

République Algérienne Démocratique et Populaire  
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique



Université des Sciences et de la Technologie Houari Boumédiène

Faculté d'Informatique  
Département IA and SD

Master Systèmes Informatiques intelligents

Module : RCR

---

## Rapport de TP

---

Réalisé par :

BENCHIKH Mehdi, 212131079203

BESSAI Tayeb, 212131044836

Année universitaire : 2024 / 2025

# Table des matières

<b>1</b>	<b>TP1 : SAT</b>	<b>5</b>
1.1	Introduction . . . . .	5
1.2	Étape 1 : Création du répertoire . . . . .	5
1.3	Étape 2 : Exécution du solveur SAT . . . . .	5
1.3.1	Le fichier Test1.cnf . . . . .	6
1.3.2	Le fichier Test2.cnf . . . . .	7
1.4	Étape 3 . . . . .	8
1.4.1	Traduction de la base de connaissances . . . . .	8
1.4.2	Exécution . . . . .	9
1.5	Étape 4 . . . . .	10
1.5.1	Fichier 1 : uf200-015 . . . . .	10
1.5.2	Fichier 2 : uuf200-079 . . . . .	11
1.6	Étape 5 . . . . .	12
<b>2</b>	<b>TP2 – Logique des prédicats avec Tweety</b>	<b>15</b>
2.1	Introduction générale . . . . .	15
2.1.1	TweetyProject . . . . .	15
2.2	Base de croyances et étapes de modélisation . . . . .	15
2.3	Exemple Personnalisé : Véhicules et Vitesse . . . . .	16
2.4	Ajout de formules à la base de croyances . . . . .	17
2.4.1	Composants clés pour la modélisation logique . . . . .	17
2.4.2	Exemples de formules ajoutées . . . . .	18
2.4.3	Explication des formules . . . . .	19
2.5	Vérification de la vérité des formules . . . . .	19
2.5.1	Composants utilisés pour l'inférence . . . . .	19
2.5.2	Exemple d'inférence avec <code>VehicleFolExample</code> . . . . .	19
2.5.3	Résultat obtenu (exécution Java) . . . . .	20
2.5.4	Étapes réalisées . . . . .	23
<b>3</b>	<b>TP3 : Logique Modale</b>	<b>25</b>
3.1	Introduction à la Logique Modale . . . . .	25
3.2	Choix de Modélisation : Pourquoi Python ? . . . . .	25
3.3	Description de la Modélisation Réalisée . . . . .	26
3.4	Exécution du Projet et Exemples . . . . .	27
3.4.1	Menu Principal . . . . .	27
3.4.2	Choix du Scénario 3 : Logique Déontique . . . . .	27
3.4.3	Exécution et Résultats pour un Monde Spécifique (Scénario 3) . . . . .	28

<b>4</b>	<b>Partie 4 – Logique des défauts</b>	<b>29</b>
<b>5</b>	<b>TP5 : Les Réseaux Sémantiques</b>	<b>35</b>
5.1	Partie 1 : Implémenter l'algorithme de propagation de marqueurs dans les réseaux sémantiques . . . . .	35
5.1.1	Code . . . . .	36
5.1.2	Result . . . . .	37
5.2	Partie 2 : Implémenter l'algorithme d'héritage . . . . .	38
5.2.1	Code . . . . .	38
5.2.2	Result . . . . .	39
5.3	Partie 3 : Implémenter l'algorithme d'exception . . . . .	40
5.3.1	Code . . . . .	40
5.3.2	Result . . . . .	40
<b>6</b>	<b>TP6 Langage de Description</b>	<b>41</b>
6.1	Choix de l'outil . . . . .	41
6.2	Fonctionnement de Owlready2 . . . . .	41
6.3	Composants de la TBox en logique des descriptions . . . . .	42
6.4	Exemple de code avec Owlready2 . . . . .	42
6.5	Rôle des fichiers .owl . . . . .	43
6.6	Affichage d'une ontologie OWL . . . . .	44
6.7	Modélisation - Exercice 3 . . . . .	44
6.8	Code Python Owlready2 . . . . .	45
6.9	Capture d'écran du fichier owl . . . . .	46

# Chapitre 1

## TP1 : SAT

### 1.1 Introduction

L'inférence logique est un processus permettant de déduire de nouvelles informations à partir d'une base de connaissances existante. Les **solveurs SAT** (Satisfiability) jouent un rôle crucial dans ce domaine, permettant de déterminer la satisfiabilité des formules logiques. Dans ce rapport, nous utiliserons le solveur SAT UBSCAT pour démontrer l'inférence logique à travers plusieurs exemples.

