz and others 1995; Riabov *et al.* 2014). Using our formalism, the parameters are as follows: $K^*(g_{\mathbb{S}}, \mathbb{s}_0, q_{\mathbb{s}^*}, h)$. In this case, the solution predicate contains the solution constraints $\gamma_{\mathbb{S}}$.

Of course this is merely an instance of a general planning algorithm. We haven't evaluated the performance of any instance. All we propose in this chapter is the formalism.

Figure 4.7 is an illustration of how such a general planner is structured. The darker inner circle represents the search space $\mathbb{S}$ as defined in definition 34. The gears represent different aspects of the solution constraints $\gamma_{\mathbb{S}}$.

Now that a general planner has been formed it is quite relevant to provide instances of it on every planning paradigms to show that it can fit them all.

4.5 Classical Planning Paradigms

One of the most comprehensive work on summarizing the automated planning domain was done by Ghallab *et al.* (2004). This book explains the different planning paradigm of its time and gives formal description of some of them.

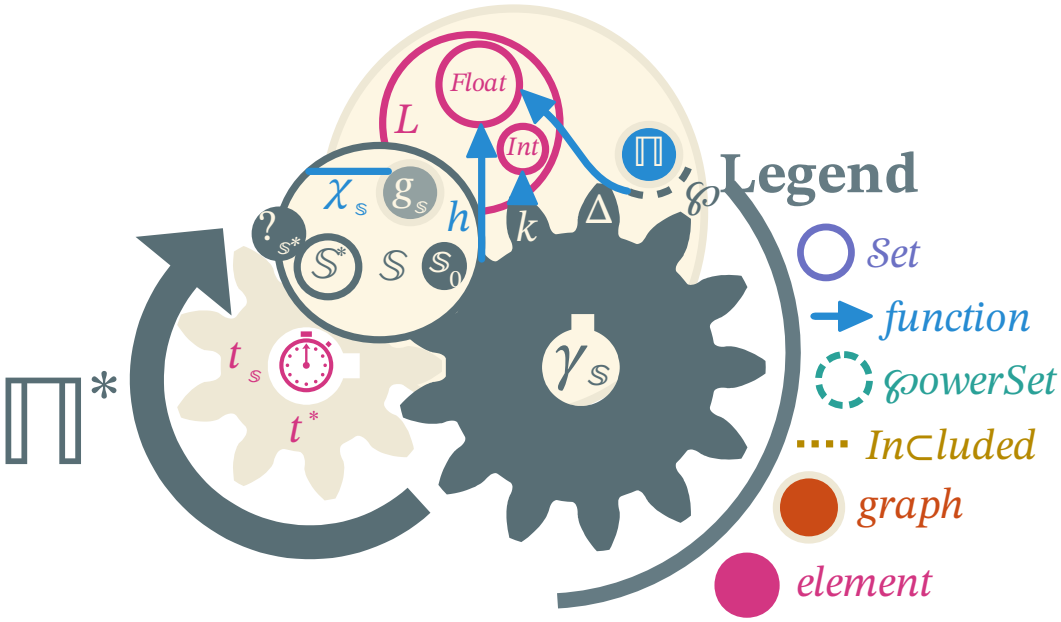


Figure 4.7: Venn diagram extended with our general planning formalism.

This work has been updated later (Ghallab *et al.* 2016) to reflect the changes occurring in the planning community.

In the following sections we will show instances of the general planner for each planning formalism. We will omit the last three parameters h, $\gamma_{\mathbb{S}}$, $\mathcal{D}$. We do so in order to use the partial application of chapter 2 and obtain a specific planner ready to be fully instantiated by the user (the domain is built in a way so that it doesn't depend on the paradigm).

4.5.1 State-transition planning

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

The parameters for state space planning are trivial:

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

This formulation takes advantage of several tools previously described. It uses the partial application of a function to omit the last parameters. It also defines the search graph using the set of all states $\square$ as the vertices

set and the set of all available actions A as the set of edges while considering actions as relations that can be applied to states to make the search progress toward an eventual solution. We also use the binary nature of states to use the effects of the root operator as the solution predicate.

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

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

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

4.5.2 Plan space planning

Plan Space Planning

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

The transformation into a general planner is more complicated than for state-based planning as the progress is made through refinements. We note the set of possible refinements of a given plan $r = \mathbb{\pi} \rightarrow \{\!\{\odot : \otimes(\mathbb{\pi})\}\!\}$ with $\otimes$ being the flaws and $\odot$ the resolvers. Each refinement is a new plan in which we fixed a *flaw* using one of the possible *resolvers* (see definition 35 and definition 36).

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

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

Partial Order Causal Links

Details about flaws, resolvers and the overall POCL algorithm will be presented in section 6.1.1.

This approach can usually give a partial plan if we set $t_{\mathbb{S}}$ too low for the algorithm to complete. This plan is not a solution but can eventually be used as an approximation for certain use cases (like intent recognition, see chapter 7).

4.5.3 Case based planning

Another plan oriented planning is called CBP. This kind of planning relies on a library $\mathcal{C}$ of already complete plans and tries to find the most appropriate one to repair. To repair a plan we use a derivative of the refinement function noted r_* that will make either a classical refinement or repair a part of the plan.

Case-Based Planning

$$\mathbb{\Pi}_{\mathcal{C}} = \mathbb{\Pi}^*\left(r_*, \{\!|\min : \mathbb{\pi} \in \mathcal{C} \wedge \mathbb{\pi}(pre(\omega)) \vDash eff(\omega)|\!\}, \mathbb{s}(pre(\omega)) \neq \varnothing\right)$$

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

4.5.4 Probabilistic planning

Probabilistic planning tries to deal with uncertainty by working on a policy instead of a plan. The initial problem holds probability laws that govern the execution of any actions. It is sometimes accompanied with a reward function instead of a deterministic goal. We use the set of states as search space with the policy as the iterator.

$$\mathbb{\Pi}_{\mathbb{p}} = \mathbb{\Pi}^*\left(\mathbb{p}\mathbb{o}\mathbb{l}, pre(\omega), \mathbb{s} \vDash eff(\omega)\right)$$

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

4.5.5 Hierarchical planning

HTN are a totally different kind of planning paradigm. Instead of a goal description, HTN uses a root task that needs to be decomposed. The task decomposition is an operation that replaces a task (action) by one of its methods $\mathbb{\Pi}$. We note r_+ the set of classical refinements in a plan along with any action decomposition (see chapter 6).

Hierarchical Task Network

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

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

4.6 Discussion on General Planning

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

Hierarchical Task Network with Task Insersion

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

5 COLOR Framework

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

Stanford Research Institute Problem Solver

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

Action Description Language

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.

soUnd Complete Partial Order Planner

5.1 PDDL

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

Artificial Intelligence Planning Systems

With time, the planning competitions became known under the name of IPC. With each installment, the language evolved to address issues encountered the previous years. The current version of PDDL is **3.1** (Kovacs 2011). Its syntax goes similarly as described in listing 5.1.

International Planning Competitions

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

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

Listing 5.1: Simplified explanation of the syntax of PDDL.

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

Example 42. Consider the classic block world domain expressed in listing 5.2. It uses a predicate to express whether a block is on the table because several blocks can be on the table at once. However it uses a 0-ary function to describe the one block allowed to be held at a time. The description of the stack of blocks is done with a unary function to give the block that is on top of another one. To be able to express the absence of blocks it uses a constant named `no-block`. All the actions described are pretty straightforward: `stack` and `unstack` make sure it is possible to add or remove a block before doing it and `pick-up` and `put-down` manages the handling operations.

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

```
33       :effect
34       (and (assign (in-hand) ?x)
35     (assign (on-block ?y) no-block))))
```

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

However, PDDL is far from a universal standard. Some efforts have been made to standardize the domain of automated planning in the form of optional requirements. The latest of the PDDL standard is the version 3.1 (Kovacs 2011). It has 18 atomic requirements as represented in figure 5.1. Most requirements are parts of PDDL that either increase the complexity of planning significantly or that require extra implementation effort to meet. For example, the `quantified-precondition` adds quantifiers into the logical formula of preconditions forcing a check on all fluents of the state to check the validity

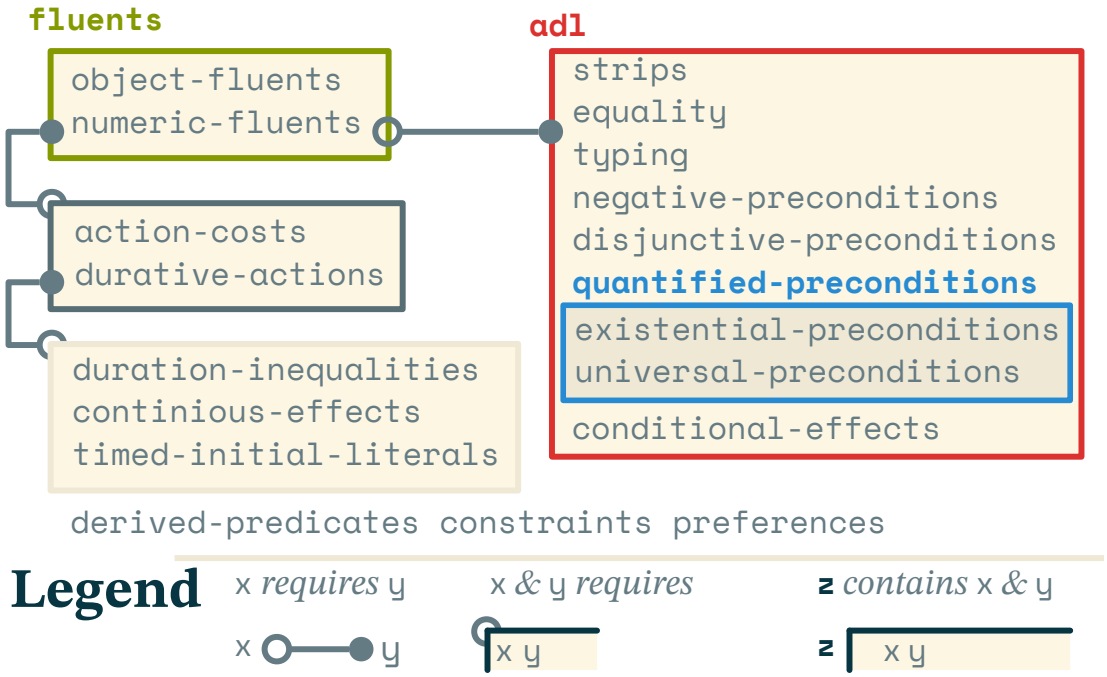


Figure 5.1: Dependencies and grouping of PDDLs requirements.

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

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

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

5.2 Temporality oriented

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

5.2.1 PDDL+

PDDL+ is an extension of PDDL 2.1 that handles processes and events (Fox and Long 2002). It can be viewed as similar to PDDL 3.1 continuous effects but it differs on the expressivity. A process can have an effect on fluents at any time. The effect can happen either from the agent's own doing or being purely environmental. It might be possible in certain cases to model this using the durative actions, continuous effects and timed initial literals of PDDL 2.2 (Edelkamp and Hoffmann 2004) or later versions.

In listing 5.3, we reproduce an example from Fox and Long (2002). It shows the syntax of durative actions in PDDL+. The language uses *timed preconditions* to specify how an action behaves with time and what is needed for its success. It can also specify condition on the duration of the action. The timed preconditions are also available in PDDL 3.1, but the `increase` and `decrease` rate of fluents is an exclusive feature of PDDL+.

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

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

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

5.2.2 ANML

A more recent proposition of a temporal orriented language, is called ANML (Smith *et al.* 2008). This language is used to express complex temporal constraints and dynamical values through time. It does so by using temporal quantifiers applied to classical logical fluents. For example we can write `[start+5, end-2) heater == on ;` to signify that the heater must stay on for that ammount of time. In listing 5.4, we can see the syntax used for durative actions in ANML. This action is a simple travel from a specified location to another while having a fixed duration. While it also does support HTN ploblem descriptions, the support seems to be still limited (To *et al.* 2016).

```
1 action Navigate (location from, to) {
2   duration := 5 ;
3   [all] { arm == stowed ;
4     position == from :-> to ;
5     batterycharge :consumes 2.0 } }
```

Listing 5.4: Example of a simple ANML action.

5.3 Probabilistic

Sometimes, acting can become unpredictable. An action can fail for many reasons, from logical errors down to physical constraints. This calls 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.

The idea behind that probabilistic planning is to account for the probability of success when choosing an action. The resulting 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 is exponentially harder than classical planning. Indeed the planner needs to take into account every outcome of every action in the plan and react accordingly.

5.3.1 PPDDL

PPDDL was used during the 4^{th} and 5^{th} IPC for its probabilistic track (Younes and Littman 2004). It allows for probabilistic effects as demonstrated in listing 5.5. These effects, have an associated probability that denotes the likelihood of them happening. This allows for planners to select actions with adverse effects if they deems the risks worth the damage.

```
1 (define (domain bomb-and-toilet)
2     (:requirements :conditional-effects :probabilistic-effects)
3     (:predicates (bomb-in-package ?pkg) (toilet-clogged)
4                  (bomb-defused))
```

```
    (:action dunk-package
             :parameters (?pkg)
             :effect (and (when (bomb-in-package ?pkg)
                                (bomb-defused))
                          (probabilistic 0.05 (toilet-clogged)))))
```

Listing 5.5: Example of PPDDL use of probabilistic effects from Younes's paper.

5.3.2 RDDL

Another language used by the 7th IPC's uncertainty track is RDDL (Sanner 2010). This language has been chosen because of its ability to express problems that are hard to encode in PDDL or PPDDL. Indeed, RDDL is capable of expressing probabilistic networks 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 of a BNF is provided in listing 5.6 from Sanner (2010). In this listing, we can see that actions in RDDL don't need preconditions or effects. In that case the reward is the closest information to the classical goal and the action is simply a parameter that will influence the probability distribution of the events that conditioned the reward. This distribution is a part of the planning problem and is entirely defined in the listing 5.6.

```
////////////////////////////////////////////////////////////////////////
// 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 5.6: Example of RDDL syntax by Sanner.

5.4 Multi-agent

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

5.4.1 MAPL

MAPL is another extension of PDDL 2.1 that was introduced to handle synchronization of actions (Brenner 2003). This is done using modal operators over fluents. In that regard, MAPL is closer to the PDDL+ extension proposed earlier. It encodes durative actions that will later be integrated into the PDDL 3.0 standard. It also seems to share a similar syntax to PDDL 3.0. MAPL also introduces a synchronization mechanism using speech as a communication vector. This seems very specific as explicit communication isn't a requirement of collaborative work. Listing 5.7 is an example of the syntax of MAPL domains. In that example, a durative action `Move` is used to account for the temporal aspect of connecting travels. This describes the fact that a connection is possible only if it departs after the arrival of the previous connecting travel. This enables agents to plan their travel accordingly to maximize the sharing of ressources and enhance cooperative behaviors.

```
(: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 5.7: Example of MAPL syntax by Brenner.

5.4.2 MA-PDDL

Another aspect of multi-agent planning is the ability to affect tasks and to manage interactions between agents efficiently. For this MA-PDDL seems more adapted than MAPL. It is an extension of PDDL 3.1, that makes easier to plan for a team of heteroneous agents (Kovács 2012). In the example in listing 5.8, 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. It is then possible to treat agents as variables and use all available conditional expression on them to ensure proper multi-agent planning. 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 5.8: Example of MA-PDDL syntax by Kovacs.

5.5 Hierarchical

Another approach to planning is using HTNs to resolve some planning problems. Instead of searching to satisfy a goal, HTNs try to find a decomposition to a root task that fits the initial state requirements and that generates an executable plan.

5.5.1 UMCP

One of the first planners 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 post conditions). The syntax is exposed in listing 5.9. 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 5.9: Example of the syntax used by UCMP.

5.5.2 SHOP2

The next HTN planner is SHOP2 by Nau *et al.* (2003). It remains to this day, the reference implementation of an HTN planner. The SHOP2 formalism is quite similar to UCMP's: each method has a signature, a precondition formula and eventually a decomposition description. This decomposition is a set of methods like in UCMP. The methods can also be partially ordered allowing more expressive plans. An example of the syntax of a method is given in listing 5.10. This Lisp-like language is using a `:method` keyword to define subtasks and their order. However, this only allows totally ordered plans.

```
(: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 5.10: Example of method in the SHOP2 language.

5.5.3 HDDL

A more recent example of HTN formalism comes from the PANDA framework by Bercher *et al.* (2014). This framework is considered the current standard of HTN planning and allows for great flexibility in domain description. It is proposed to be the language for the new hierarchical planning track at the IPC. PANDA takes previous formalisms and generalizes them into a new language exposed in listing 5.11. That language was called HDDL. It uses the same `:method` keyword to define the same kind of totally ordered methods. With this complete example, it seams that defining several methods for a given task isn't possible in this language.

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

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

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

```
    :parameters (?v - vehicle ?l - location ?p - package ?s1 ?s2 -
    capacity-number)
    :task (unload ?v ?l ?p)
    :subtasks (drop ?v ?l ?p ?s1 ?s2)
  )

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

Listing 5.11: Example of HDDL syntax as used in the PANDA framework.

5.5.4 HPDDL

A very recent language proposition was done by Ramoul (2018). He proposes HPDDL with a simple syntax similar to the one of UCMP. In listing 5.12 we give an example of HPDDL method. Its expressive power seems similar to that of UCMP and SHOP. It works in Java using the PDDL4J framework (Pellier and Fiorino 2018).

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

Listing 5.12: Example of HPDDL syntax as described by Ramoul.

5.6 Ontological

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

This kind of approaches are interesting for more than compatibility. Indeed, it becomes then easier to have very expressive planning domains linked to complex ontologies. This also allows to take into account semantic information of the domain description to sign,ificantly improve heuristics using "common sense" logic (Babli *et al.* 2015). The main issue is then of complexity and performance on large domains.

5.6.1 WebPDDL

The first instance of such a system isWebPDDL. This language, shown in listing 5.13, 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 dependent on their parameters. This allows for greater expressivity than regular PDDL, but can be partially emulated using PDDL 3.1 constraints and object fluents.

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

Listing 5.13: Example of WebPDDL syntax by Mc Dermott.