### 1.2 Étape 1 : Création du répertoire

On commence par créer un dossier contenant les deux fichiers de test sous format CNF ainsi que le solveur UBSCAT.

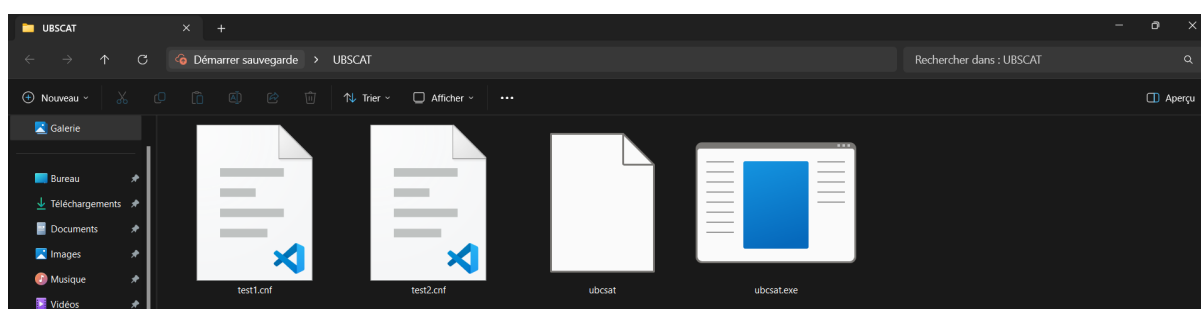


FIGURE 1.1 – répertoire du TP

### 1.3 Étape 2 : Exécution du solveur SAT

Dans l'étape 2, nous exécutons le solveur SAT et testons la satisfiabilité des deux fichiers : **test1.cnf** et **test2.cnf**.

### 1.3.1 Le fichier Test1.cnf

- Variables : 5
- Clauses : 9
- Commande : `ubcsat -alg saps -i test1.cnf -solve`

```
# Output Columns: |run|found|best|beststep|steps|
#
# run: Run Number
# found: Target Solution Quality Found? (1 => yes)
# best: Best (Lowest) # of False Clauses Found
# beststep: Step of Best (Lowest) # of False Clauses Found
# steps: Total Number of Search Steps
#
#      F  Best      Step      Total
#      Run N Sol'n    of      Search
#      No. D Found    Best    Steps
#
#      1 1      0        3        3
#
# Solution found for -target 0

-1 2 -3 -4 5

Variables = 5
Clauses = 9
TotalLiterals = 23
TotalCPUtimeElapsed = 0.000
FlipsPerSecond = 1
RunsExecuted = 1
SuccessfulRuns = 1
PercentSuccess = 100.00
Steps_Mean = 3
Steps_CoeffVariance = 0
Steps_Median = 3
CPUtime_Mean = 0
CPUtime_CoeffVariance = 0
CPUtime_Median = 0
```

FIGURE 1.2 – résultat fichier test1

La base **test1.cnf** est donc satisfiable. La solution trouvée est :

- $x_1 = 0$  (variable 1 est fausse)
- $x_2 = 1$  (variable 2 est vraie)
- $x_3 = 0$  (variable 3 est fausse)
- $x_4 = 0$  (variable 4 est fausse)
- $x_5 = 1$  (variable 5 est vraie)