5.6.2 OPT

This previous work was updated by McDermott (2005). The new version is called OPT and allows for some further expressivity. It can express hierarchical domains with links between actions and even advanced data structure. The syntax is mostly an update of WebPDDL. In listing 5.14, we can see that the URI was replaced by simpler names, the action notation was simplified to make the parameter and return value more natural. Axioms were replaced by facts with a different notation.

```
(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 5.14: Example of the updated OPT syntax as described by Mc Dermott.

5.7 Hybrids

Due to the limitations of PDDL, the research on hybrid planner seems to be limited. We can cite the previously discussed ANML for its limited support of HTN problems. It is interesting to note that most hybrid planners are starting as temporally oriented planners. Indeed, PDDL support on this particular area is quite limited and complex durative actions and fluents are only

available as of PDDL 3. This shows the link between language and standard availability and ease of planner conception.

In this section we will not discuss simple hybrid planners that combine two separate planners into one like Duet (Gerevini *et al.* 2008) because the hybridation is limited to a call to the other planner when encountering a different paradigm.

5.7.1 SIADEX

One of the few hybrid planner is called SIADEX (Castillo *et al.* 2006). Like ANML, it combines HTN and temporal orriented planning. The main difference is that SIADEX is an HTN planner that also does temporal constraints and ANML is the opposite. As it is mainly an HTN planner, SIADEX supports partially ordered methods. It is based on STNs to solve for temporal constraints while doing the hierarchical decomposition process. In listing 5.15, we illustrate an example of a composite action `travel-to` with two methods `Fly` and `Drive`. A durative primitive action is also shown with complex temporal constraints. The interesting aspect is the ability to express composite action with such complex temporalities.

Simple Temporal Networks

```
(:task travel-to
 :parameters (?destination)
 (:method Fly
  :precondition (flight ?destination)
  :tasks ((go-to-an-airport)
          (take-a-flight-to ?destination)))
 (:method Drive
  :precondition (not (flight ?destination))
  :tasks ((take-my-car)
          (drive-to ?destination))))

(:durative-action drive-to
 :parameters(?destination)
 :duration (= ?duration
           (/ (distance ?current ?destination)
              (average-speed my-car)))
 :condition(and (current-position ?current)
                (available my-car))
 :effect(and (current-position ?destination)
             (not (current-position ?current))))
```

Listing 5.15: Example of SIADEX composite and durative actions.

5.8 Color and general planning representation

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

5.8.1 Framework

In order to describe this planning framework into SELF, we simply put all fields of the actions into properties. Entities are used as fluents, and the entire knowledge domain as constraints.

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

Listing 5.16: Definition of default planning notions in <+self>

The file presented in listing 5.16, gives the definition of the syntax of fluents and actions in SELF. The first line includes the default syntax file using the first statement syntax. The fluents are simply typed as entities. This allows them to be either parameterized entities or statements. States are either a set of fluents or a logical statement between states or fluents. When a state is represented as a set, it represents the conjunction of all fluents in the set.

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

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

It is interesting to note that some of the most popular feature on action languages such as object types hierarchies and instanciation of actions and planning domain are native to SELF and therefore already included by default. So if one wants to make a planning domain and then to create a specific planning problem from it, it is as simple as simply either programatically append the problem file to the existing knowledge database or to simply use the include notation `"domain.w" = ?;` to extend the domain definition with new ones.

5.8.2 Example domain

Using the classical example domain used earlier, we can write the following file in listing 5.17.

```
1 "planning.w" = ? ; //include base terminology
2
3 (! on !, held(!), down(_)) :: Fluent;
4
5 pickUp(x) pre (~ on x, down(x), held(~));
6 pickUp(x) eff (~(down(x)), held(~));
7
8 putDown(x) pre (held(x));
9 putDown(x) eff (held(~), down(x));
10
11 stack(x, y) pre (held(x), ~ on y);
12 stack(x, y) eff (held(~), x on y);
13
14 unstack(x, y) pre (held(~), x on y);
15 unstack(x, y) eff (held(x), ~ on y);
```

Listing 5.17: Blockworld written in <+self> to work with Color

At line line 1, we need to include the file defined in listing 5.16. After that, line 3 defines the allowed arrity of each relation/function used by fluents. This restricts the cardinality eventually between parameters (one to many, many to one, etc.).

Line 5 encodes the action $pickup$ defined earlier. It is interesting to note that instead of using a constant to denote the absence of the block, we can use an anonymous exclusive quantifier to make sure no block is held. This is quite useful to make concise domains that stay expressive and intuitive.

5.8.3 Differences with PDDL

SELF+Color is more concise than PDDL. It will infer most types and declarations. Variables are also inferred if they are used more than once in a statement while also used as parameters.

While PDDL uses a fixed set of extensions to specify the capabilities of the domain, SELF uses inclusion of other files to allow for greater flexibility. In PDDL, everything must be declared while in SELF, type inference allows for usage without definition. It is interesting to note that the use of variable names `x` and `y` are arbitrary and can be changed for each statement and the domain will still be functionally the same. The line 3 in listing 5.2 is a specific feature of SELF that is absent in PDDL. It is possible to specify constraints on the cardinality of properties. This limits the number of different combinations of values that can be true at once. This is typically done in PDDL using several predicates or constraints.

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

problem, it cannot take advantages of clues that lies in the name and parameters of an action (Babli *et al.* 2015).

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

Also another difference between SELF and PDDL is the underlying planning framework. We presented the one of SELF (listing 5.16) but PDDL seems to suppose a more classical state based formalism. For example, the fluents are of two kinds depending on if they are used as preconditions or effects. In the first case, the fluent is a formula that is evaluated like a predicate to know if the action can be executed in any given state. Effects are formulas enforcing the values of existing fluent in the state. SELF just supposes that the new knowledge is enforcing and that the fluents are of the same kind since verification about the coherence of the actions is made prior to its application in planning.

5.9 Conclusion

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

An interesting perspective on the subject of Color is to use that planning formalism for multi-agent planning. Indeed, the ability to merge and extend arbitrary part of a planning domain makes it very suitable for distributed planning as well as for cooperation and negociation. We can imagine an agent expressing its concern for the cost of a plan or the value of a variable used in an instance using the same language as the planning domain is expressed and interpreted in. We will explore a subset of the multi-agent aspect of the language with intent recognition in chapter 7.

6 Online and Flexible Planning Algorithms

The planning process works using distinct execution phases. In figure 6.1, we illustrate the components of the process.

Figure 6.1: Planning phases for online planning

The first phase has been explained in chapter 5, it transforms the input domain description into a machine ready form. This allows for easy manipulation of the planning entities by the planning algorithm. The second phase is the initialization. It starts pre-processing the planning domain and problem so that the planning phase gets significantly faster. Adding code to this part is a tradeoff between overhead and planning performance. Only the planning phase is meant to have real-time constraints on its execution the rest is usually a linear process and can be negligible in terms of execution times. After a result is found, it is processed or further refined (if time constraints allow for it) and returned to the user in a readable form.

In this chapter, we present planners and approaches to inverted planning and intent recognition. To do that we must first have an efficient online planning algorithm that can take into account observed plans or fluents and find the most likely plan to be pursued by an external agent. The planning process must be done in real time and take into account new observations to make new predictions. This requires the use of online planners.

Classical planning can be used for such a work but lacks flexibility when needing to re-plan at high frequency. The planner must be either able to reuse previously found plans or be able to compute quickly plans that are good approximation of the intended goal. Further discussions of inverted planning and intent recognition can be found in chapter 7.

In order to make an efficient online planner, we chose to explore more expressive and flexible approaches to use the semantics of the planning domain to attempt to guide the search to a more sensical plan. This approach uses either repair heuristics or explanations to provide fast predictions of the intended goals.

First we will discuss existing planning algorithms of the sort. Next we propose and evaluate our first planner that uses plan repair instead of

re-planning. And finally, we present a hierarchical planner that is able to produce intermediary abstract plans at any time of its resolution.

6.1 Existing Flexible Planning Algorithms

In order to make a planner capable of repairing plans, the most fitting paradigm is PSP as described in section 4.5.2. Using the plan space for search allows to modify the refinement process into repairing existing plans.

The second approach using explanations is hierarchical. The planner will use a HTN planning domain that contains composite actions (or tasks) that have several methods (as plans) to realize them.

First, PSP will be presented in more details regarding its classical formulation and definition.

6.1.1 Plan Space Planning

All PSP algorithms work in a similar way: their search space is the set of all plans and their iteration operation is plan refinement. This means that every PSP planner searches for *flaws* in the current plan and then computes a set of *resolvers* that potentially fix each of them. The algorithm usually starts with an empty plan only having the initial and goal steps and recursively refine the plan until all flaws have been solved.

Fast Forward

In general, PSP is faster than naive classical planning. However, with the advent of efficient state based heuristics used in FF (Hoffmann 2001) and LAMA (Richter and Westphal 2010), plan space planning has been left behind regarding raw performance. While PSP delays commitment and therefore can make very efficient choices that can be faster than classical planning, most formulations of PSP problems lead to significant increase in complexity (Tan and Gruninger 2014). The backtracking in PSP algorithms along with heavy data structures such as plans to modify at each iteration makes the approach slower by design without an excellent heuristic.

Works on PSP didn't stop at that point (Nguyen and Kambhampati 2001) since it has unique advantages over classical planning. Indeed, by using backward chaining, PSP algorithms are sound and complete and therefore guarantee to find a solution if it exists (Sjöberg and Nissar 2015).

As PSP finds partially ordered plans, it is also by nature more flexible. Indeed, multiple totally ordered plans are contained within a partially ordered one, they are called linearizations. So, when wanting to have several plans with low diversity, PSP is the way to go.

6.1.1.1 Definitions

In section 4.5.2 we have formalized how PSP works in the general planning formalism. However, this formalism is not used by the rest of the community. This means that we still need to define the classical POCL algorithm. In order to define this algorithm, we need to explain the notions of flaws and resolvers.

Definition 35 (Flaws)**.** Flaws are constraints violations within a plan. The set of flaws in a plan π is noted $\bigotimes_{\pi}$. There are different kinds of flaws in classical PSP and additional ones can be defined depending on the application.

Classical flaws often have a few common features. They are **constructive** since they all require an *addition* of causal links and steps in a plan to be fixed. They have a *proper fluent* f that is the cause of the violation in the plan the flaw is representing and a *needer* a_n that is the action requiring the proper fluent to be fulfilled. In classical PSP flaws are either:

- **Subgoals**, also called *open condition* that are yet to be supported by a *provider* a_p. We note subgoals $\otimes^{\ddagger}_{a_n}(f)$.
- **Threats** are caused by steps that can break a causal link with their effects. They are called *breakers* of the threatened link. A step a_b threatens a causal link $l_t = a_p \xrightarrow{f} a_n$ if and only if $(eff(a_b) \not\models f) \wedge (a_b \not\succ a_p \wedge a_n \not\succ a_b)$. Said otherwise, the breaker can potentially cancel an effect of a providing step a_p, before it gets used by its needer a_n. We note threats $\otimes^{\dagger}_{a_n}(f, a_b)$.

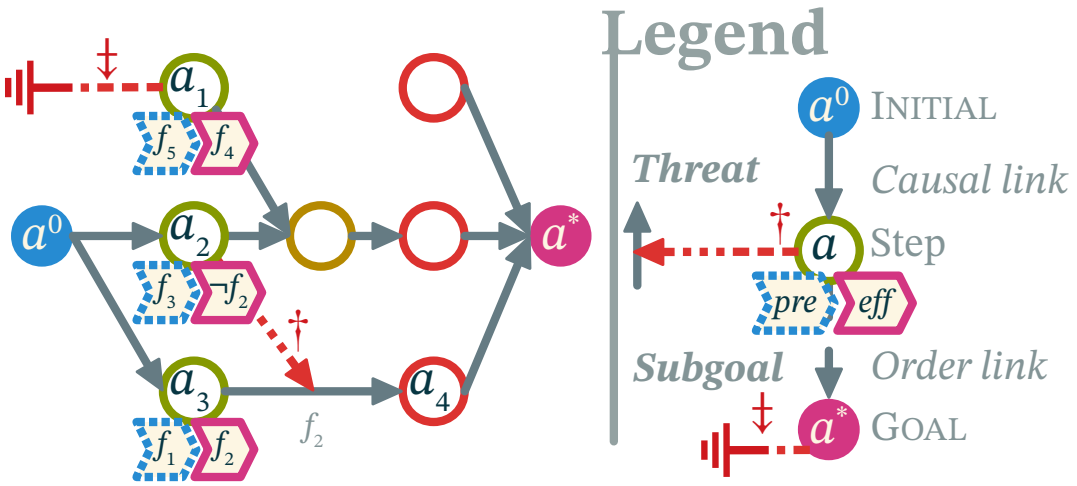


Figure 6.2: Example of partial plan having flaws

Example 43. In figure 6.2 we present a partially ordered plan with two typical flaws. The first is a subgoal missing from the plan to fulfill the f_5 precondition of a_1.

The second flaw in the figure 6.2 is the threat between a_2 and a causal link outgoing from a_3. This happens because nothing prevents a_2 to be executed after a_3 and negate the fluent f_2 needed by the next step.

These flaws need to be fixed in order for the plan to be valid. In POCL it is done by finding their resolvers.

Definition 36 (Resolvers)**.** A resolver is a plan refinement that attempts to solve a flaw $\otimes_{a_n}$. Since classical flaws are constructive, the classical resolvers are called *positive*. They are defined as follows:

- *For subgoals*, the resolvers are a potential causal link containing the proper fluent f of a given subgoal in their causes while taking the needer step a_n as their target and a **provider** step a_p as their source. They are noted $\odot^{+}_{a_p}(\otimes^{\ddagger}_{a_n}(f)) = a_p \xrightarrow{f} a_n$.
- *For threats*, we usually consider only two resolvers: **demotion** ($a_b \succ a_p$) and **promotion** ($a_n \succ a_b$) of the breaker relative to the threatened link. We call the added causeless causal link a **guarding** link. The resolvers for threats are noted $\odot^{+}_{\succ}(\otimes^{\dagger}_{a_n}(f, a_b)) = a_p \rightarrow a_b$ for promotion and $\odot^{+}_{\prec}(\otimes^{\dagger}_{a_n}(f, a_b)) = a_b \rightarrow a_n$ for demotion.

It is possible to introduce extra resolvers to fix custom flaws. In such a case we call positive resolvers, those which add causal links and steps to the plan and negative those that removes causal links and steps. It is preferable to engineer flaws and resolver not to mix positive and negative aspect at once because of the complicated side effects that might result from it.

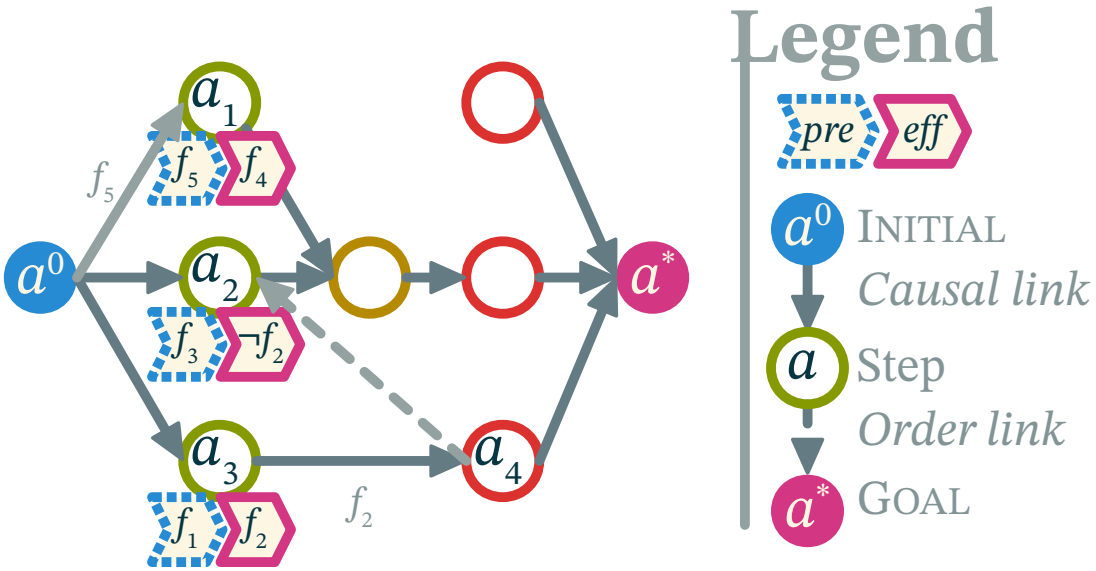


Figure 6.3: Example of resolvers that fixes the previously illustrated flaws

Example 44. From our previous example, we present the complete plan in figure 6.3. The subgoal needs to be fixed by inserting another causal link to

provide the missing fluent and inserting any necessary steps to do so. In that case the initial state happens to provide the necessary fluent so we simply add a causal link for it.

For the threat, the solution is to either promote or demote a_2 so that it doesn't interfere with the causal link between a_3 and a_4. We chose here to demote a_2 so it requires a_4 to be executed before it.

The application of a resolver does not necessarily mean progress. It can have consequences that may require reverting its application in order to respect the backtracking of the POCL algorithm.

Definition 37 (Side effects)**.** Flaws that are caused by the application of a resolver are called *related flaws*. They are inserted into the *agenda* with each application of a resolver:

An agenda is a flaw container used for the flaw selection of POCL.

- *Related subgoals* are all the new open conditions inserted by new steps.
- *Related threats* are the causal links threatened by the insertion of a new step or the deletion of a guarding link.

Flaws can also become irrelevant when a resolver is applied. It is always the case for the targeted flaw, but this can also affect other flaws. Those *invalidated flaws* are removed from the agenda upon detection:

- *Invalidated subgoals* are subgoals satisfied by the new causal links or the removal of their needer.
- *Invalidated threats* happen when the breaker no longer threatens the causal link because the order guards the threatened causal link or either of them have been removed.

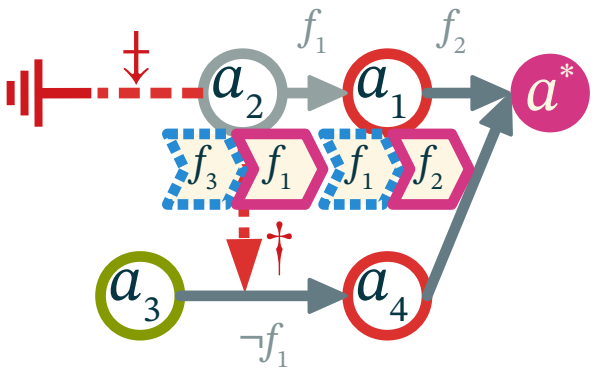


Figure 6.4: Example of the side effects of the application of a resolver

Example 45. In our example, by adding the step a_2 to fix an unsupported subgoal needed by a_1, we introduced another subgoal to support the new step that also threatens the causal link between a_3 and a_4.

Algorithm 5 POCL Algorithm

```
1:  function POCL(Agenda 𝒜, Action ω)
2:      if 𝒜 = ∅ then                                  ▷ Populated agenda needs to be provided
3:          return Success                              ▷ Stops all recursion
4:      Flaw ⊗ ← {|𝒜|}                                 ▷ Heuristically chosen flaw
5:      Resolvers ⊙ ← solve(⊗, {|𝕄(ω)|})               ▷ The root operator has only one method for PSP
6:      for all ⊙ ∈ ⊙ do                               ▷ Non-deterministic choice operator
7:          apply(⊙, 𝕥)                                ▷ Apply resolver to partial plan
8:          Agenda 𝒜′ ← update(𝒜)
9:          if POCL(𝒜′, ω) = Success then              ▷ Refining recursively
10:             return Success
11:         revert(𝒜, 𝕥)                               ▷ Failure, undo resolver application
12:     𝒜 ← 𝒜 ∪ {⊗}                                    ▷ Flaw was not resolved
13:     return Failure                                 ▷ Revert to last non-deterministic choice
```

6.1.1.2 Classical POCL Algorithm

In algorithm 5 we present a generic version of POCL inspired by Ghallab *et al.* (2004, sec. 5.4.2).

For our version of POCL we follow a refinement procedure that works in several generic steps. In figure 6.5 we detail the resolution of a subgoal as done in the algorithm 5.

The first is the search for resolvers. It is often done in two separate steps: first, select the candidates and then check each of them for validity. This is done using the polymorphic function `solve` at line 5.

In the case of subgoals, variable unification is performed to ensure the compatibility of the resolvers. Since this step is time-consuming, the operator is instantiated accordingly at this step to factor the computational effort. Composite operators have also all their methods instantiated at this step if they are selected as a candidate.

Then a resolver is picked non-deterministically for applications (this can be heuristically driven). At line 7 the resolver is effectively applied to the current plan. All side effects and invalidations are handled during the update of the agenda at line 8. If a problem occurs, line 11 backtracks and tries other resolvers. If no resolver fits the flaw, the algorithm backtracks to previous resolver choices to explore all the possible plans and ensure completeness.

6.1.1.3 Existing PSP Planners

Related works already tried to explore new ideas to make PSP an attractive alternative to regular state-based planners like the appropriately named "Reviving partial order planning" (Nguyen and Kambhampati 2001) and VHPOP (Younes and Simmons 2003). More recent efforts (Coles *et al.* 2011; Sapena *et al.* 2014) adapted the powerful heuristics from state-based planning to

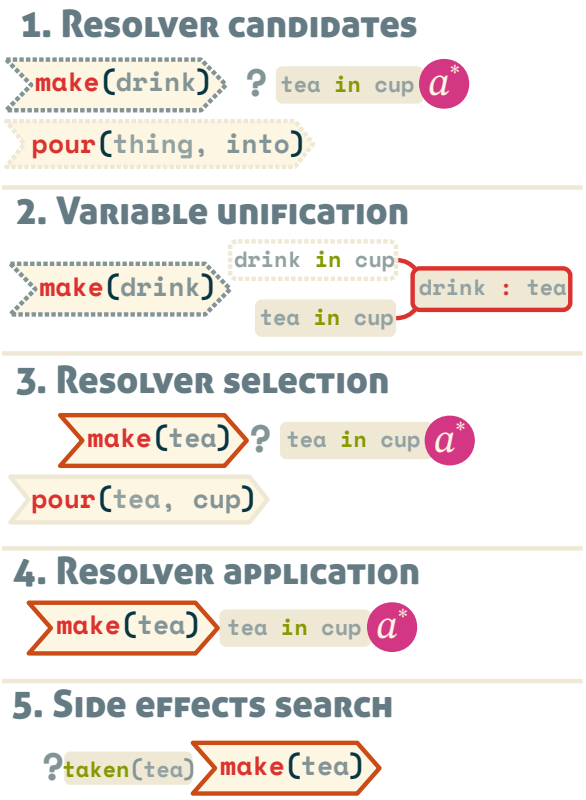


Figure 6.5: Refinement process of POCLs as used in HEARTs

PSP's approach. An interesting approach of these last efforts is found in (Shekhar and Khemani 2016) with meta-heuristics based on offline training on the domain. Yet, we clearly note that only a few papers lay the emphasis upon plan quality using PSP (Ambite and Knoblock 1997; Say *et al.* 2016).

6.1.2 Plan Repair & Reuse

In online planning, the plan is computed frequently from a changing initial state. This means that the previous plan is very often available. In order to take advantage of the effort invested in previous plans, it is tempting to simply reuse the existing plan instead of replanning from scratch. Most work on the field focus on monitoring execution and finding ways to make resilient plans.

In such a case, the planning models can be subject to uncertainty. Indeed, the execution of an action can fail because an external event changed a pre-condition required to do it or because the model itself is inaccurate.

The idea of reusing plan emerged early on (Nebel and Koehler 1995) but with a caveat: often repairing needed more effort than replanning. So plan repair became more of a gamble and needed incentives to reuse an existing plan given an application. For example, such process is useful for multi-agent planning where a significant change of plan is expansive among agents (Ephrati and Rosenschein 1993; Alami *et al.* 1995; Sugawara 1995; Borrajo 2013; Luis and Borrajo 2014). This motivation for plan repair comes from plan merging. Applications of plan merging ranges from cooperation problems to plan optimization.

The question of the efficiency of replanning vs. repairing has been since studied extensively under several aspects (Van Der Krogt and De Weerdt 2005; Fox *et al.* 2006). The emergence of diverse planning and requirement on plan stability of execution monitoring gave new research on the subject. At this point, most of the literature focuses on a case-based planning, where a plan library is already provided and the planner must select a plan and repair it to fit a given case (Gerevini *et al.* 2013; Borrajo *et al.* 2015).

A recent work of Zhuo and Kambhampati (2017) gives an interesting approach to the problem by questioning the domain. Indeed, the need to re-plan can be an opportunity to revise issues in the current model and to improve it by adding newly found solution to complete the given planning model.

In our case, we focus on PSP and how to make plan repair efficiently using that technique. Classical PSP algorithms don't take as an input an existing plan but can be enhanced to fit plan to repair, as for instance in (Van Der Krogt and De Weerdt 2005). Usually, PSP algorithms take a problem as an input and use a loop or a recursive function to refine the plan into a solution. We can't solely use the refining recursive function to be able to use our existing partial plan. This causes multiple side effects if the input plan is suboptimal. This problem was already explored in LGP-adapt (Borrajo 2013). This work explains how reusing a partial plan often implies replanning parts of the plan.

6.1.3 Hierarchical Task Networks

> "HTN planners differ from classical planners in what they plan for and how they plan for it. In an HTN planner, the objective is not to achieve a set of goals but instead to perform some set of tasks."

Ghallab *et al.* (2004)

Planning using HTN gives a completely different approach to the problem and its formulation. In this formalism, actions are composite tasks and there is no goal other than to complete the root task. One can find similarities with our general planning formalism and it isn't a coincidence. Indeed, HTN is more general than planning and therefore one needs to be able to allow for this level of expressivity.

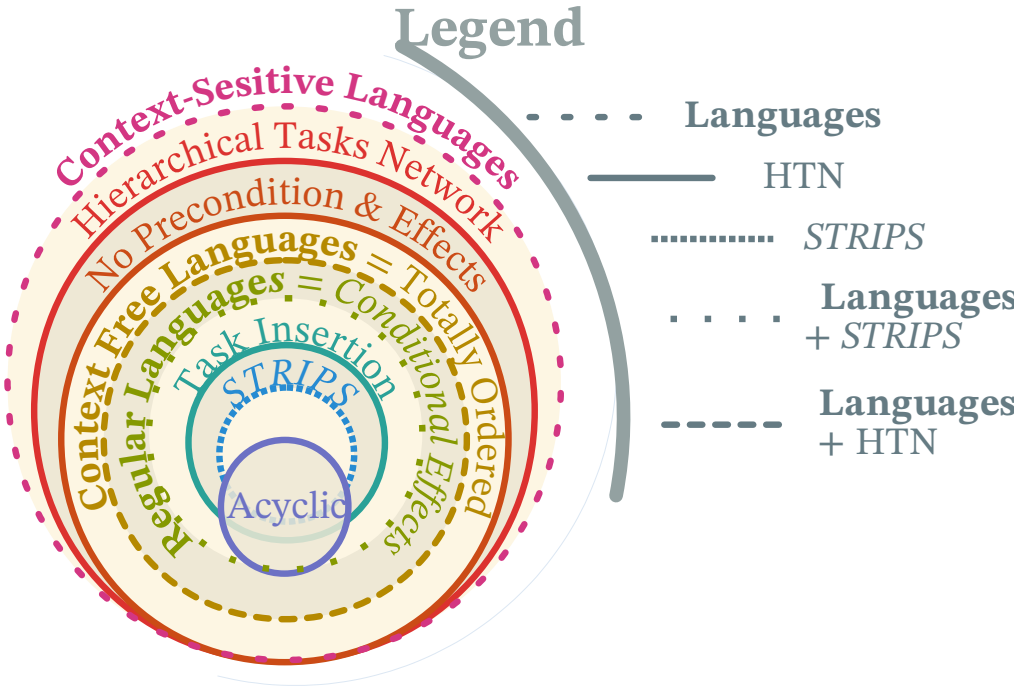


Figure 6.6: Venn diagram of the expressivity classes of HTNs paradigms.

Figure 6.6 shows how the domain expressivity are classed in relation to one another. The most expressive formalism can encode context-sensitive languages. HTN problems are less expressive than these kind of languages. Restricting the expressivity, we can remove precondition and effects of tasks and even enforce total order. That last restriction reduces the expressivity to the level of context-free languages. We can see that STRIPS is about the smallest subset of problems in regard to expressivity. This means that, while it is possible to transform *some* HTN problems into STRIPS like classical planning problem, for most of them this is impossible to do without losing expressivity.

Generated by the CFGs of Chomsky

This expressivity comes at a cost. HTN problems are on a complexity category that is significantly harder than regular STRIPS planning:

From Bercher and Höller (2018) HTN Tutorial at ICAPS 2018

Table 6.1: HTN models expressivity associated with their respective complexity classes

Restrictions	*Complexity*
Classical	P-SPACE
Task Insertion	NEXP-TIME
Totally Ordered	EXP-TIME
Acyclic	NEXP-TIME
Tail-recursive	EXP-SPACE

Table 6.1 gives the complexity of each kind of HTN problem restrictions. Classical HTN problems require P-SPACE algorithms to solve. Adding task insertion the complexity becomes NEXP-TIME and so on.

HTN is often combined with classical approaches since it allows for a more natural expression of domains making expert knowledge easier to encode. These kinds of planners are named **decompositional planners** when no initial plan is provided (Fox 1997). Most of the time the integration of HTN simply consists in calling another algorithm when introducing a composite operator during the planning process. The Duet planner by Gerevini *et al.* (2008) does so by calling an instance of an HTN planner based on the task insertion called SHOP2 (Nau *et al.* 2003) to decompose composite actions. Some planners take the integration further by making the decomposition of composite actions into a special step in their refinement process. Such works include the discourse generation oriented DPOCL (Young and Moore 1994) and the work of Kambhampati *et al.* (1998) generalizing the practice for decompositional planners.

In our case, we chose a class of hierarchical planners based on PSP algorithms (Bechon *et al.* 2014; Dvorak *et al.* 2014; Bercher *et al.* 2014) as a reference approach. The main difference here is that the decomposition is integrated into the classical POCL algorithm by only adding new types of flaws. This allows keeping all the flexibility and properties of POCL while adding the expressivity and abstraction capabilities of HTN.

6.2 LOLLIPOP

aLternative Optimization with partiaL pLan Injection Partial Ordered Planner

LOLLIPOP is a planning algorithm made to test the feasibility of plan repair using PSP techniques. The repairing is done through the addition of special *negative* flaws and resolver to the classical POCL algorithm. This causes some issues specific to that kind of resolvers that POCL is not equiped to solve. Additionally, the plan to repair may have a few different types of inconsistencies that can void the valiodity of the resulting plan or affect negatively the performances of the repair.

In this section, we explore this technique and its related issues and advantages. We also start by explaining how existing relaxation based heuristics can be adapted into a plan repair problem.

6.2.1 Operator Graph

Operator graphs, are a depedancy graph of all the operators in the domain with edges annotated with the fluent that can become a potential causal link.

Definition 38 (Operator Graph)**.** An operator graph g_O of a set of operators O is a labeled directed graph that binds two operators with the causal link $o_1 \xrightarrow{f} o_2$ if and only if there exists at least one fluent so that $(f \in \mathit{eff}(o_1)) \wedge f \vDash \mathit{pre}(o_2)$.

This definition was inspired by the notion of domain causal graph as explained in (Göbelbecker *et al.* 2010) and originally used as a heuristic in (Helmert *et al.* 2011). Causal graphs have fluents as their nodes and operators as their edges. Operator graphs are the opposite: they are an *operator dependency graph* for a set of actions. A similar structure was used in (Peot and Smith 1994) that builds the operator dependency graph of goals and uses precondition nodes instead of labels.

6.2.1.1 Building the graph

While building the operator graph, we need a **providing map** that indicates, for each fluent, the list of operators that can provide it. This is a simpler version of the causal graph that is reduced to an associative table easier to maintain. The list of providers can be sorted to drive resolver selection (as detailed in section 6.2.3). We note $g_{\mathcal{D}}$ the operator graph built with the set of operators in the domain $\mathcal{D}$.

Example 46. In the figure 6.8, we illustrate the application of this mechanism on our example from figure 6.7. Continuous lines correspond to the *domain operator graph* computed during domain compilation time.

The generation of the operator graph is detailed in algorithm 6. It explores the operator space and builds a providing and a needing map that gives the provided and needed fluents for each operator. Once done it iterates on every precondition and searches for a satisfying cause to add the causal links to the operator graph.

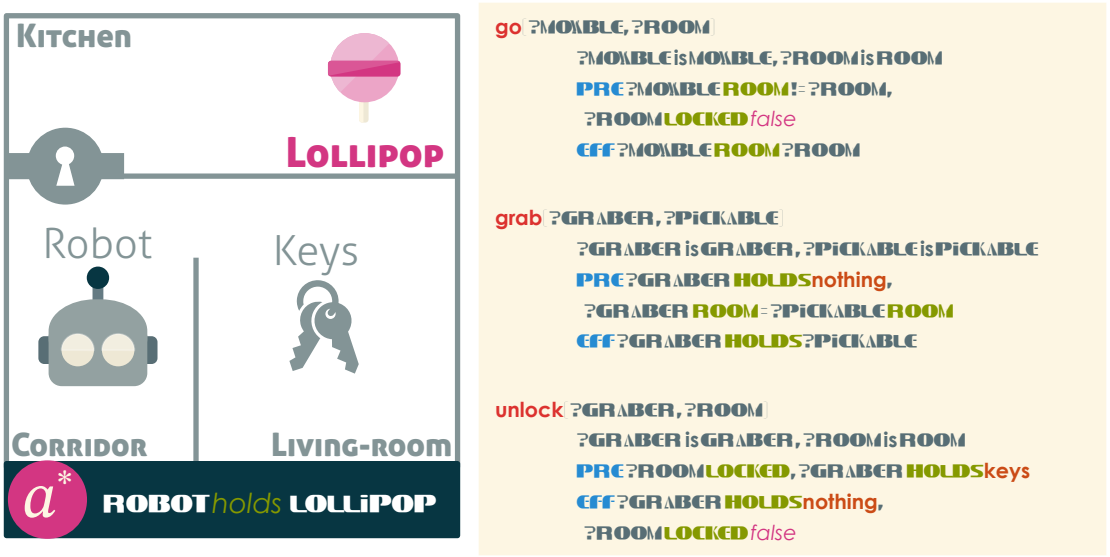


Figure 6.7: Example domain and problem featuring a robot that aims to fetch a lollipop in a locked kitchen. The operator `go` is used for movable objects (such as the robot) to move to another room. The `grab` operator is used by grabbers to hold objects and the `unlock` operator is used to open a door when the robot holds the key.

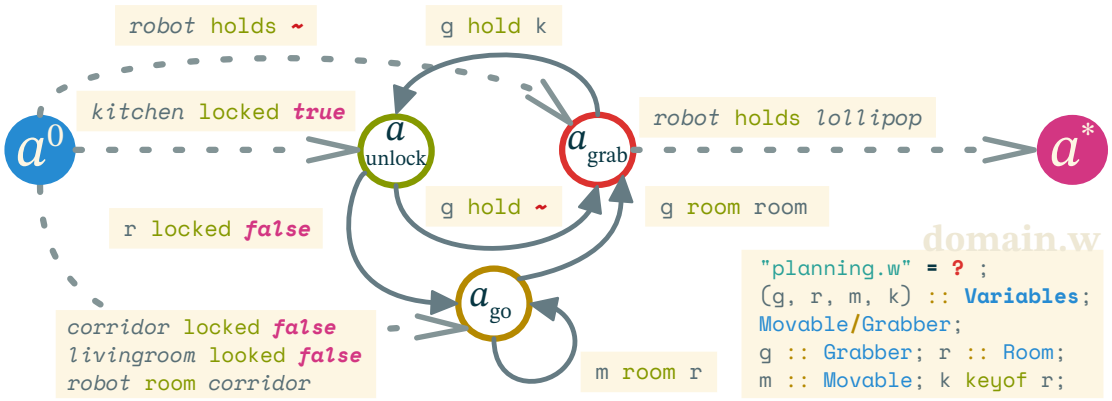


Figure 6.8: Diagram of the operator graph of example domain. Full arrows represent the domain operator graph and dotted arrows the dependencies added to inject the initial and goal steps.

Algorithm 6 Operator graph generation and update algorithm

```
1:  function addVertex(Action a)
2:      cache(a)                          ▷ Update of the providing and needing map
3:      if binding then                   ▷ boolean that indicates if the binding was requested
4:          bind(a)
5:  function cache(Action a)
6:      for all f ∈ eff(a) do             ▷ Adds a to the list of providers of f
7:          add(A_p, f, a)
8:      ...                               ▷ Same operation with needing actions and preconditions
9:  function bind(Action a)
10:     for all f ∈ pre(a) do
11:         if f ∈ A_p then
12:             for all 𝕥 ∈ get(A_p, f) do
13:                 Link l ← getEdge(𝕥, a)        ▷ Create the link if needed
14:                 addCause(l, f)                ▷ Add the fluent as a cause
15:     ...                               ▷ Same operation with needing and effects
```

6.2.1.2 Plan extraction from heuristic indexes

Using heuristics or pre-computed indexes is common in classical planning. A good portion of the research into PSP has been intended toward making the paradigm as efficient than classical planning when using the new generation of heuristics from FD and LAMA (Richter and Westphal 2010). An example of this research is called VHPOP (Younes and Simmons 2003) that uses a kind of meta-heuristic by combining several heuristic approaches to allow for efficient flaw selection in POCL.

Fast Downward

In our case, the approach is completely different. We propose the use of data structures usually used for heuristics as input data of a plan repair algorithm. The idea is to extract a partial plan that acts as scafolding for the planning algorithm to build a valid plan.

To apply the notion of operator graphs to planning problems, we just need to add the initial and goal steps to the operator graph. In figure 6.8, we depict this insertion with our previous example using dotted lines. However, since operator graphs may have cycles, they can't be used directly as input to POCL algorithms to ease the initial back chaining. Moreover, the process of refining an operator graph into a usable one could be more computationally expensive than POCL itself.

In order to give a head start to the LOLLIPOP algorithm, we propose to build operator graphs differently with the algorithm detailed in algorithm 7. A similar notion was already presented as "basic plans" in (Sebastia *et al.* 2000). These "basic" partial plans use a more complete but slower solution for the generation that ensures that each selected steps are *necessary* for the solution. In our case, we built a simpler solution that can solve some basic planning problems but that also makes early assumptions (since our algorithm can handle them). It does a simple and fast backward construction of a

partial plan driven by the providing map. Therefore, it can be tweaked with the powerful heuristics of state search planning.

Algorithm 7 Safe operator graph generation algorithm