### 1.3.2 Le fichier Test2.cnf

- Variables : 5
- Clauses : 12
- Commande : `ubcsat -alg saps -i test2.cnf -solve`

```
# Output Columns: |run|found|best|beststep|steps|
#
# run: Run Number
# found: Target Solution Quality Found? (1 => yes)
# best: Best (Lowest) # of False Clauses Found
# beststep: Step of Best (Lowest) # of False Clauses Found
# steps: Total Number of Search Steps
#
#      F  Best      Step      Total
#  Run N Sol'n      of      Search
#  No. D Found      Best      Steps
#
#      1 0      1      2      100000
# No Solution found for -target 0

Variables = 5
Clauses = 12
TotalLiterals = 28
TotalCPUTimeElapsed = 0.005
FlipsPerSecond = 19999542
RunsExecuted = 1
SuccessfulRuns = 0
PercentSuccess = 0.00
Steps_Mean = 100000
Steps_CoeffVariance = 0
Steps_Median = 100000
CPUTime_Mean = 0.00500011444092
CPUTime_CoeffVariance = 0
CPUTime_Median = 0.00500011444092
```

FIGURE 1.3 – résultat fichier test2

La base **test2.cnf** est donc insatisfiable.

## 1.4 Étape 3

### 1.4.1 Traduction de la base de connaissances

On suppose :

- Na, Nb, Nc ; Où Na = Nautilie de a
- Cea, Ceb, Cec ; Où Cea = Céphalopode de a
- Ma, Mb, Mc ; Où Ma = Mollusque de a
- Coa, Cob, Coc ; Où Coa = Coquille de a

Même chose en ce qui concerne b et c.

#### Les clauses sous Forme Normale Conjonctive :

- Les nautilies sont des céphalopodes :  $(\neg Na \vee Cea), (\neg Nb \vee Ceb), (\neg Nc \vee Cec)$
- Les céphalopodes sont des mollusques :  $(\neg Cea \vee Ma), (\neg Ceb \vee Mb), (\neg Cec \vee Mc)$
- Les mollusques ont une coquille :  $(\neg Ma \vee Coa), (\neg Mb \vee Cob), (\neg Mc \vee Coc)$
- Les céphalopodes n'en ont pas une coquille :  $(\neg Cea \vee \neg Coa), (\neg Ceb \vee \neg Cob), (\neg Cec \vee \neg Coc)$
- Les nautilies en ont une coquille :  $(\neg Na \vee Coa), (\neg Nb \vee Cob), (\neg Nc \vee Coc)$
- a est un nautilie : Na
- b est un céphalopode : Ceb
- c est un mollusque : Mc

#### Fichier CNF :

Le fichier CNF va être représenté par les informations suivantes :

- Na = 1, Nb = 2, Nc = 3
- Cea = 4, Ceb = 5, Cec = 6
- Coa = 7, Cob = 8, Coc = 9
- Ma = 10, Mb = 11, Mc = 12

```

zoo.cnf
1  p cnf 12 18
2  -1 4 0
3  -2 5 0
4  -3 6 0
5  -4 10 0
6  -5 11 0
7  -6 12 0
8  -10 7 0
9  -11 8 0
10 -12 9 0
11 -4 -7 0
12 -5 -8 0
13 -6 -9 0
14 -1 7 0
15 -2 8 0
16 -3 9 0
17  1 0
18  5 0
19 12 0

```

FIGURE 1.4 – contenu fichier zoo

### 1.4.2 Exécution

- Variables : 12
- Clauses : 18
- Commande : `ubcsat -alg saps -i zoo.cnf -solve`

```

# Output Columns: |run|found|best|beststep|steps|
#
# run: Run Number
# found: Target Solution Quality Found? (1 => yes)
# best: Best (Lowest) # of False Clauses Found
# beststep: Step of Best (Lowest) # of False Clauses Found
# steps: Total Number of Search Steps
#
#      F  Best      Step      Total
#      Run N Sol'n    of      Search
#      No. D Found    Best    Steps
#
#      1 0      2      6      100000
# No Solution found for -target 0

Variables = 12
Clauses = 18
TotalLiterals = 33
TotalCPUtimeElapsed = 0.006
FlipsPerSecond = 16666550
RunsExecuted = 1
SuccessfulRuns = 0
PercentSuccess = 0.00
Steps_Mean = 100000
Steps_CoeffVariance = 0
Steps_Median = 100000
CPUtime_Mean = 0.00600004196167
CPUtime_CoeffVariance = 0
CPUtime_Median = 0.00600004196167

```

FIGURE 1.5 – résultat fichier zoo

La base **zoo.cnf** est donc insatisfiable.