```
1:  function safe(Action ω)
2:      Stack<Action> open ← [a*]
3:      Stack<Action> closed ← ∅
4:      while open ≠ ∅ do
5:          Action a ← pocl(open)                    ▷ Remove a from open
6:          push(closed, a)
7:          for all f ∈ pre(a) do
8:              Actions A_p ← getProviding(𝕋, f)     ▷ Sorted by usefulness
9:              if A_p = ∅ then
10:                 S ← S \ {𝕋}
11:                 continue
12:             Action a' ← getFirst(𝕋)
13:             if a' ∈ closed then
14:                 continue
15:             if a' ∉ S then
16:                 push(open, a')
17:             S ← S ∪ {a'}
18:             Link l ← getEdge(a', a)              ▷ Create the link if needed
19:             addCause(l, f)                       ▷ Add the fluent as a cause
```

This algorithm is useful since it is specifically used on goals. The result is a valid partial plan that can be used as input to POCL algorithms.

6.2.2 Negative Refinements

The classical POCL algorithm works upon a principle of positive plan refinements. The two standard flaws (subgoals and threats) are fixed by *adding* steps, causal links, or variable binding constraints to the partial plan. Online planning needs to be able to *remove* parts of the plan that are not necessary for the solution. Since we assume that the input partial plan is quite complete, we need to define new flaws to optimize and fix this plan. These flaws are called *negative* as their resolvers apply subtractive refinements on partial plans.

Definition 39 (Alternative)**.** An alternative $\otimes^{\curvearrowright}$ is a negative flaw that occurs when there is a better provider choice for a given link. An alternative to a causal link $a_p \xrightarrow{f} a_n$ is a provider a_b that has a better *utility value* than a_p.

The **utility value** of an operator is a measure of usefulness being the base of our ranking mechanism detailed in section 6.2.3. It uses the incoming and outgoing degrees of the operator in the domain operator graph to measure its usefulness.

Finding an alternative to an operator is computationally expensive. It requires searching a better provider for every fluent needed by a step. To simplify that search, we select only the best provider for a given fluent and check if the one used is the same. If not, we add the alternative as a flaw. This search is done only on updated steps for online planning. Indeed, the safe operator graph mechanism is guaranteed to only choose the best provider (algorithm 7 at line 12). Furthermore, subgoals won't introduce new fixable alternatives as they are guaranteed to select the best possible provider.

Definition 40 (Orphan)**.** An orphan $\otimes^{\dashrightarrow}$ is a negative flaw that occurs when a step in the partial plan (other than the initial or goal step) is not participating in the plan. Formally a_o is an orphan if and only if $a_o \neq a^0 \wedge a_o \neq a^* \wedge (|\chi_{\prec}(a_o)| = 0) \vee \{\!|= (\varnothing) : \chi_{\prec}(a_o)|\!\}$.

With $\chi_{\prec}(a_o)$ being the set of *outgoing causal links* of a_o in π. This last condition checks for *dangling orphans* that are linked to the goal with only bare causal links (introduced by threat resolution).

The solution to an alternative is a negative refinement that simply removes the targeted causal link. This causes a new subgoal as a side effect, which will focus on its resolver by its rank (explained in section 6.2.3) and then pick the first provider (the most useful one). The resolver for orphans is the negative refinement that is meant to remove a step and its incoming causal link while tagging its providers as potential orphans.

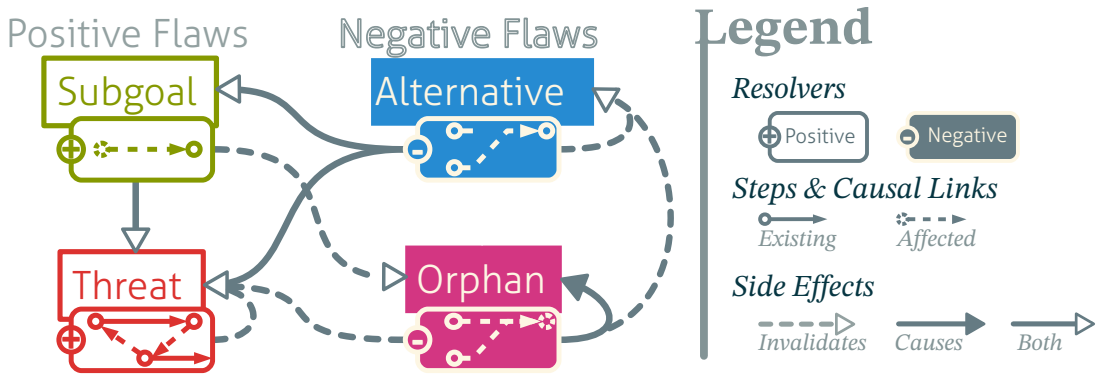


Figure 6.9: Schema representing flaws with their signs, resolvers and side effects relative to each other

The side effects mechanism also needs an upgrade since the new kinds of flaws can interfere with one another. This is why we extend the side effect definition (cf. definition 37) with a notion of sign.

Definition 41 (Signed Side Effects)**.** A signed side effect is either a regular *causal side effect* or an *invalidating side effect*. The sign of a side effect indicates if the related flaw needs to be added or removed from the agenda.

The figure 6.9 exposes the extended notion of signed resolvers and side effects. When treating positive resolvers, nothing needs to change from the classical method. When dealing with negative resolvers, we need to search

for extra subgoals and threats. Deletion of causal links and steps can cause orphan flaws that need to be identified for removal.

In the method described in (Peot and Smith 1993), a **invalidating side effect** is explained under the name of *DEnd* strategy. In classical POCL, it has been noticed that threats can disappear in some cases if subgoals or other threats were applied before them. For our mechanisms, we decide to gather under this notion every side effect that removes the need to consider a flaw. For example, orphans can be invalidated if a subgoal selects the considered step. Alternatives can remove the need to compute further subgoal of an orphan step as orphans simply remove the need to fix any flaws that concern the selected step.

These interactions between flaws are decisive for the validity and efficiency of the whole model, that is why we aim to drive flaw selection in a rigorous manner.

6.2.3 Usefulness Heuristic

Resolvers and flaws selection are the keys to improving performances. Choosing a good resolver helps to reduce the branching factor that accounts for most of the time spent on running POCL algorithms (Kambhampati 1994). Flaw selection is also important for efficiency, especially when considering negative flaws which can conflict with other flaws.

Conflicts between flaws occur when two flaws of opposite sign target the same element of the partial plan. This can happen, for example, if an orphan flaw needs to remove a step needed by a subgoal or when a threat resolver tries to add a promoting link against an alternative. The use of side effects will prevent most of these occurrences in the agenda but a base ordering will increase the general efficiency of the algorithm.

Based on the figure 6.9, we define a base ordering of flaws by type. This order takes into account the number of flaw types affected by causal side effects.

1. **Alternatives** will cut causal links that have a better provider. It is necessary to identify them early since they will add at least another subgoal to be fixed as a related flaw.
2. **Subgoals** are the flaws that cause most of the branching factor in POCL algorithms. This is why we need to make sure that all open conditions are fixed before proceeding on finer refinements.
3. **Orphans** remove unneeded branches of the plan. Yet, these branches can be found out to be necessary for the plan to meet a subgoal. Since a branch can contain many actions, it is preferable to leave the orphan in the plan until they are no longer needed. Also, threats involving orphans are invalidated if the orphan is resolved first.

4. **Threats** occur quite often in the computation. Searching and solving them is computationally expensive since the resolvers need to check if there are no paths that fix the flaw already. Many threats are generated without the need of resolver application (Peot and Smith 1993). That is why we rank all related subgoals and orphans before threats because they can add causal links or remove threatening actions that will fix the threat.

The resolvers need to be ordered as well, especially for the subgoal flaws. Ordering resolvers for a subgoal is the same operation as choosing a provider. Therefore, the problem becomes "how to rank operators?" Usually, each operator has an assigned cost in the domain, but more often than not, costs are hard to estimate manually. In our case we need an automated way to rank operators. The most relevant information on an operator is how useful it may be to other actions in the plan and how hard is it to realize.

Since this may be computationally expensive to compute while planning, the evaluation of the cost of an operator is done offline using the operator graph.

The first metric to compute this heuristic is the degree of the operator.

Definition 42 (Degree of an operator)**.** Degrees are a measurement of the usefulness of an operator. Such a notion is derived from the incoming and outgoing degrees of a node in the operator graph.

We note $|\chi_{\prec\cdot}(a)|$ being the *outgoing degree* of a in the directed graph formed by π and $|\chi_{\succ\cdot}(a)|$ being the *incoming degree* of a in the directed graph formed by π respectively the outgoing and incoming degrees of an operator in a plan π. These represent the number of causal links that goes out or toward the operator. We call proper degree of an operator $|\mathit{eff}(a)|$ and $|\mathit{pre}(a)|$ the number of preconditions and effects that reflect its intrinsic usefulness.

There are several ways to use the degrees as indicators. The *utility value* increases with every outgoing degree, since this reflects a positive participation in the plan. It decreases with every negative degree since actions with higher incoming degrees are harder to satisfy. The utility value bounds are useful when selecting special operators. For example, a user-specified constraint could be laid upon an operator to ensure it is only selected as a last resort. This operator will be affected with the smallest utility value possible. More commonly, the highest value is used for initial and goal steps to ensure their selection.

Our ranking mechanism is based on scores noted $₹(a)$. A score is a tuple of metrics:

- $₹_1(a) = |\chi_{\prec\cdot}(a)|$ is the positive degree of a in the domain operator graph. This will give a measurement of the predicted usefulness of the operator.

In this section χ is the connectivity of the operator graph g_O.

- ₹$_2(a) = |\otimes_a^{\ddagger}|$ is the number of open conditions of a in the domain operator graph. This is symptomatic of action that can't be satisfied without a compliant initial step.
- ₹$_3(a) = |pre(a)|$ is the proper negative degree of a. Having more preconditions will likely add subgoals.
- ₹$_4(a) = \{\!|\min_{+\infty}(n) : n \in \mathbb{N} \wedge (a \rightarrow a) \in \chi^n \wedge \chi^n \neq \chi^+|\!\}$ is the size of the shortest cycle involving a in the operator graph or $+\infty$ if there is none. Having this value at 1 is usually symptomatic of a *toxic operator* (cf. definition 44). Having an operator behaving this way can lead to backtracking because of operator instantiation.

The computation of the cost of the operator is done by multiplying the score tuple with a weighted parameter tuple α given by the user. The cost is then:

$$¢(a) = -\sum_{i=1}^{4} \alpha_i ₹_i(a)$$

In practice, α_1 is positive, and the rest is negative. It is also better to make sure that $-1 \leq \alpha_4 \leq 0$ so that the penalties goes down as the cycles gets bigger.

This respects the criteria of having a bound for the *utility value* as it ensures that it remains positive with 0 as a minimum bound and $+\infty$ for a maximum. The initial and goal steps have their utility values set to the upper bound to ensure their selection over other steps.

Choosing to compute the resolver selection at operator level has some positive consequences on the performances. Indeed, this computation is much lighter than approaches with heuristics on plan space (Shekhar and Khemani 2016) as it reduces the overhead caused by real time computation of heuristics on complex data. In order to reduce this overhead more, the algorithm sorts the providing associative array to easily retrieve the best operator for each fluent. This means that the evaluation of the heuristic is done only once for each operator. This reduces the overhead and allows for faster results on smaller plans.

6.2.4 Algorithm

The LOLLIPOP algorithm uses the same refinement algorithm as described in algorithm 5. The differences reside in the changes made on the behavior of resolvers and side effects. In line 7 of algorithm 5, the LOLLIPOP algorithm applies negative resolvers if the selected flaw is negative. Another change resides in the initialization of the solving mechanism and the domain as detailed in algorithm 8. This algorithm contains several parts. First, the `domainInit` function corresponds to the code computed during the domain compilation time. It will prepare the rankings, the operator graph, and its

caching mechanisms. It will also use strongly connected component detection algorithms to detect cycles. These cycles are used during the base score computation (line 11). We add a detection of illegal fluents and operators in our domain initialization (line 5). Illegal operators are either inconsistent or toxic.

Definition 43 (Inconsistent operators)**.** An operator a is contradictory if and only if $(pre(a) = \bot) \vee (eff(a) = \bot)$.

Definition 44 (Toxic operators)**.** Toxic operators have effects that are already in their preconditions or empty effects. An operator a is toxic if and only if $pre(a) \cap eff(a) \neq \emptyset \vee eff(a) = \emptyset$.

Toxic actions can damage a plan as well as make the execution of POCL algorithms longer than necessary. This is fixed by removing the toxic fluents ($pre(a) \nsubseteq eff(a)$) and by updating the effects with $eff(a) = eff(a) \setminus pre(a)$. If the effects become empty, the operator is removed from the domain.

The lollipopInit function is executed during the initialization of the solving algorithm. We start by realizing the scores, then we add the initial and goal steps in the providing map by caching them. Once the ranking mechanism is ready, we sort the providing map. With the ordered providing map, the algorithm runs the fast generation of the safe operator graph for the problem's goal.

The last part of this initialization (line 21) is the agenda population that is detailed in the populate function. During this step, we perform a search of alternatives based on the list of updated fluents. Online updates can make the plan outdated relative to the domain. This forms liar links :

Definition 45 (Liar links)**.** A liar link is a link that doesn't hold a fluent in the preconditions or effect of its source and target. We note:

$$a_i \xrightarrow{f} a_j | f \notin eff(a_i) \cap pre(a_j)$$

A liar link can be created by the removal of an effect or preconditions during online updates (with the causal link still remaining).

We call lies the fluents that are held by links without being in the connected operators. To resolve the problem, we remove all lies. We delete the link altogether if it doesn't bear any fluent as a result of this operation. This removal triggers the addition of orphan flaws as side effects.

While the list of updated operators is crucial for solving online planning problems, a complementary mechanism is used to ensure that LOLLIPOP is complete. User-provided plans have their steps tagged. If the failure has backtracked to a user-provided step, then it is removed and replaced by subgoals that represent each of its participation in the plan. This mechanism loop until every user provided steps have been removed.

Algorithm 8 LOLLIPOP initialization (preprocessing) mechanisms

```
1: function domainInit(Actions A)
2:     operatorgraph g_O
3:     Score ₹
4:     for all Action a ∈ A do
5:         if isIllegal(a) then                          ▷ Remove toxic and useless fluents
6:             A ← A \ {a}                               ▷ If entirely toxic or useless
7:             continue
8:         addVertex(a, g_O)                             ▷ Add and bind all operators
9:         p ← cache(a)                                  ▷ Cache operator in providing map
10:    Cycles C ← stronglyConnectedComponent(g_O)        ▷ Using DFS
11:    □ ← baseScores(A, 𝒟^𝕄)
12:    i ← inapplicables(𝒟^𝕄)
13:    e ← eagers(𝒟^𝕄)
14: function lollipopInit(Problem ℙ, Providing p, Reward ₹)
15:    ¢ ← realize(₹, ℙ)                                 ▷ Compute the scores
16:    cache(p, a^0)                                     ▷ Cache initial step in providing ...
17:    cache(p, a^*)                                     ▷ ... as well as goal step
18:    p ← sort(p, ¢)                                    ▷ Sort the providing map
19:    if 𝕄(ω) = ∅ then
20:        𝕄(ω) ← {safe(ℙ)}          ▷ Computing the safe operator graph if the plan is empty
21:    populate(𝒜, ℙ)                                    ▷ populate agenda with first flaws
22: function populate(Agenda 𝒜, Problem ℙ)
23:    for all Update u ∈ U do                           ▷ Updates due to online planning
24:        Fluents F ← eff(u_new) \ eff(u_old)            ▷ Added effects
25:        for all Fluent f ∈ F do
26:            for all Operator a ∈ better(p, f, a) do
27:                for all Link l ∈ χ⁺(a) do
28:                    if f ∈ l then
29:                        addAlternative(a, f, a, l_←, ℙ)   ▷ With l_← the target of l
30:        F ← eff(u_old) \ eff(u_new)                    ▷ Removed effects
31:        for all Fluent f ∈ F do
32:            for all Link l ∈ χ⁺(u_new) do
33:                if isLiar(l) then
34:                    L ← L \ {l}
35:                    addOrphans(a, u, ℙ)
36:        ...                  ▷ Same with removed preconditions and incoming liar links
37:    for all Operator a ∈ A_𝕄(ω) do
38:        addSubgoals(a, ℙ)
39:        addThreats(a, ℙ)
```

6.2.5 Theoretical and Empirical Results

As proven in (Penberthy *et al.* 1992), the classical POCL algorithm is *sound* and *complete*.

First, we define some new properties of partial plans. The following properties are taken from the original proof. We present them again for convenience.

Definition 46 (Full Support)**.** A partial plan $\mathbb{\pi}$ is fully supported if each of its steps $a \in A_{\mathbb{\pi}}$ is fully supported. A step is fully supported if each of its preconditions $f \in pre(a)$ is supported. A precondition is fully supported if there exists a causal link l that provides it. We note:

$$\{\!|\wedge \circ \vDash \circ \chi : A_{\mathbb{\pi}}|\!\} \wedge (\otimes^{\dagger}(\mathbb{\pi}) = \varnothing)$$

with $\{\!|\}\!|$ being the mapping function, χ being the connectivity function of the graph formed by the plan $\mathbb{\pi}$ and $A_{\mathbb{\pi}}$ being the set of all actions in the plan.

Definition 47 (Partial Plan Validity)**.** A partial plan is a **valid solution** of a problem $\mathbb{p}$ if it is *fully supported* and *contains no cycles*. The validity of $\mathbb{\pi}$ regarding a problem $\mathbb{p}$ is noted $\mathbb{\pi} \vDash (\mathbb{p} \equiv \Downarrow \mathbb{\pi} \wedge lBrace =: \chi^{+}_{\mathbb{\pi}}|\!\} = \varnothing)$.

6.2.5.1 Proof of Soundness

In order to prove that this property applies to LOLLIPOP, we need to introduce some hypothesis:

- operators updated by online planning are known.
- user provided steps are known.
- user provided plans don't contain illegal artifacts. This includes toxic or inconsistent actions, lying links and cycles.

Based on the definition 47 we state that:

$$\forall f \in pre(\omega) : \Downarrow f \wedge \{\!|=: \chi^{+}_{\mathbb{\pi}}|\!\} = \varnothing \implies \mathbb{\pi} \vDash \mathbb{p} \tag{6.1}$$

where ω is the root operator with $pre(\omega) = a^{*}$.

This means that $\mathbb{\pi}$ is a solution if all preconditions of a^{*} are fully supported without cycles in the plan. We can satisfy these preconditions using operators if and only if their preconditions are all satisfied and if there is no other operator that threatens their supporting links.

First, we need to prove that equation 6.1 holds on LOLLIPOP initialization. We use our hypothesis to rule out the case when the input plan is invalid. The algorithm 7 will only solve open conditions in the same way subgoals do it. Thus, safe operator graphs are valid input plans.

Since the soundness is proven for regular refinements and flaw selection, we need to consider the effects of the added mechanisms of LOLLIPOP. The newly introduced refinements are negative, they don't add new links:

$$\forall \otimes \in \bigotimes_{\pi} \forall \odot \in \bigodot_{\pi}(\otimes) : \{\!|=: \chi^{+}_{\pi}|\!\} = \{\!|=: \chi^{+}_{\odot(\pi)}|\!\} \tag{6.2}$$

with $\otimes$ being any flaw in π, $\odot$ being the set of resolvers of said flaw and $\odot(\pi)$ being the resulting partial plan after the application of the resolver. Said otherwise, an iteration of LOLLIPOP won't add cycles inside a partial plan.

The orphan flaw targets steps that have no path to the goal and so can't add new open conditions or threats. The alternative targets existing causal links. Removing a causal link in a plan breaks the full support of the target step. This is why an alternative will always insert a subgoal in the agenda corresponding to the target of the removed causal link. Invalidating side effects also doesn't affect the soundness of the algorithm since the removed flaws are already solved. This makes:

$$\forall \otimes \in \bigotimes^{-}_{\pi} \forall \odot \in \bigodot_{\pi}(\otimes) : \Downarrow \pi \implies \Downarrow \odot(\pi) \tag{6.3}$$

with $\bigotimes^{-}_{\pi}$ being the set of negative flaws in the plan <+π>. This means that negative flaws don't compromise the full support of the plan.

Equation 6.2 lead to equation 6.1 being valid after the execution of LOLLIPOP. The algorithm is sound.

6.2.5.2 Proof of Completeness

The soundness proof shows that LOLLIPOP's refinements don't affect the support of plans in terms of validity. It was proven that POCL is complete. There are several cases to explore to transpose the property to LOLLIPOP:

Lemma 1 (Conservation of Validity)**.** *If the input plan is a valid solution, LOLLIPOP returns a valid solution.*

Proof. With equation 6.2 and the proof of soundness, the conservation of validity is already proven. □

Lemma 2 (Reaching Validity with incomplete partial plans)**.** *If the input plan is incomplete, LOLLIPOP returns a valid solution if it exists.*

Proof. Since POCL is complete and the equation 6.3 proves the conservation of support by LOLLIPOP, then the algorithm will return a valid solution if the provided plan is an incomplete plan and the problem is solvable. □

Lemma 3 (Reaching Validity with empty partial plans)**.** *If the input plan is empty and the problem is solvable, LOLLIPOP returns a valid solution.*

Proof. This is proven using lemma 2 and POCL's completeness. However, we want to add a trivial case to the proof: $pre(\omega) = \emptyset$. In this case the algorithm 5 will return a valid plan with only the root operator.

□

Lemma 4 (Reaching Validity with a dead-end partial plan)**.** *If the input plan is in a dead-end, LOLLIPOP returns a valid solution.*

Proof. Using input plans that can be in an undetermined state is not covered by the original proof. The problem lies in the existing steps in the input plan. Still, using our hypothesis we add a failure mechanism that makes LOLLIPOP complete. On failure, the needer of the last flaw is deleted if it wasn't added by LOLLIPOP. User-defined steps are deleted until the input plan acts like an empty plan. Each deletion will cause corresponding subgoals to be added to the agenda. In this case, the backtracking is preserved and all possibilities are explored as in POCL. □

Since all cases are covered, this proves the property of completeness.

6.2.5.3 Experimental Results

The experimental results focused on the properties of LOLLIPOP for online planning. Since classical POCL is unable to perform online planning, we tested our algorithm considering the time taken for solving the problem for the first time. We profiled the algorithm on a benchmark problem containing each of the possible issues described earlier.

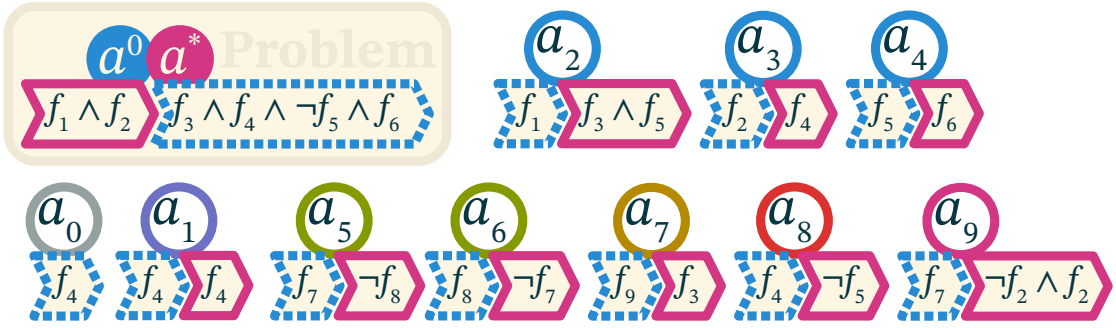


Figure 6.10: Domain used to compute the results. First line is the initial and goal steps along with the useful actions. Second line contains a threatening action a_8, two co-dependent actions a_5 and a_6, a useless action a_0, a toxic action a_1, a dead-end action a_7 and an inconsistent action a_9

In figure 6.10, we expose the planning domain used for the experiments. During the domain initialization, the actions a_0 and a_1 are eliminated from the domain since they serve no purpose in the solving process. The action a_9 is stripped of its negative effect because it is inconsistent with the effect f_2.

As the solving starts, LOLLIPOP computes a safe operator graph (full lines in figure 6.11). As we can see, this partial plan is nearly complete already. When the main refining function starts it receives an agenda with only a few flaws remaining.

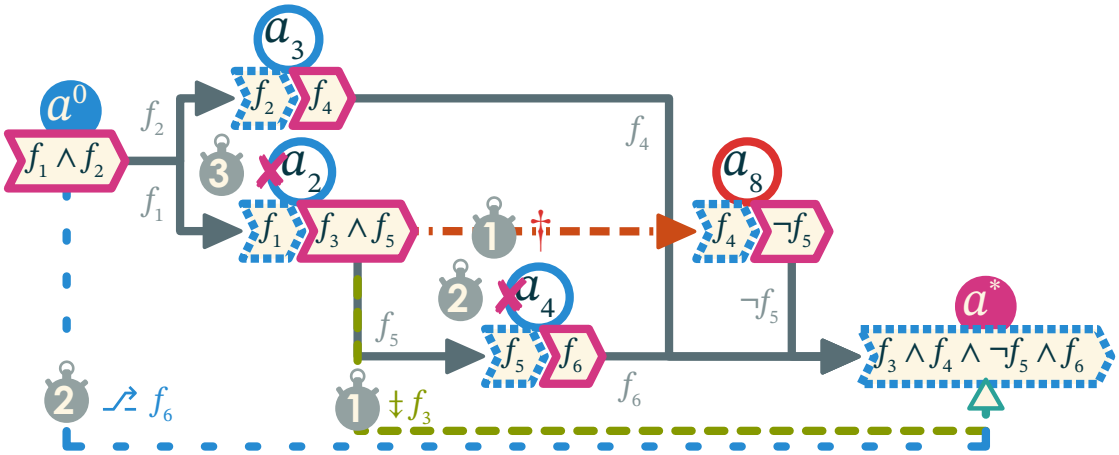


Figure 6.11: In full lines the initial safe operator graph. In thin, sparse and irregularly dotted lines respectively a subgoal, alternative and threat caused causal link.

Then the main refinement function starts (time markers **1**). LOLLIPOP selects as resolver a causal link from a_2 to satisfy the open condition of the goal step. Once the first threat between a_2 and a_8 is resolved the second threat is invalidated. On a second execution, the domain changes for online planning with f_6 added to the initial step. This solving (time markers **2**) adds as flaw an alternative on the link from a_4 to the goal step. A subgoal is added that links the initial and goal step for this fluent. An orphan flaw is also added that removes a_4 from the plan. Another solving takes place as the goal step doesn't need f_3 as a precondition (time markers **3**). This causes the link from a_2 to be cut since it became a liar link. This adds a_2 as an orphan that gets removed from the plan even if it was hanging by the bare link to a_8.

The measurements exposed in table 6.2 were made with an Intel® Core™ i7-4720HQ with a 2.60GHz clock. Only one core was used for the solving. The same experiment done only with the chronometer code gave a result of $70ns$ of errors. We can see an increase of performance in the online runs because of the way they are conducted by LOLLIPOP.

Table 6.2: Average times of 1.000 executions on the problem. The first column is for a simple run on the problem. Second and third columns are times to replan with one and two changes done to the domain for online planning.

Experiment	*Single*	*Online 1*	*Online 2*
Time (*ms*)	0.86937	0.38754	0.48123

6.3 HEART

6.3.1 Domain Compilation

In order to simplify the input of the domain, the causes of the causal links in the methods are optional. If omitted, the causes are inferred by unifying the preconditions and effects with the same mechanism as in the subgoal resolution in our POCL algorithm. Since we want to guarantee the validity of abstract plans, we need to ensure that user provided plans are solvable. We use the following formula to compute the final preconditions and effects of any composite action a:

$$a = \left\langle \bigwedge_{a' \in \chi_{\prec}(a^0_{\pi})}^{\pi \in \Pi(a)} pre(a'), \bigwedge_{a' \in \chi_{\succ}(a^0_{\pi})}^{\pi \in \Pi(a)} eff(a'), \ldots \right\rangle$$

An instance of the classical POCL algorithm is then run on the problem $\mathbb{p}_a = \langle \mathcal{D}, \gamma_{\mathbb{p}}, a \rangle$ to ensure its coherence. The domain compilation fails if POCL cannot be completed. Since our decomposition hierarchy is acyclic ($a \notin A_a$, see definition 49) nested methods cannot contain their parent's action as a step.

6.3.2 Abstraction in POCL

In order to properly introduce the changes made for using HTN domains in POCL, we need to define a few notions.

$$a \hookrightarrow^{\prec}_{\pi} a' = \langle \{\!\{(a \mapsto a') : \chi_{\prec}(a)\}\!\}, A_{\pi} \setminus \{a\} \cup \{a'\} \rangle$$

FIXME All after this line.

Transposition is needed to define decomposition.

Definition 48 (Transposition)**.** In order to transpose the causal links of an action a' with the ones of an existing step a in a plan π, we use the following operation:

$$a \rhd^-_\pi a' = \left\{ \phi^-(l) \xrightarrow{causes(l)} a' : l \in L^-_\pi(a) \right\}$$

It is the same with $a' \xrightarrow{causes(l)} \phi^+(l)$ and L^+ for $a \rhd^+ a'$. This supposes that the respective preconditions and effects of a and a' are equivalent. When not signed, the transposition is generalized: $a \rhd a' = a \rhd^- a' \cup a \rhd^+ a'$.

Example 47. $a \rhd^- a'$ gives all incoming links of a with the a replaced by a'.

Definition 49 (Proper Actions)**.** Proper actions are actions that are "contained" within an entity (either a domain, plan or action). We note this notion $A_a = A_a^{lv(a)}$ for an action a. It can be applied to various concepts:

- For a *domain* or a *problem*, $A_\mathbb{p} = A_\mathcal{D}$.
- For a *plan*, it is $A^0_\pi = S_\pi$.
- For an *action*, it is $A^0_a = \bigcup_{m \in methods(a)} S_m$. Recursively: $A^n_a = \bigcup_{b \in A^0_a} A^{n-1}_b$. For atomic actions, $A_a = \varnothing$.

Example 48. The proper actions of $make(drink)$ are the actions contained within its methods. The set of extended proper actions adds all proper actions of its single proper composite action $infuse(drink, water, cup)$.

Definition 50 (Abstraction Level)**.** This is a measure of the maximum amount of abstraction an entity can express, defined recursively by:

We use Iverson brackets here $[]$, meaning that $[\bot] = 0 \wedge [\top] = 1$.

$$lv(x) = \left(\max_{a \in A_x} (lv(a)) + 1 \right) [A_x \neq \varnothing]$$

Example 49. The abstraction level of any atomic action is 0 while it is 2 for the composite action $make(drink)$.

The most straightforward way to handle abstraction in regular planners is illustrated by Duet (Gerevini *et al.* 2008) by managing hierarchical actions separately from a task insertion planner. We chose to add abstraction in POCL in a manner inspired by the work of Bechon *et al.* (2014) on a planner called HiPOP. The difference between the original HiPOP and our implementation of it is that we focus on the expressivity and the ways flaw selection can be exploited for partial resolution. Our version is lifted at runtime while the original is grounded for optimizations. All mechanisms we have implemented use POCL but with different management of flaws and resolvers. The original algorithm 5 is left untouched.

One of those changes is that resolver selection needs to be altered for subgoals. Indeed, as stated by the authors of HiPOP: the planner must ensure the selection of high-level operators in order to benefit from the hierarchical

aspect of the domain. Otherwise, adding operators only increases the branching factor. Composite actions are not usually meant to stay in a finished plan and must be decomposed into atomic steps from one of their methods.

Definition 51 (Decomposition Flaws)**.** They occur when a partial plan contains a non-atomic step. This step is the neeeder a_n of the flaw. We note its decomposition $a_n\oplus$.

- *Resolvers:* A decomposition flaw is solved with a **decomposition resolver**. The resolver will replace the neeeder with one of its instantiated methods $m \in methods(a_n)$ in the plan $\mathbb{TT}$. This is done by using transposition such that: $a_n\oplus^m_{\mathbb{TT}} = \langle S_m \cup (S_{\mathbb{TT}} \setminus \{a\}), a_n \triangleright^- I_m \cup a_n \triangleright^+ G_m \cup (L_{\mathbb{TT}} \setminus L_{\mathbb{TT}}(a_n))$.
- *Side effects:* A decomposition flaw can be created by the insertion of a composite action in the plan by any resolver and invalidated by its removal:

$$\bigcup_{a_m \in S_m}^{f \in pre(a_m)} \mathbb{TT}' \ddagger_f a_m \bigcup_{a_b \in S_{\mathbb{TT}'}}^{l \in L_{\mathbb{TT}'}} a_b \otimes l \bigcup_{a_c \in S_m}^{lv(a_c) \neq 0} a_c \oplus$$

Example 50. When adding the step $make(tea)$ in the plan to solve the subgoal that needs tea being made, we also introduce a decomposition flaw that will need this composite step replaced by its method using a decomposition resolver. In order to decompose a composite action into a plan, all existing links are transposed to the initial and goal step of the selected method, while the composite action and its links are removed from the plan.

The main differences between HiPOP and HEART in our implementations are the functions of flaw selection and the handling of the results (one plan for HiPOP and a plan per cycle for HEART). In HiPOP, the flaw selection is made by prioritizing the decomposition flaws. Bechon *et al.* (2014) state that it makes the full resolution faster. However, it also loses opportunities to obtain abstract plans in the process.

HiErarchical Abstraction for Real-Time partial order planner

6.3.3 Planning in cycle

The main focus of our work is toward obtaining **abstract plans** which are plans that are completed while still containing composite actions. In order to do that the flaw selection function will enforce cycles in the planning process.

Definition 52 (Cycle)**.** A cycle is a planning phase defined as a triplet $c = \langle lv(c), agenda, \mathbb{TT}_{lv(c)} \rangle$ where: $lv(c)$ is the maximum abstraction level allowed for flaw selection in the $agenda$ of remaining flaws in partial plan $\mathbb{TT}_{lv(c)}$. The resolvers of subgoals are therefore constrained by the following: $a_p \downarrow_f a_n : lv(a_p) \leq lv(c)$.

During a cycle all decomposition flaws are delayed. Once no more flaws other than decomposition flaws are present in the agenda, the current plan is saved

and all remaining decomposition flaws are solved at once before the abstraction level is lowered for the next cycle: $lv(c') = lv(c) - 1$. Each cycle produces a more detailed abstract plan than the one before.

Abstract plans allow the planner to do an approximate form of anytime execution. At any given time the planner is able to return a fully supported plan. Before the first cycle, the plan returned is $\pi_{lv(a_0)}$.

Example 51. In our case using the method of intent recognition of Sohrabi *et al.* Sohrabi *et al.* (2016), we can already use $\pi_{lv(a_0)}$ to find a likely goal explaining an observation (a set of temporally ordered fluents). That can make an early assessment of the probability of each goal of the recognition problem.

For each cycle c, a new plan $\pi_{lv(c)}$ is created as a new method of the root operator a_0. These intermediary plans are not solutions of the problem, nor do they mean that the problem is solvable. In order to find a solution, the HEART planner needs to reach the final cycle c_0 with an abstraction level $lv(c_0) = 0$. However, these plans can be used to derive meaning from the potential solution of the current problem and give a good approximation of the final result before its completion.

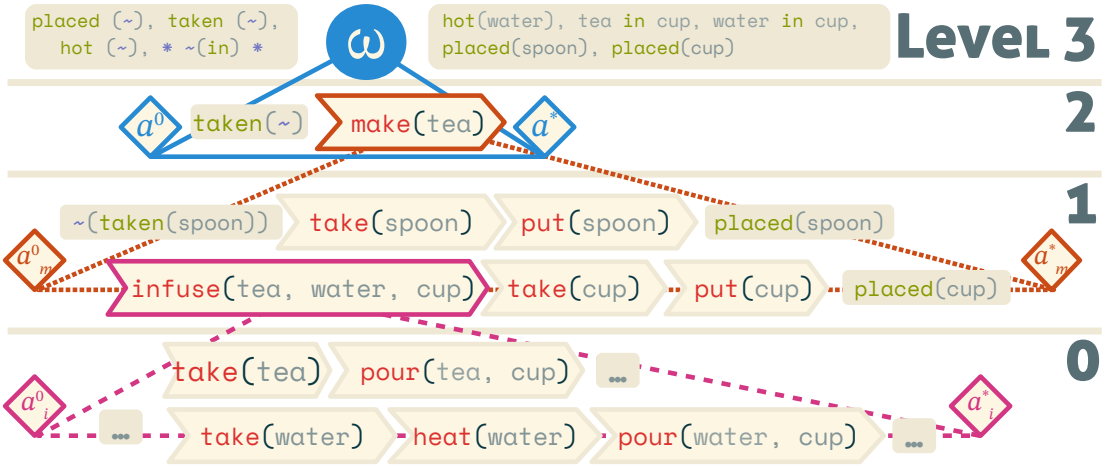


Figure 6.12: Illustration of how the cyclical approach is applied on the example domain. Atomic actions that are copied from a cycle to the next are omitted.

Example 52. In the figure 6.12, we illustrate the way our problem instance is progressively solved. Before the first cycle c_2, all we have is the root operator and its plan π_3. Then within the first cycle, we select the composite action $make(tea)$ instantiated from the operator $make(drink)$ along with its methods. All related flaws are fixed until all that is left in the agenda is the abstract flaws. We save the partial plan π_2 for this cycle and expand $make(tea)$ into a copy of the current plan π_1 for the next cycle. The solution of the problem will be stored in π_0 once found.