## 1.5 Étape 4

Nous avons téléchargé deux fichiers : un satisfiable et l'autre non, et nous les avons testés en utilisant ubcsat.

### 1.5.1 Fichier 1 : uf200-015

- Variables : 200
- Clauses : 860
- Commande : `ubcsat -alg saps -i uf200-015.cnf -solve`

```
# Output Columns: |run|found|best|beststep|steps|
#
# run: Run Number
# found: Target Solution Quality Found? (1 => yes)
# best: Best (Lowest) # of False Clauses Found
# beststep: Step of Best (Lowest) # of False Clauses Found
# steps: Total Number of Search Steps
#
#      F  Best      Step      Total
#      Run N Sol'n    of      Search
#      No. D Found    Best    Steps
#
#      1 1      0      1925      1925
#
# Solution found for -target 0
-1 -2 -3 4 -5 6 -7 8 9 10
-11 -12 -13 14 15 -16 -17 -18 -19 20
-21 -22 -23 24 25 26 -27 -28 29 30
31 32 33 34 35 -36 -37 38 -39 -40
-41 -42 -43 44 -45 46 -47 -48 49 -50
-51 -52 53 54 55 -56 57 -58 59 60
61 62 63 64 -65 66 67 68 -69 -70
71 -72 73 74 -75 76 -77 78 -79 80
81 -82 -83 -84 85 86 87 88 -89 -90
91 -92 93 94 95 -96 97 98 -99 -100
101 102 103 -104 -105 106 -107 -108 109 110
111 112 113 114 115 -116 -117 -118 -119 120
121 -122 -123 -124 -125 126 -127 -128 129 -130
-131 -132 133 -134 -135 136 137 -138 139 -140
141 -142 -143 -144 -145 146 -147 -148 149 150
151 -152 153 154 -155 156 -157 -158 159 -160
-161 -162 -163 164 -165 -166 167 -168 169 170
171 -172 -173 -174 -175 176 177 -178 179 180
181 -182 -183 -184 185 186 -187 -188 -189 190
191 -192 193 194 -195 196 -197 198 -199 -200
```

FIGURE 1.6 – résultat fichier 1

La base **uf200-015** est donc satisfiable.

## 1.5.2 Fichier 2 : uuf200-079

- Variables : 200
- Clauses : 860
- Commande : `ubcsat -alg saps -i uuf200-079.cnf -solve`

```
# run: Run Number
# found: Target Solution Quality Found? (1 => yes)
# best: Best (Lowest) # of False Clauses Found
# beststep: Step of Best (Lowest) # of False Clauses Found
# steps: Total Number of Search Steps
#
#           F   Best       Step       Total
#   Run N Sol'n   of       Search
#   No. D Found   Best     Steps
#
#           1 0       1       38305       100000
# No Solution found for -target 0

Variables = 200
Clauses = 860
TotalLiterals = 2580
TotalCPUtimeElapsed = 0.024
FlipsPerSecond = 4166679
RunsExecuted = 1
SuccessfulRuns = 0
PercentSuccess = 0.00
Steps_Mean = 100000
Steps_CoeffVariance = 0
Steps_Median = 100000
CPUtime_Mean = 0.0239999294281
CPUtime_CoeffVariance = 0
CPUtime_Median = 0.0239999294281
```

FIGURE 1.7 – résultat fichier 2

La base **uuf200-079** est donc insatisfiable.