6.3.4 Properties of Abstract Planning

In this section, we prove several properties of our method and resulting plans: HEART is complete, sound and its abstract plans can always be decomposed into a valid solution.

The completeness and soundness of POCL has been proven in (Penberthy *et al.* 1992). An interesting property of POCL algorithms is that flaw selection strategies do not impact these properties. Since the only modification of the algorithm is the extension of the classical flaws with a decomposition flaw, all we need to explore, to update the proofs, is the impact of the new resolver. By definition, the resolvers of decomposition flaws will take into account all flaws introduced by its resolution into the refined plan. It can also revert its application properly.

Lemma 5 (Decomposing preserves acyclicity)**.** *The decomposition of a composite action with a valid method in an acyclic plan will result in an acyclic plan. Formally* $\forall a_s \in S_{\pi} : a_s \not\succ_{\pi} a_s \implies \forall a'_s \in S_{a \oplus^m_{\pi}} : a'_s \not\succ_{a \oplus^m_{\pi}} a'_s$.

Proof. When decomposing a composite action a with a method m in an existing plan π, we add all steps S_m in the refined plan. Both π and m are guaranteed to be cycle free by definition. We can note that $\forall a_s \in S_m : \left(\nexists a_t \in S_m : a_s \succ a_t \wedge \neg f \in \mathit{eff}(a_t)\right) \implies f \in \mathit{eff}(a)$. Said otherwise, if an action a_s can participate a fluent f to the goal step of the method m then it is necessarily present in the effects of a. Since higher level actions are preferred during the resolver selection, no actions in the methods are already used in the plan when the decomposition happens. This can be noted $\exists a \in \pi \implies S_m \cup S_{\pi}$ meaning that in the graph formed both partial plans m and π cannot contain the same edges therefore their acyclicity is preserved when inserting one into the other.

□

Lemma 6 (Solved decomposition flaws cannot reoccur)**.** *The application of a decomposition resolver on a plan* π*, guarantees that* $a \notin S_{\pi'}$ *for any partial plan refined from* π *without reverting the application of the resolver.*

Proof. As stated in the definition of the methods (definition 30): $a \notin A_a$. This means that a cannot be introduced in the plan by its decomposition or the decomposition of its proper actions. Indeed, once a is expanded, the level of the following cycle $c_{lv(a)-1}$ prevents a to be selected by subgoal resolvers. It cannot either be contained in the methods of another action that are selected afterward because otherwise following definition 50 its level would be at least $lv(a) + 1$.

□

Lemma 7 (Decomposing to abstraction level 0 guarantees solvability)**.** *Finding a partial plan that contains only decomposition flaws with actions of abstraction level 1, guarantees a solution to the problem.*

Proof. Any method m of a composite action $a : lv(a) = 1$ is by definition a solution of the problem $\mathbb{p}_a = \langle \mathcal{D}, C_{\mathbb{p}}, a \rangle$. By definition, $a \notin A_a$, and $a \notin A_{a \oplus_{\mathbb{\pi}}^m}$ (meaning that a cannot reoccur after being decomposed). It is also given by definition that the instantiation of the action and its methods are coherent regarding variable constraints (everything is instantiated before selection by the resolvers). Since the plan $\mathbb{\pi}$ only has decomposition flaws and all flaws within m are guaranteed to be solvable, and both are guaranteed to be acyclical by the application of any decomposition $a \oplus_{\mathbb{\pi}}^m$, the plan is solvable.

□

Lemma 8 (Abstract plans guarantee solvability)**.** *Finding a partial plan $\mathbb{\pi}$ that contains only decomposition flaws, guarantees a solution to the problem.*

Proof. Recursively, if we apply the previous proof on higher level plans we note that decomposing at level 2 guarantees a solution since the method of the composite actions are guaranteed to be solvable.

□

From these proofs, we can derive the property of soundness (from the guarantee that the composite action provides its effects from any methods) and completeness (since if a composite action cannot be used, the planner defaults to using any action of the domain).

6.3.5 Computational Profile

In order to assess its capabilities, HEART was evaluated on two criteria: quality and complexity. All tests were executed on an Intel® Core™ i7-7700HQ CPU clocked at 2.80GHz. The Java process used only one core and was not limited by time or memory (32 GB that wasn't entirely used up) . Each experiment was repeated between 700 and 10000 times to ensure that variations in speed were not impacting the results.

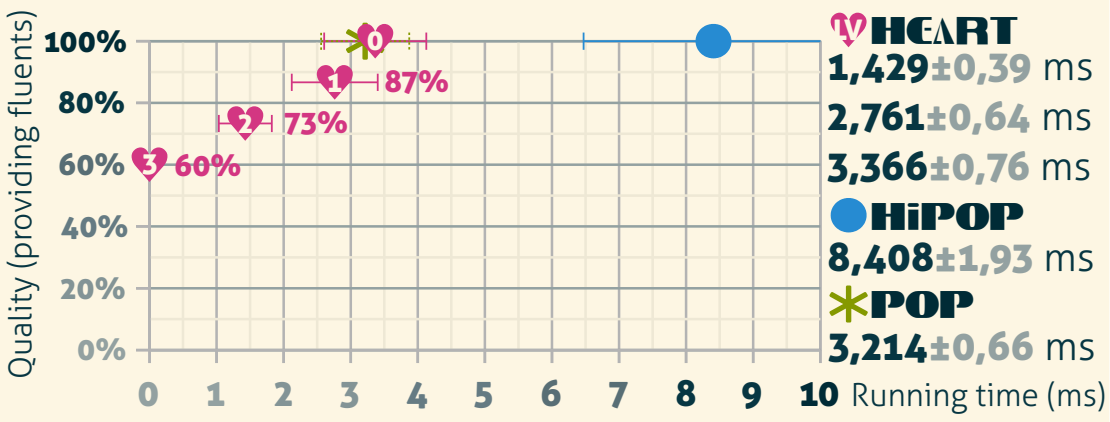


Figure 6.13: Evolution of the quality with computation time.

Figure 6.13 shows how the quality is affected by the abstraction in partial plans. The quality is measured by counting the number of providing fluents in the plan $\left|\bigcup_{a \in S_{\pi}} eff(a)\right|$. This metric is actually used to approximate the probability of a goal given observations in intent recognition ($P(G|O)$ with noisy observations, see (Sohrabi *et al.* 2016)). The percentages are relative to the total number of unique fluents of the complete solution. These results show that in some cases it may be more interesting to plan in a leveled fashion to solve HTN problems. For the first cycle of level 3, the quality of the abstract plan is already of 60%. This is the quality of the exploitation of the plan *before any planning*. With almost three quarters of the final quality and less than half of the complete computation time, the result of the first cycle is a good quality/time compromise.

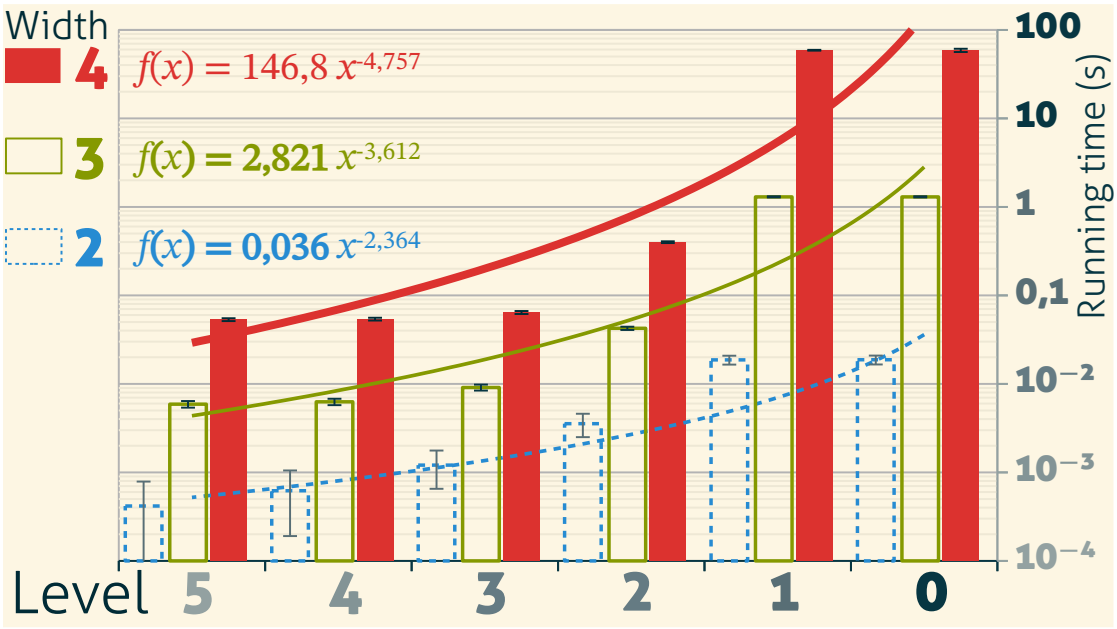


Figure 6.14: Impact of domain shape on the computation time by levels. The scale of the vertical axis is logarithmic. Equations are the definition of the trend curves.

In the second test, we used generated domains. These domains consist of an action of abstraction level 5. This action has a single method containing a number of actions of levels 4. We call this number the width of the domain. All needed actions are built recursively to form a tree shape. Atomic actions only have single fluent effects. The goal is the effect of the higher level action and the initial state is empty. These domains do not contain negative effects. Figure 6.14 shows the computational profile of HEART for various levels and widths. We note that the behavior of HEART seems to follow an exponential law with the negative exponent of the trend curves seemingly being correlated to the actual width. This means that computing the first cycles has a complexity that is close to being *linear* while computing the last cycles is of the same complexity as classical planning which is at least *P-SPACE* (depending on the expressivity of the domain) (Erol *et al.* 1995).

6.4 Conclusion

In this chapter we showed two planners that are oriented toward real time and flexibility. This makes fast decision-making easier and improves the expressive power of domain writing tools such as the one we presented in chapter 5.

Such planners may be used in intent recognition using inverted planning. This technique is described in the next chapter.

7 Toward Intent Recognition

Since the original goal of this thesis was on intent recognition of dependent persons, we need to explain some more about that specific domain. The problem is to infer the goals of an external agent through only observations without intervention. In the end, the idea is to infer that goal confidently enough to start assisting the other agent without explicit instructions.

7.1 Domain problems

This field was widely studied. Indeed, at the end of the last century, several works started using *abduction* to infer intents from observational data. Of course this comes as a challenge since there is a lot of uncertainty involved. We have no reliable information on the possible goals of the other agent and we don't have the same knowledge of the world. Also, observations can sometimes be incomplete or misleading, the agent can abandon goals or pursue several goals at once while also doing them sub-optimally or even fail at them. To finish, sometimes agents can actively try to hide their intents to the viewer.

7.1.1 Observations and inferences

As explained above, we can only get close to reality and any progress in relevance and detail is exponentially expensive in terms of computing and memory resources. That is why any system will maintain a high degree of abstraction that will cause errors inherent in this approximation.

This noise phenomenon can impact the activity and situation recognition system and therefore seriously impact the intention recognition and decision-making system with an amplification of the error as it is processed. It is also important to remember that this phenomenon of data noise is also present in inhibition and that the lack of perception of an event is as disabling as the perception of the wrong event.

It is possible to protect recognition systems in an appropriate way, but this often implies a restriction on the levels of possibilities offered by the system such as specialized recognition or recognition at a lower level of relevance.

7.1.2 Cognitive inconsistencies

In the field of personal assistance, activity recognition is a crucial element. However, it happens that these events are very difficult to recognize. The data noise mentioned above can easily be confused with an omission on the part of the observed agent. This dilemma is also present during periods of inactivity, the system can start creating events from scratch to fill what it may perceive as inhibition noise.

These problems are accompanied by others related to the behavior of the observed agent. For example, they may perform unnecessary steps, restart ongoing activities or suddenly abandon them. It is added that other aspects of observations can make automated inferences such as ambiguous actions or the agent performing an action that resembles another complicated.

However, some noise problems can be easily detected by simple cognitive processes such as impossible sequences (e. g. closing a closed door). Contextual analyses provide a partial solution to some of these problems.

7.1.3 Sequentiality

Since our recognition is based on a highly temporal planning aspect, we must take into account the classic problems of sequentiality.

A first problem is to determine the end of one plan and the beginning of another. Indeed, it is possible that some transitions between two planes may appear to be a plane in itself and therefore may cause false positives. Another problem is that of intertwined planes. A person can do two things at once, such as answering the phone while cooking. An action in an intertwined plan can then be identified as a discontinuation of activity or a logical inconsistency. A final problem is that of overloaded actions. Not only can an agent perform two tasks simultaneously, but also perform an action that contributes to two activities. These overloaded actions make the process of intention recognition complex because they are close to data noise.

7.2 Existing approaches

The problem of intention recognition has been strongly addressed from many angles. It is therefore not surprising that there are many paradigms in the field. The first studies on the subject highlight the fact that intention recognition problems are problems of abductive logic or graph coverage. Since then, many models have competed in imagination and innovation to improve the field. These include constraint system-based models that provide a solution based on pre-established rules and compiled plan libraries, those that use state or action networks that then launch algorithms on this data, and reverse planning systems.

7.2.1 Constraint

One of the approaches to intention recognition is the one that builds a system around a strong logical constraint. There is often a time constraint system that is complemented by various extensions to cover as many sequential problems as possible.

7.2.1.1 Deductive Approach

In order to solve a problem of intention recognition, abductive logic can be used. Contrary to deductive logic, the goal is to determine the objective from the observed actions. Among the first models introduced is Goldman *et al.* (1999)'s model, which uses the principle of action to construct a logical representation of the problem. This paradigm consists in creating logical rules as if the action in question was actually carried out, but in hypothesizing the predicates that concretize the action and thus being able to browse the research space thus created in order to find all the possible plans to contain the observed actions and concretizing defined intentions. This model is strongly based on first-order logic and SWI Prolog logic programming languages. Although revolutionary for the time, this system pale in comparison to recent systems, particularly in terms of prediction accuracy.

7.2.1.2 Algebraic Approach

Some paradigms use algebra to determine possible plans from observed actions. In particular, we find the model of Bouchard *et al.* (2006) which extends the subsumption relationship from domain theory to the description of action and sequence of action in order to introduce it as an order relationship in the context of the construction of a lattice composed of possible plans considering the observed actions. This model simply takes into account the observed actions and selects any plan from the library of plans that contains at least one observed action. Then this paradigm will construct all the possible plans that correspond to the Cartesian product of the observed actions with the actions contained in the selected plans (while respecting their order). This system makes it possible to obtain a subsumption relationship that corresponds to the fact that the plans are more or less general. Unfortunately, this relationship alone does not provide any information on which plan is most likely.

That is why Roy *et al.* (2011) created a probabilistic extension of this model. This uses frequency data from a system learning period to calculate the influence probabilities of each plane in the recognition space. This makes it possible to calculate probabilistic intervals for each plan, action as well as for a plan to know a given action. In order to determine the probability of each plane knowing the upper bound of the lattice (plane containing all observed actions) the sum of the conditional probabilities of the plane for each

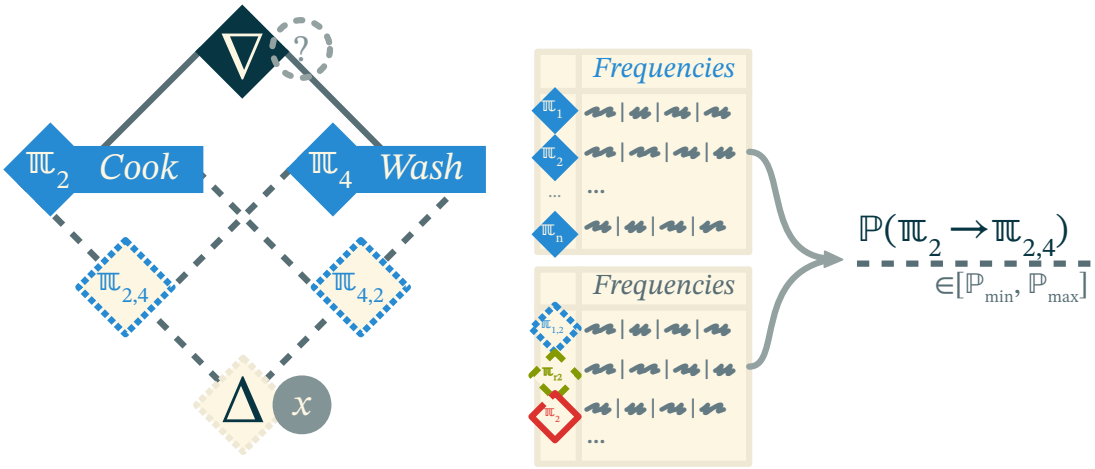


Figure 7.1: The lattice formed by observations (top), matching plans, possible hypothesis and problem (bottom)

observed action divided by the number of observed actions is made. This gives a probability interval for each plane allowing the ordinates. This model has the advantage of considering many possible plans but has the disadvantage of seeing a computational explosion as soon as the number of observed actions increases and the context is not taken into account.

7.2.1.3 Grammatical Approach

Another approach is that of grammar. Indeed, we can consider actions as words and sequences as sentences and thus define a system that allows us to recognize shots from incomplete sequences. Vidal *et al.* (2010) has therefore created a system of intention recognition based on grammar. It uses the evaluated grammar system to specify measurements from observations. These measures will make it possible to select specific plans and thus return a hierarchical hypothesis tree with the actions already carried out, the future and the plans from which they are derived. This model is very similar to first order logic-based systems, and uses a SWI Prolog type logic language programming system. Given the scope of maritime surveillance, this model, although taking very well into account the context and the evolution of the measures, is only poorly adapted to an application in assistance, particularly in the absence of a system for discriminating against results plans.

7.2.1.4 Linear programming approach

Another class of approaches is that of diverting standard problem-solving tools to solve the problem of intention recognition. It is therefore possible, by modifying traditional algorithms or by transforming a problem, to ensure that the solution of the tool corresponds to the one sought.

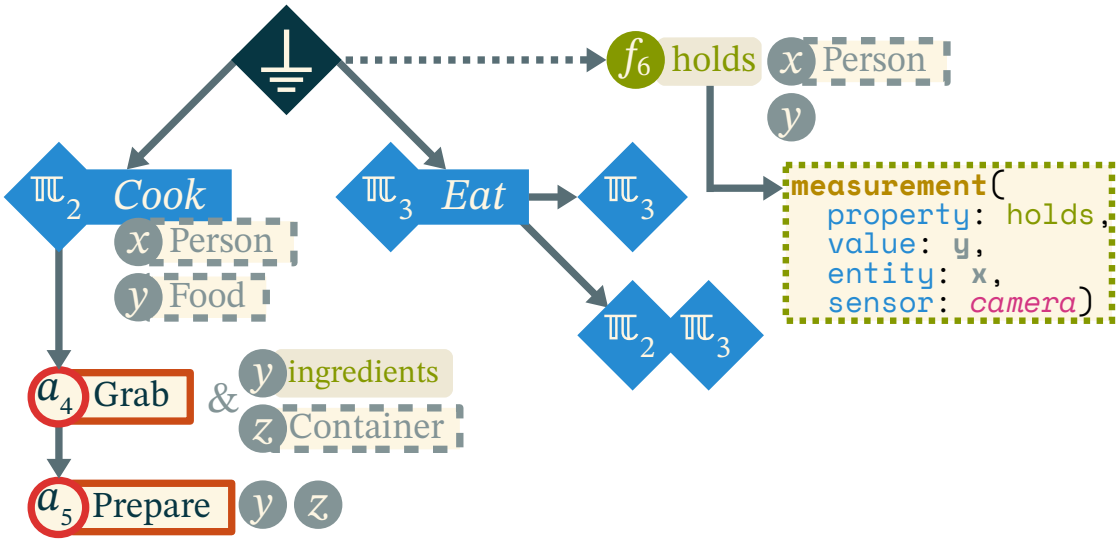


Figure 7.2: The valued grammar used for intent recognition

Inoue and Inui (2011) develops the idea of a model that uses linear programming to solve the recognition problem. Indeed, observations are introduced in the form of causes in relation to hypotheses, in a first-order logic predicate system. Each atom is then weighed and introduced into a process of problem transformation by feedback and the introduction of order and causality constraints in order to force the linear program toward optimal solutions by taking into account observations. Although ingenious, this solution does not discriminate between possible plans and is very difficult to apply to real-time recognition situations, mainly because of the problem transformation procedure required each time the problem is updated.

7.2.1.5 Markovian Logic Approach

Another constraint paradigm is the one presented by Raghavana *et al.* using a Markovian extension of first-order logic. The model consists of a library of plans represented in the form of Horn clauses indicating which actions imply which intentions. The aim is therefore to reverse the implications in order to transform the deduction mechanism into an abduction mechanism. Exclusionary constraints and a system of weights acquired through learning must then be introduced to determine the most likely intention. Once again, despite the presence of a system of result discrimination, there is no consideration of context and abductive transformation remains too cumbersome a process for real-time recognition.

7.2.2 Networks

7.2.2.1 And/Or trees approach

As in its early days, intention recognition can still be modeled in the form of graphs. Very often in intention recognition, trees are used to exploit the advantages of acyclicity in resolution and path algorithms. In the prolific literature of Geib et al. we find the model at the basis of PHATT (Geib 2002) which consists of an AND/OR tree representing a HTN that contains the intentions as well as their plans or methods. A prior relationship is added to this model and it is through this model that constraints are placed on the execution of actions. Once an action is observed, all the successors of the action are unlocked as potential next observed action. We can therefore infer by hierarchical path the candidate intentions for the observed sequence.

Since this model does not allow discrimination of results, Geib and Goldman (2005) then adds probabilities to the explanations of the observations. The degree of coverage of each possible goal is used to calculate the probability of each goal. That is, the goal with the plan containing the most observed action and the least unobserved action will be the most likely. This is very ingenious, as the coverage rate is one of the most reliable indicators. However, the model only takes into account temporality and therefore has no contextual support. The representation in the form of a tree also makes it very difficult to be flexible in terms of the plans, which are then fixed a priori.

7.2.2.2 HTN Approach

The HTN model is often used in the field, such as the hierarchical tree form used by Avrahami-Zilberbrand *et al.* (2005). The tree consists of nodes that represent various levels of action and intent. A hierarchical relationship links these elements together to define each intention and its methods. To this tree is added an anteriority relationship that constrains the execution order. This paradigm uses time markers that guarantee order to use an actualization algorithm that also updates a hypothesis tree containing possible intentions for each observation.

A probabilistic extension of the Avrahami-Zilberbrand and Kaminka (2006) applies a hierarchical hidden Markov model to the action tree. Using three types of probability that of plan tracking, execution interleaving and interruption, we can calculate the probability of execution of each plan according to the observed sequence. The logic and contextual model filtered on the possible plans upstream leaving us with few calculations to order these plans.

This contextual model uses a decision tree based on a system of world properties. Each property has a finite (and if possible very limited) number of possible values. This allows you to create a tree containing for each node a property and an arc for each value. This is combined with other nodes or leaves that are actions. While running through the tree during execution,

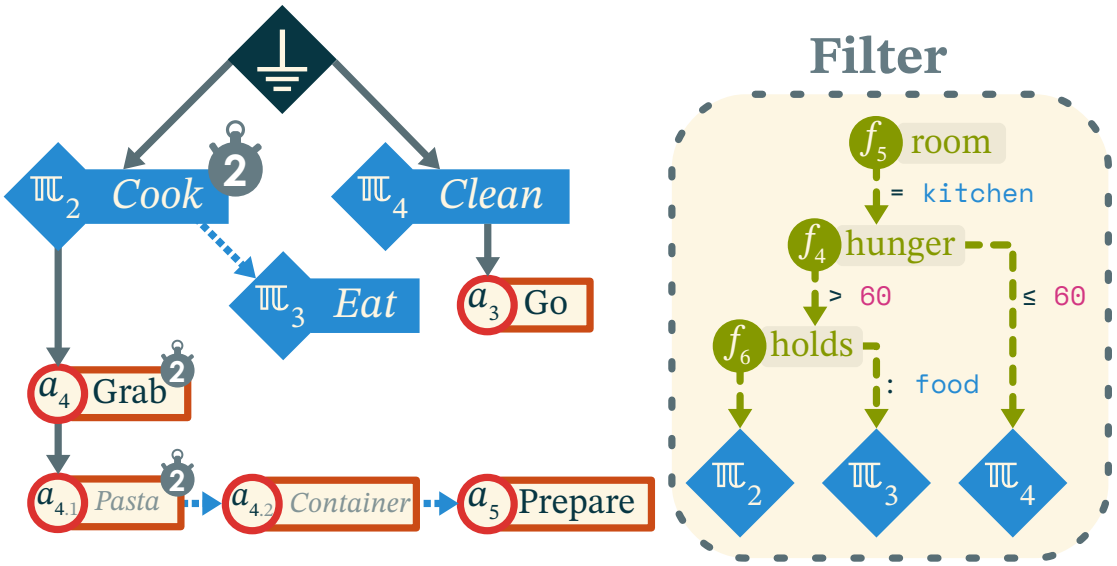


Figure 7.3: The HTN and decision tree used for intent recognition

the branches that do not correspond to the current value of each property are pruned. Once a leaf is reached, it is stored as a possible action. This considerably reduces the research space but requires a balanced tree that is not too large or restrictive.

7.2.2.3 Hidden Markovian Approach

When we approach stochastic models, we very often find Markovian or Bayesian models. These models use different probabilistic tools ranging from simple probabilistic inference to the fusion of stochastic networks. It can be noted that probabilities are often defined by standard distributions or are isomorphic to weighted systems.

A stochastic model based on THRs is the one presented by Blaylock and Allen (2006). This creates hierarchical stacks to categorize abstraction levels from basic actions to high level intentions. By chaining a hidden Markov model to these stacks, the model is able to affect a probability of intention according to the observed action.

7.2.2.4 Bayesian Approach

Another stochastic paradigm is the one of Han and Pereira (2013). It uses Bayesian networks to define relationships between causes, intentions and actions in a given field. Each category is treated separately in order to reduce the search space. The observed actions are then selected from the action network and extracted. The system then uses the intention network to build a temporary Bayesian network using the NoisyOR method. The network created is combined in the same way with the network of causes and makes it

possible to have the intention as well as the most probable cause according to the observations.

7.2.2.5 Markovian network approach

The model of Kelley *et al.* (2012) (based on (Hovland *et al.* 1996)) is a model using hidden Markov networks. This stochastic network is built here by learning data from robotic perception systems. The goal is to determine intent using past observations. This model uses the theory of mind by invoking that humans infer the intentions of their peers by using a projection of their own.

Another contextual approach is the one developed for robotics by Hofmann and Williams (2007). The stochastic system is completed by weighting based on an analysis of vernacular corpuses. We can therefore use the context of an observation to determine the most credible actions using the relational system built with corpus analysis. This is based on the observation of the objects in the scene and their condition. This makes common sense actions much more likely and almost impossible actions leading to semantic contradictions.

7.2.2.6 Bayesian Theory of Mind

This principle is also used as the basis of the paradigm of Baker and Tenenbaum (2014) which forms a Bayesian theory of the mind. Using a limited representation of the human mind, this model defines formulas for updating beliefs and probabilities a posteriori of world states and actions. This is constructed with sigmoid distributions on the simplex of inferred beliefs. Then the probabilities of desire are calculated in order to recover the most probable intention. This has been validated as being close to the assessment of human candidates on simple intention recognition scenarios.

7.3 Inverted planning

Another way to do intent recognition do not rely on having a plan library at all by using inverted planning. In fact, intent recognition is the opposite problem as planning. In planning we compute the plan from the goal and in intent recognition we seek the goal from the plan. This means that planning is a *deduction* problem while intent recognition is an *abduction* problem. It is therefore possible to transform an intent recognition problem into a planning one.

More intuitively, this transformation relies on the *theory of mind*. This notion of psychology states that one of the easiest ways to predict the mind of another agent is by projecting our own way of thinking onto the target. The

familiar way to understand this is to ask the question, "what would **I** do if I were them ?". This is obviously imperfect since we don't have the complete knowledge of the other mind but is often good enough at basic predictions.

This theory is based on the Belief, Desire and Intention (BDI) model. In our case the belief part is akin to the knowledge database, the desires are the set of possible goals (weighted by costs) and the intent is the plan that achieve a selected goal.

A good analysis of this way of thinking in the context of intent recognition can be found in Baker *et al.* (2007)'s work on the subject.

To get further, the work of Ramırez and Geffner (2009) is the founding paper on the principle of transforming the intent recognition problem into a planning one. That work was later improved by Chen *et al.* (2013) in order to support multiple and concurrent goals at once.

In order to do that Ramirez rediscovers an old tool called constraints encoding into planning Baioletti *et al.* (1998). This allows to force the selection of operators in a given a certain order or adding arbitrary constraints on the solution.

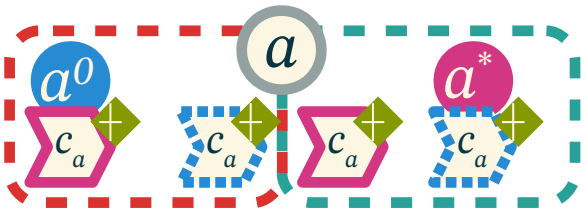


Figure 7.4: Representation of the encoding of constraints using extra fluents.

The encoding on itself is quite straight forward : Adding artificial fluents to derive an action's behavior considering its selection. In the case of Ramirez's work, the fluents ensure that the observed actions are selected at the start. The resulting plan is then compared to another plan computed while avoiding the observed action. The difference in cost is proportional to the likelihood of the goal to be pursued.

This problem transformation was more recently improved significantly by Sohrabi *et al.* (2016). Indeed, their work allows for using observed fluents instead of actions. This modification allows for a more accurate and flexible prediction with less advanced observations. It also takes into account the missing and noisy observation to affect negatively the likelihood of a goal. Along with the use of diverse planning, this technique allows for seamless multi-goal recognition.

In order to see this technique used in practice we refer to the works of Talamadupula *et al.* (2011).

7.3.1 Probabilities and approximations

Now that the intuition is covered, we need to prove that the result of the planning process is indeed correlated to the probability of the agent pursuing that plan knowing the observations. But first we need to formalize how probabilities work.

An *event* is a fixed fluent, a logical proposition that can occur. The likelihood of an event happening ranges from 0 (impossible) to 1 (certain). This is represented by a relation named **probability** of any event e noted $\mathbb{P}(e) \in [0,1]_{\mathbb{R}}$. This means that probabilities are real numbers restricted between 0 and 1.

Definition 53 (Conditional probabilities)**.** Conditional probabilities are probabilities of an event assuming that another related event happened. This allows to evaluate the ways in which events are affecting one another. The probability of the event A occurring knowing that B occurred is written:

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}$$

with, $\mathbb{P}(A \cap B)$ being the probability that both events occur. In the case of two *independent* events $\mathbb{P}(A|B) = \mathbb{P}(A)$.

We note the set of goals that can be pursued G and the temporal sequence of observations $\mathcal{O}$. Using conditional probabilities, we seek to have a measure of $\mathbb{P}(g|\mathcal{O})$ for any goal $g \in G$.

In that section we explain how inverted planning does that computation.

For any set of observations $\mathcal{O}$ the probability of the set is the product of the probability of any observation $o \in \mathcal{O}$. We can then note $\mathbb{P}(\mathcal{O}) = \prod_{o \in \mathcal{O}} \mathbb{P}(o)$.

We assume that the observed agent is pursuing one of the known goals. The event of an agent pursuing a specific goal is noted g. This means that $\mathbb{P}(G) = \sum_{g \in G} \mathbb{P}(G) = 1$ because the event is considered certain.

Using *conditional probabilities* we can also note $\mathbb{P}(G|\pi) = 1$ for a valid plan π that achieves any goals $g \in G$.

Theorem 4 (Bayes)**.** *Bayes's theorem allows to find the probability of an event based on prior knowledge of other factors related to said event. It is another basic way to compute conditional probabilities as follows:*

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}$$

In the Bayesian logic, one should start with a *prior* probability of an event and actualize it with any new information to make it more precise. In our case, we suppose that $\mathbb{P}(G)$ is given or computed by an external tool.

From the direct application of Bayes's theorem and the previous assumptions, we have :

$$\mathbb{P}(\pi|\mathcal{O}) = \frac{\mathbb{P}(\mathcal{O}|\pi)\mathbb{P}(\pi)}{\mathbb{P}(\mathcal{O})} = \frac{\mathbb{P}(\mathcal{O}|\pi)\mathbb{P}(\pi|G)\mathbb{P}(G)}{\mathbb{P}(\mathcal{O})} \tag{7.1}$$

Using the event π transitively we can simplify to:

$$\mathbb{P}(G|\mathcal{O}) = \frac{\mathbb{P}(\mathcal{O}|G)\mathbb{P}(G)}{\mathbb{P}(\mathcal{O})} \tag{7.2}$$

Theorem 5 (Total Probability)**.** *If we have a countable set of disjoint events E we can compute the probability of all events happening as the sum of the probabilities of each event:*

$$\mathbb{P}(E) = \sum_{e \in E} \mathbb{P}(e)$$

Since we consider that we have all likely plans for a given goal we can neglect the very improbable ones and assert that the events of any given plans being acted are independent from one another. Also, the total probability of all plans is certain. From the total probability formula:

$$\mathbb{P}(\mathcal{O}|G) = \sum_{\pi \in \Pi_G} \mathbb{P}(\mathcal{O}|\pi)\mathbb{P}(\pi|G) \tag{7.3}$$

In equation 7.2, we have $\mathbb{P}(G)$ and $\mathbb{P}(\mathcal{O})$ known from prior knowledge. Along with equation 7.3, we can say that:

$$\mathbb{P}(G|\mathcal{O}) = \alpha\mathbb{P}(\mathcal{O}|G)\mathbb{P}(G)$$

With α being a normalizing constant. Also, using the previous formula we can assert that:

$$\mathbb{P}(g|\mathcal{O}) \propto \sum_{\pi \in \Pi_g} \mathbb{P}(\pi|\mathcal{O})$$

This means that if the cost of the plan is related to its likeliness of being pursued knowing the observation sequence, we can evaluate the probability of any goal being pursued. This allows for Sohrabi's problem transformation to work.

That transformation is simply affecting the cost of a plan by dissuading any missed or added fluents while rewarding correct predicted fluents relative to the observations. This process is illustrated in figure 7.5.

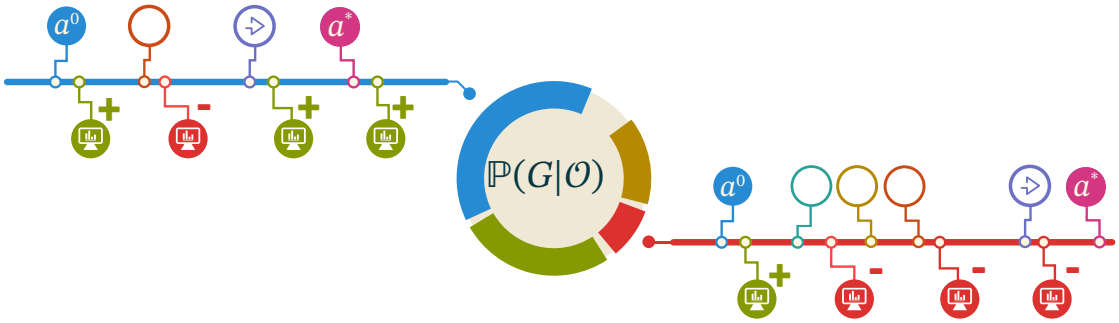


Figure 7.5: Illustration of how plan costs are affecting the resulting probabilities.

7.4 Intent recognition Using Abstract Plans

The initial objective of this thesis was to make intent predictions use abstract plans and repairs. Since plan repair is very susceptible to the heuristic, it is not a reliable tool for general and uncertain planning and will perform worse on all cases not handled well by a given heuristic.

This is not the case of abstract HTN planning since the plan generated is valid while incomplete. The idea here is to quantify the quality of an abstract plan to weigh its plausibility and the likelihood of missing fluents using Sohrabi's method.

The unforeseen problem is that, while in theory this seems like a very efficient approach, in practice it can lack a lot of performance since turning a sequence of observed fluents into a backward chaining heuristic is tricky at best.

In this section, some idea of how that could be done is explored along with perspectives for further works regarding the subject.

7.4.1 Linearization

In order to use the approach of Sohrabi, we need total ordered plans. This is quite an issue since our planner generates partial order plans. Each of these plans have one or several *linearizations*: totally ordered plans that correspond to all possible orders of the plan.

The idea here is to merge parallel actions into one using graph quotient. This is the same mechanism behind sheaves. To do this we use the fact that there are no threats in valid plans and therefore parallel actions have compatible preconditions and effects. This allows to merge several actions into one that is equivalent in terms of fluents and cost. To merge two actions into one we do the following: $pre(a_m) = pre(a_1) \circ pre(a_2)$ and $eff(a_m) = eff(a_1) \circ eff(a_2)$. We use the application of states over other states and ignore the order.

7.4.2 Abstraction

Since abstract plans use composite actions that have explicit preconditions and effects, they can be treated as a normal action. Indeed, while it is possible that an abstract action has fewer requirements and effects than what is done in its methods, it can't have more since the methods must *at least* fulfill the parent action's "contract."

Adding to that, while merging actions in the linearization step, it may be appropriate to merge actions into a composite action that holds all the merged actions in its only method.

7.4.3 Example

In this section we present an example as well as a description of the execution of our planning algorithm and how the results are used for intent recognition.

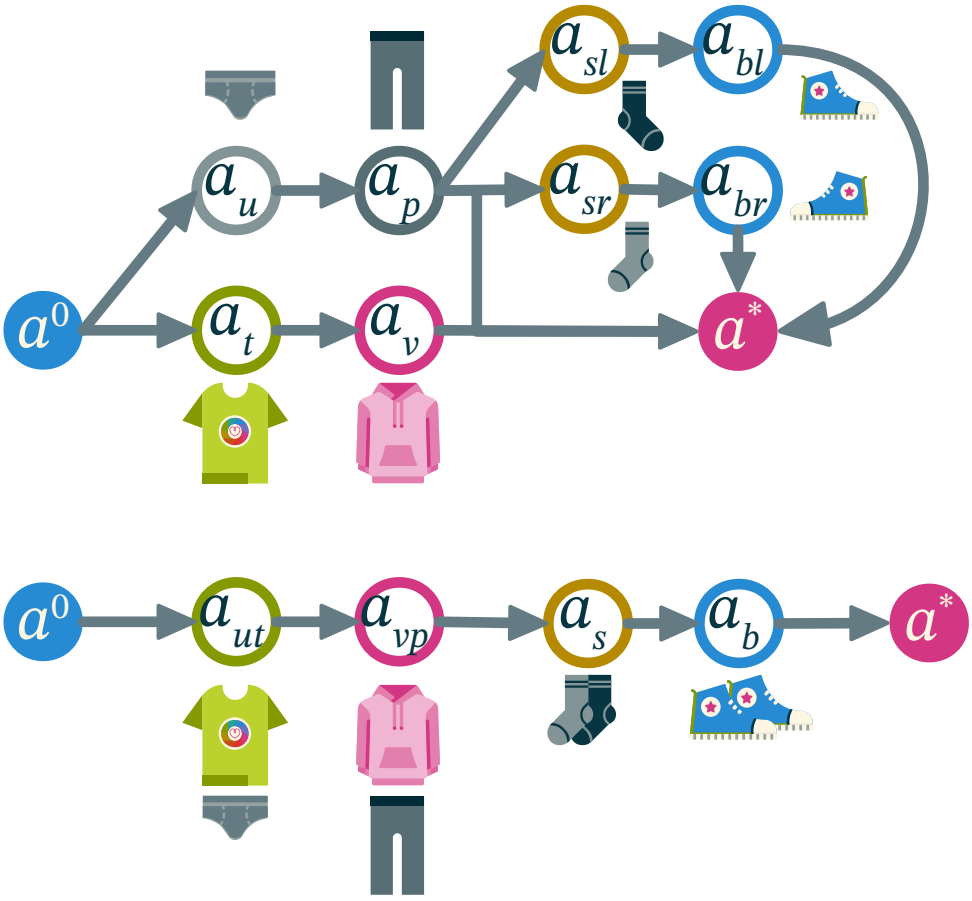


Figure 7.6: Example of linearization of partial order plans.

Example 53. Figure 7.6 illustrates an example of such a linearization. The linearization happens by merging actions that can be done simultaneously.

Of course, this is not a valid totally ordered plan since classical planning supposes that the agent can only perform an action at once.

In our example, a person wants to get dressed. In a partial order plan there is no need to order between top clothing and bottom clothing. It is also necessary to specify if the right or left socks and shoes should be put on first.

In a first time, we explain how the planning algorithm can affect the result of the intent recognition.

7.4.3.1 Partial Order Approach

Using a previously described POP algorithm (chapter 6) we can create a planning domain with the following actions from the example in figure 7.6: a_u, a_t, a_p, a_v, $a_{s(l|r)}$ and $a_{b(l|r)}$. Each action instinctively requires you to wear clothes that are underneath their related garment and put it on. For example, the action a_{br} requires the right socket to be put on (effect of a_{sr}) and will result in putting the right shoe on.

The goal is to put on all the clothes from a state of none are already put on. The planner will select each action to fulfill the goal and add causal links to prevent threats between actions and to fulfill their preconditions. The resulting plan is the upper one in figure 7.6.

Now the intent recognition part quick in and linearize the plan. To do so, the algorithm identifies actions that can be done simultaneously and merge them into a single action. The purpose of this merging is to make the plan totally ordered while keeping fluents in chronological order. Here we can see that the linear plan describes actions of putting clothes by layers instead of by the garment. Details are lost but the chronological order and cost is kept (by adding costs of all actions being merged).

This allows for existing intent recognition methods to be applied on the resulting plan without losing on the advantages of partial order plans. This technique results in the plan on the bottom of figure 7.6. This plan is totally ordered and can be used by the classical inverted planning approaches as described previously.

7.4.3.2 Abstract Plan Approach

This example also illustrates the advantages of using abstract plans for intent recognition. If the domain is properly designed for it, we can use an abstract plan that has composite actions made by layers of clothing.

In that approach, we reuse the previous domain but extend it with the following actions: a_{ut}, a_{vp}, a_s, a_b. These actions will behave in the same way as the merged actions of the linearization. Each action corresponds to a layer

of clothing and can be decomposed into the atomic actions relative to that layer. For example, the action a_{ut} is the action related to putting on the first layer of clothing. It can be decomposed into the action of putting underwear a_u and the action of putting on a t-shirt a_t.

Using our HEART planner, it is possible to request only an abstract plan to contain this level of action. The planning process will therefore result into the plan on the bottom of figure 7.6 that is also the linearized plan found earlier.

Since the abstract plans are easier to compute, the intent recognition becomes faster for the same result in that case. The main factor in the efficiency of this method is the design of the domain.

7.5 Conclusion

In this chapter, we present existing intent recognition techniques and expose how inverted planning can be fitted to our planning approach.

8 Conclusion and Perspectives

In this document, we underlined various issues, from knowledge representation to how planning can be used to guess each other's intent.

First we briefly exposed dome mathematical bases to our various formalisms. We built a coherent logical system to present the tools necessary to describe the theorical aspects of later contributions.

Once those tools presented, we used them to design a knowledge representation system that is partially described by structure and allows for higher order logic. This framework and dynamic language allows for a significant increase in expressivity while remaining concise and understandable.

Those qualities were needed in the task of providing automated planning with a unifying framework that can handle all existing paradigms using a general description of the planning process itself.

This makes the design of a general planning language possible. This language based on our earlier contribution makes hybrid domain description easier and is useful in making planner that aims to take advantage of several planning approaches at once.

Using this framework, we designed two planners:

- One for real-time plan repair, using an operator graph and a usefulness heuristic to guide the planning process.
- Another that allows for abstract intermediate plans to be returned before the planning process is complete.

Each approaches are evaluated against similar algorithms. Results shows that, at least on some problems, the planning process can be made faster using these kind of technique, especially when a complete solution is not required for a given application.

An example of such an application can be found in the last chapter. Using the theory of mind, it is possible to apply automated planning to the problem of intent recognition. We show the advantages of inverted planning and how the abstract plans of our latest prototype may be exploited for fast and accurate intent recognition.

8.1 Perspectives and discussions

8.1.1 Knowledge representation.

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.

8.1.1.1 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 8.1.