## 1.6 Étape 5

Vérifier si une formule  $\phi$  est une conséquence logique d'une base BC :

**Entrée :**

- BC : ensemble de clauses en forme CNF
- $\phi$  : littéral qu'on veut tester
- SAT\_SOLVER : programme exécutable (ex : ubcsat)

**Sortie :**

- Vrai si  $BC \models \phi$
- Faux sinon

**Algorithme :**

1. Traduire  $\phi$  en forme CNF (si ce n'est pas déjà fait)
2. Négocier  $\phi \rightarrow \neg\phi$  (aussi en CNF)
3. Créer un nouveau fichier CNF :
  - Inclure toutes les clauses de BC
  - Ajouter les clauses de  $\neg\phi$
4. Enregistrer ce fichier temporaire, par exemple : `test_inference.cnf`
5. Appeler le solveur SAT :
  - exécution : `ubcsat -alg saps -i test_inference.cnf -solve`
6. Lire le résultat :
  - Si aucune solution satisfaisante trouvée  $\rightarrow$  retourner Vrai
  - Sinon  $\rightarrow$  retourner Faux

### Implémentation en Python :

```
def check_inference(file, literal, ubcsat_path="./ubcsat"):
    """
    Vérifie si le fichier CNF d'origine + clauses supplémentaires est insatisfiable.
    """
    bc_clauses, nb_vars = read_file(file)

    if bc_clauses: # Vérifier si le fichier a été lu correctement

        # Ajouter l'inverse du literal  $\phi$ 
        all_clauses = bc_clauses +  $[-1 * literal]$ 

        # Ajuster le nombre de variables si nécessaire
        max_var_utilisee = max(abs(lit) for clause in all_clauses for lit in clause)
        nb_vars = max(nb_vars, max_var_utilisee)

        # Écrire dans un fichier temporaire
        cnf_temp = create_temp_file(all_clauses, nb_vars)

        try:
            result = subprocess.run(
                [ubcsat_path, "-alg", "saps", "-i", cnf_temp, "-solve"],
                capture_output=True, text=True
            )

            output = result.stdout
            x = "{}{}{}{}".format(-1 * literal)
            if "Solution Found" not in output:
                print(f"\n✅ BC U {x} est insatisfiable, donc BC infère bien {literal} (BC  $\models$  {literal})")
                return True
            else:
                print(f"\n❌ BC U {x} est satisfiable, donc BC n'infère pas {literal} (BC  $\not\models$  {literal})")
                return False

        finally: # Nettoyer le fichier temporaire
            os.remove(cnf_temp)
```

FIGURE 1.8 – implémentation en Python

```
C:\Users\mehdi\OneDrive\Bureau\RCR Project\UBCSAT>python script.py
Entrer le fichier CNF : uf200-015.cnf
Entrer un literal  $\phi$  (ex: 3 pour c, -3 pour  $\neg c$ ) : 3
✅ BC U {-3} est insatisfiable, donc BC infère bien 3 (BC  $\models$  3)
```

FIGURE 1.9 – résultat de l'algorithme

# Chapitre 2

## TP2 – Logique des prédicats avec Tweety

### 2.1 Introduction générale

La logique du premier ordre (First Order Logic, ou FOL) est une extension de la logique propositionnelle qui permet de raisonner sur des objets, des propriétés et des relations. Dans ce TP, nous utilisons **TweetyProject**, une bibliothèque Java dédiée à la modélisation et au raisonnement en logique, pour représenter et manipuler des connaissances exprimées en FOL.

#### 2.1.1 TweetyProject

Tweety est un framework modulaire pour la logique symbolique, notamment la logique propositionnelle, la logique des prédicats, les logiques modales, ainsi que d'autres formes de raisonnement non monotone. Il fournit des composants permettant de :

- Définir des signatures logiques (sortes, constantes, prédicats),
- Parser des formules exprimées en FOL,
- Construire des bases de croyances (ensembles de formules),
- Appliquer des moteurs d'inférence (raisonneurs) pour interroger les connaissances.

### 2.2 Base de croyances et étapes de modélisation

Une **base de croyances** (ou `FolBeliefSet` en Tweety) est un ensemble de formules logiques supposées vraies, représentant les connaissances disponibles dans un domaine donné.