```
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 8.1: 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 8.2.

```
1 0 :: Integer;
2 *n : (++(n) :: Integer);
3 (*m, *n) : ((m=n) : (++m = ++n));
4 *n : (++n ~= 0);
5 *n : ((n + 0) = n);
6 (*n, *m) : ((n + ++m)= ++(n + m));
7 *n : ((n × 0) = 0);
8 (*n, *m) : ((n × ++m) = (n + (n × m)));
```

Listing 8.2: 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.

8.1.1.2 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.

8.1.1.3 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.

9 References

Alami, R., F. Robert, F. Ingrand, and S. Suzuki
Multi-robot cooperation through incremental plan-merging, *Robotics and Automation, 1995. Proceedings., 1995 IEEE International Conference on*, IEEE, 1995, 3,2573–2579.

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.

Ambite, J. L., and C. A. Knoblock
Planning by Rewriting: Efficiently Generating High-Quality Plans., DTIC Document, 1997.

American Heritage Dictionary
Formal (adj.), *American Heritage Dictionary of the English Language*, 2011a.

American Heritage Dictionary
Circularity (n.d.), *American Heritage Dictionary of the English Language*, 2011b.

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

Avrahami-Zilberbrand, D., and G. A. Kaminka
Hybrid symbolic-probabilistic plan recognizer: Initial steps, *Proceedings of AAAI workshop on modeling others from observations (MOO-06)*, 2006.

Avrahami-Zilberbrand, D., G. A. Kaminka, and H. Zarosim
Fast and complete plan recognition: Allowing for duration, interleaved execution, and lossy observations, *IJCAI Workshop on Modeling Others from Observations*, 2005.

Awodey, S.
Category theory, 2nd ed. Oxford logic guides 52Oxford ; New York: Oxford University Press, 2010.

Baader, F., D. Calvanese, D. McGuinness, P. Patel-Schneider, and D. Nardi
The description logic handbook: Theory, implementation and applications, Cambridge university press, 2003.

Babli, M., E. Marzal, and E. Onaindia
On the use of ontologies to extend knowledge in online planning, *KEPS 2018*, 54, 2015.

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

Baioletti, M., S. Marcugini, and A. Milani
Encoding planning constraints into partial order planning domains, *International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann Publishers Inc., 1998, 608–616.

Baker, C. L., R. R. Saxe, and J. B. Tenenbaum
Bayesian theory of mind: Modeling joint belief-desire attribution, *Proceedings of the thirty-second annual conference of the cognitive science society*, 2011, 2469–2474.

Baker, C. L., and J. B. Tenenbaum
Modeling Human Plan Recognition using Bayesian Theory of Mind, 2014.

Baker, C. L., J. B. Tenenbaum, and R. R. Saxe
Goal inference as inverse planning, *Proceedings of the Annual Meeting of the Cognitive Science Society*, Vol. 29, 2007.

Barendregt, H. P.
The Lambda Calculus: Its Syntax and Semantics. 1984, *Studies in Logic and the Foundations of Mathematics*, 1984.

Barr, M., and C. Wells
Category theory for computing science, vol. 49Prentice Hall New York, 1990.

Bechon, P., M. Barbier, G. Infantes, C. Lesire, and V. Vidal
HiPOP: Hierarchical Partial-Order Planning, *European Starting AI Researcher Symposium*, IOS Press, 2014, 264,51–60.

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., and D. Höller
Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning 2018.

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

Blaylock, N., and J. Allen
Fast hierarchical goal schema recognition, *Proceedings of the National Conference on Artificial Intelligence*, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006, 21,796.

Borrajo, D.
Multi-agent planning by plan reuse, *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, International Foundation for Autonomous Agents and Multiagent Systems, 2013, 1141–1142.

Borrajo, D., A. Roubíčková, and I. Serina
Progress in Case-Based Planning, *ACM Computing Surveys*, 47 (2), 1–39, January 2015. doi:10.1145/2674024.

Bouchard, B., S. Giroux, and A. Bouzouane
A Smart Home Agent for Plan Recognition of Cognitively-impaired Patients., *Journal of Computers*, 1 (5), 53–62, 2006.

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

Castillo, L. A., J. Fernández-Olivares, O. Garcia-Perez, and F. Palao
Efficiently Handling Temporal Knowledge in an HTN Planner., *ICAPS*, 2006, 63–72.

Chen, J., Y. Chen, Y. Xu, R. Huang, and Z. Chen
A Planning Approach to the Recognition of Multiple Goals, *International Journal of Intelligent Systems*, 28 (3), 203–216, 2013. doi:10.1002/int.21565.

Chomsky, N.
Three models for the description of language, *IRE Transactions on information theory*, 2 (3), 113–124, 1956.

Coles, A., A. Coles, M. Fox, and D. Long
Popf2 : A forward-chaining partial order planner, *IPC*, 65, 2011.

Collins English Dictionary
Abstraction (n.d.), *Collins English Dictionary Complete and Unabridged*, 2014.

Curry, H. B., R. Feys, and W. Craig
Studies in Logic and the Foundations of Mathematics, *Combinatory logic*, Vol. 1North-Holland Amsterdam, 1958.

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

Dornhege, C., P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel
Semantic attachments for domain-independent planning systems, *Towards service robots for everyday environments*, Springer, 2012, 99–115.

Dvorak, F., A. Bit-Monnot, F. Ingrand, and M. Ghallab
A flexible ANML actor and planner in robotics, *Planning and Robotics (PlanRob) Workshop (ICAPS)*, 2014.

Edelkamp, S., and J. Hoffmann
PDDL2.2: The language for the classical part of the 4th international planning competition, *4th International Planning Competition (IPC'04), at ICAPS'04*, 2004.

Ephrati, E., and J. S. Rosenschein
Multi-agent planning as the process of merging distributed sub-plans, *In Proceedings of the Twelfth International Workshop on Distributed Artificial Intelligence (DAI-93*, Citeseer, 1993.

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.

Erol, K., D. S. Nau, and V. S. Subrahmanian
Complexity, decidability and undecidability results for domain-independent planning, *Artificial intelligence*, 76 (1-2), 75–88, 1995.

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.
Natural hierarchical planning using operator decomposition, *European Conference on Planning*, Springer, 1997, 195–207.

Fox, M., A. Gerevini, D. Long, and I. Serina
Plan Stability: Replanning versus Plan Repair., *ICAPS*, 2006, 6,212–221.

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.

Geib, C. W.
Problems with intent recognition for elder care, *Proceedings of the AAAI-02 Workshop "Automation as Caregiver*, 2002, 13–17.

Geib, C. W., and R. P. Goldman
Partial observability and probabilistic plan/goal recognition, *Proceedings*

of the International workshop on modeling other agents from observations (MOO-05), 2005.

Gerevini, A. E., A. Roubíčková, A. Saetti, and I. Serina
On the plan-library maintenance problem in a case-based planner, *International Conference on Case-Based Reasoning*, Springer, 2013, 119–133.

Gerevini, A., U. Kuter, D. S. Nau, A. Saetti, and N. Waisbrot
Combining Domain-Independent Planning and HTN Planning: The Duet Planner, *Proceedings of the European Conference on Artificial Intelligence*, 2008, 18,573–577.

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.

Godel, K., and G. W. Brown
The consistency of the axiom of choice and of the generalized continuum-hypothesis with the axioms of set theory, Princeton University Press Princeton, NJ, 1940.

Goldman, R. P., C. W. Geib, and C. A. Miller
A new model of plan recognition, *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, Morgan Kaufmann Publishers Inc., 1999, 245–254.

Göbelbecker, M., T. Keller, P. Eyerich, M. Brenner, and B. Nebel
Coming Up With Good Excuses: What to do When no Plan Can be Found, *Proceedings of the International Conference on Automated Planning and Scheduling*, AAAI Press, May 2010, 20,81–88.

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.

Han, T. A., and L. M. Pereira
Context-dependent incremental decision making scrutinizing the intentions of others via Bayesian network model construction, *Intelligent Decision Technologies*, 7 (4), 293–317, 2013.

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

Hart, P. E., N. J. Nilsson, and B. Raphael
A formal basis for the heuristic determination of minimum cost paths, *IEEE transactions on Systems Science and Cybernetics*, 4 (2), 100–107, 1968.

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

Helmert, M., G. Röger, and E. Karpas
Fast downward stone soup: A baseline for building planner portfolios, *ICAPS 2011 Workshop on Planning and Learning*, Citeseer, 2011, 28–35.

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.

Hirankitti, V., and T. Xuan
A meta-reasoning approach for reasoning with SWRL ontologies, *Internation Multiconference of Engineers*, 2011.

Hoffmann, J.
FF: The fast-forward planning system, *AI magazine*, 22 (3), 57, 2001.

Hofmann, A. G., and B. C. Williams
Intent Recognition for Human-Robot Interaction., *Interaction Challenges for Intelligent Assistants*, 2007, 60–61.

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.

Hovland, G. E., P. Sikka, and B. J. McCarragher
Skill acquisition from human demonstration using a hidden markov model, *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, Ieee, 1996, 3,2706–2711.

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.

Inoue, N., and K. Inui
ILP-Based Reasoning for Weighted Abduction., *Plan, Activity, and Intent Recognition*, 2011.

Jónsson, B.
Maximal algebras of binary relations, *Contemporary Mathematics*, 33, 299–307, 1984.

Kambhampati, S.
Design Tradeoffs in Partial Order (Plan space) Planning., *AIPS*, 1994, 92–97.

Kambhampati, S., A. Mali, and B. Srivastava
Hybrid planning for partially hierarchical domains, *AAAI/IAAI*, 1998, 882–888.

Kelley, R., A. Tavakkoli, C. King, A. Ambardekar, and M. Nicolescu
Context-based bayesian intent recognition, *Autonomous Mental Development, IEEE Transactions on*, 4 (3), 215–225, 2012.

Klein, S.
Meta-compiling Text Grammars As a Model for Human Behavior, *Proceedings of the 1975 Workshop on Theoretical Issues in Natural Language Processing*, TINLAP '75; Cambridge, Massachusetts: Association for Computational Linguistics, 1975, 84–88. doi:10.3115/980190.980217.

Klyne, G., and J. J. Carroll
Resource Description Framework (RDF): Concepts and Abstract Syntax, Language Specification W3C RecommendationW3C, 2004.

Korzybski, A.
Science and Sanity: An Introduction to Non-Aristotelian Systems and General Semantics, International Non-Aristotelian Library Publishing Company, 1933.

Korzybski, A.
Science and sanity; an introduction to non-Aristotelian systems and general semantics., Lakeville, Conn.: International Non-Aristotelian Library Pub. Co.; distributed by Institute of General Semantics, 1958.

Kovacs, D. L.
BNF Description of PDDL 3.1, Unpublished manuscript from the IPC-2011 websiteIPC, 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.

Krötzsch, M., F. Simancik, and I. Horrocks
A Description Logic Primer, June 2013.

Kunen, K.
Set theory an introduction to independence proofs, vol. 102Elsevier, 1980.

Lee, K. L. M. T. J.
Generating Qualitatively Different Plans through Metatheoretic Biases, 1999.

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.

Luis, N., and D. Borrajo
Plan merging by reuse for multi-agent planning, *Distributed and Multi-Agent Planning*, 38, 2014.

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.

Nebel, B., and J. Koehler
Plan reuse versus plan generation: A theoretical and empirical analysis, *Artificial Intelligence*, 76 (1), 427–454, 1995.

Nguyen, X., and S. Kambhampati
Reviving partial order planning, *IJCAI*, 2001, 1,459–464.

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, *Journal of Experimental & Theoretical Artificial Intelligence*, 30 (1), 143–176, 2018.

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

Peot, M. A., and D. E. Smith
Threat-removal strategies for partial-order planning, *AAAI*, 1993, 93,492–499.

Peot, M. A., and D. E. Smith
Postponing Threats in Partial-Order Planning 1994.

Raghavana, S., P. Singlab, and R. J. Mooneya
Plan Recognition using Statistical Relational Models.

Ramırez, M., and H. Geffner
Plan recognition as planning, *Proceedings of the International Conference on International Conference on Automated Planning and Scheduling*, AAAI Press, 2009, 19,1778–1783.

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

Rao, J., P. Kungas, and M. Matskin
Logic-based Web services composition: From service description to process model, *Proceedings. IEEE International Conference on Web Services, 2004.*, July 2004, 446–453. doi:10.1109/ICWS.2004.1314769.

Ray-Chaudhuri, D., and C. Berge
Hypergraph Seminar: Ohio State University 1972, Springer-Verlag, 1972.

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.

Riabov, A., S. Sohrabi, and O. Udrea
New algorithms for the top-k planning problem, *Proceedings of the Scheduling and Planning Applications woRKshop (SPARK) at the 24th International Conference on Automated Planning and Scheduling (ICAPS)*, 2014, 10–16.

Richter, S., and M. Westphal
The LAMA planner: Guiding cost-based anytime planning with landmarks, *Journal of Artificial Intelligence Research*, 39 (1), 127–177, 2010.

Roy, P. C., A. Bouzouane, S. Giroux, and B. Bouchard
Possibilistic activity recognition in smart homes for cognitively impaired people, *Applied Artificial Intelligence*, 25 (10), 883–926, 2011.

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

Sapena, O., E. Onaindıa, and A. Torreno
Combining heuristics to accelerate forward partial-order planning, *CSTPS*, 25, 2014.

Say, B., A. A. Cire, and J. C. Beck
Mathematical programming models for optimizing partial-order plan flexibility, *22nd European Conference of Artificial Intelligence*, 2016.

Schwarzentruber, F.
Hintikka's World: Agents with Higher-order Knowledge, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, Stockholm, Sweden: International Joint Conferences on Artificial Intelligence Organization, July 2018, 5859–5861. doi:10.24963/ijcai.2018/862.

Sebastia, L., E. Onaindia, and E. Marzal
A Graph-based Approach for POCL Planning, *ECAI*, 2000, 531–535.

Shekhar, S., and D. Khemani
Learning and Tuning Meta-heuristics in Plan Space Planning, *arXiv preprint arXiv:1601.07483*, 2016.

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

Sjöberg, J., and H. Nissar
Automated scheduling: Performance in different scenarios, 2015.

Smith, D. E., J. Frank, and W. Cushing
The ANML language, *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 2008.

Sohrabi, S., A. V. Riabov, and O. Udrea
Plan Recognition as Planning Revisited, *Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 25, 2016.

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.

Stentz, A., and others
The Focussed D* Algorithm for Real-Time Replanning., *IJCAI*, 1995, 95,1652–1659.

Sugawara, T.
Reusing Past Plans in Distributed Planning., *ICMAS*, 1995, 360–367.

Takesaki, M.
Theory of operator algebras II, vol. 125Springer Science & Business Media, 2013.

Talamadupula, K., S. Kambhampati, P. Schermerhorn, J. Benton, and M. Scheutz
Planning for human-robot teaming, *ICAPS 2011 Workshop on Scheduling and Planning Applications (SPARK)*, Vol. 67, 2011.

Tan, X., and M. Gruninger
The Complexity of Partial-Order Plan Viability Problems., *ICAPS*, 2014.

Thiébaux, S., J. Hoffmann, and B. Nebel
In defense of PDDL axioms, *Artificial Intelligence*, 168 (1-2), 38–69, 2005.

To, S. T., M. Roberts, T. Apker, B. Johnson, and D. W. Aha
Mixed Propositional Metric Temporal Logic: A New Formalism for Temporal Planning., *AAAI Workshop: Planning for Hybrid Systems*, 2016.

Tolksdorf, R., L. Nixon, F. Liebsch, D. Minh Nguyen, and E. Paslaru Bontas
Semantic web spaces, 2004.

Toro, C., C. Sanín, E. Szczerbicki, and J. Posada
Reflexive Ontologies: Enhancing Ontologies with Self-Contained Queries, *Cybernetics and Systems*, 39 (2), 171–189, February 2008. doi:10.1080/01969720701853467.

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

Unicode Consortium
Unicode Character Database, *About the Unicode Character Database*, June 2018b.

Van Der Krogt, R., and M. De Weerdt
Plan Repair as an Extension of Planning., *ICAPS*, 2005, 5,161–170.

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

Venn, J.
I. On the diagrammatic and mechanical representation of propositions and reasonings, *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 10 (59), 1–18, 1880.

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

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

Vidal, N., P. Taillibert, and S. Aknine
Online behavior recognition: A new grammar model linking measurements and intents, *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, IEEE, 2010, 2,129–137.

W3C
Examples for OWL. Eds. P. F. Patel-Schneider and I. Horrocks 2004a.

W3C
RDF Semantics, W3C, 2004b.

W3C
RDF Vocabulary Description Language 1.0: RDF Schema February 2004c.

W3C
OWL 2 Web Ontology Language Document Overview (Second Edition) December 2012.

W3C
RDF 1.1 Turtle: Terse RDF Triple Language January 2014.

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.

Younes, H. akan L., and R. G. Simmons
VHPOP : Versatile heuristic partial order planner, *JAIR*, 405–430, 2003.

Young, R. M., and J. D. Moore
DPOCL: A principled approach to discourse planning, *Proceedings of the Seventh International Workshop on Natural Language Generation*, Association for Computational Linguistics, 1994, 13–20.

Zhuo, H. H., and S. Kambhampati
Model-lite planning: Case-based vs. Model-based approaches, *Artificial Intelligence*, 246, 1–21, 2017.