Voici les étapes clés typiques de création et d'exploitation d'une base de croyances en logique des prédicats :

1. **Création de la signature** : définition des sortes (par exemple `Animal`, `Plant`), des constantes (comme `anna`, `bob`, `emma`), et des prédicats (tels que `flies`).
2. **Formulation de règles** : ajout de formules exprimant des faits ou des règles générales, comme « tous les animaux peuvent voler », ou « Bob mange Emma » (relation alimentaire spécifique).
3. **Enregistrement et affichage** : la base de croyances est sauvegardée et affichée afin de vérifier que les connaissances ont bien été modélisées.

## 2.3 Exemple Personnalisé : Véhicules et Vitesse

Pour illustrer ce processus, nous avons conçu un exemple simplifié autour des véhicules. Ci-dessous un extrait de notre classe Java `VehicleFolExample`, dans laquelle on modélise deux véhicules (`car` et `bike`), ainsi que des relations telles que `Drives` :

```
public class Main {
    public static void main(String[] args) {
        // Activate logical equality
        FolSignature sig = new FolSignature(true);

        // Define a sort
        Sort vehicleSort = new Sort("Vehicle");
        sig.add(vehicleSort);

        // Define constants of that sort
        Constant car = new Constant("car", vehicleSort);
        Constant bike = new Constant("bike", vehicleSort);
        sig.add(car, bike);

        // Define unary predicate: Drives(Vehicle)
        List<Sort> oneArg = new ArrayList<>();
        oneArg.add(vehicleSort);
        Predicate drives = new Predicate("Drives", oneArg);

        // Define binary predicate: FasterThan(Vehicle, Vehicle)
        List<Sort> twoArgs = new ArrayList<>();
        twoArgs.add(vehicleSort);
        twoArgs.add(vehicleSort);
        Predicate fasterThan = new Predicate("FasterThan", twoArgs);

        // Add predicates to the signature
        sig.add(drives, fasterThan);

        // Display signature
        System.out.println("Signature des prédicats : " + sig);
    }
}
```

FIGURE 2.1 – exemple 1

À ce stade, nous avons défini :

- Le type logique **Vehicle**,
- Deux objets constants : **car** et **bike**,
- Deux relations : **Drives** (un véhicule est conduit), et **FasterThan** (relation de vitesse entre deux véhicules).

Les prochaines sections illustreront comment parser des formules logiques basées sur cette signature, les ajouter à une base de croyances, et interroger cette base avec un moteur d'inférence.

## 2.4 Ajout de formules à la base de croyances

Après avoir défini la signature logique, l'étape suivante consiste à construire une **FolBeliefSet**, c'est-à-dire un ensemble de formules représentant notre base de croyances. Ces formules sont analysées à partir de chaînes de caractères via le composant **FolParser**, en utilisant la signature définie précédemment.

### 2.4.1 Composants clés pour la modélisation logique

Voici les composants principaux utilisés pour construire des formules logiques :

- **FolSignature** : Crée une signature contenant les sortes, constantes et prédicats. Cela définit le vocabulaire logique utilisable.
- **Sort** et **Constant** : Définissent les entités logiques (ex. **Vehicle**, **car**, **bike**).
- **Predicate** : Représente des relations entre objets. Par exemple, **Drives(car)** signifie "quelqu'un conduit une voiture".
- **FolParser** : Sert à convertir des formules écrites sous forme textuelle en objets manipulables en Java. Il respecte la signature définie.
- **FolBeliefSet** : Conteneur logique qui regroupe les formules logiques (règles, faits) représentant les connaissances.

### 2.4.2 Exemples de formules ajoutées

Voici un extrait Java de notre classe `VehicleFolExample` montrant comment parser et ajouter des formules dans la base de croyances :

```
public class Main {
    public static void main(String[] args) throws Exception {
        // Création de la signature
        FolSignature sig = new FolSignature(true);
        Sort vehicleSort = new Sort("Vehicle");
        sig.add(vehicleSort);

        Constant car = new Constant("car", vehicleSort);
        Constant bike = new Constant("bike", vehicleSort);
        sig.add(car, bike);

        List<Sort> oneArg = Collections.singletonList(vehicleSort);
        Predicate drives = new Predicate("Drives", oneArg);

        List<Sort> twoArgs = Arrays.asList(vehicleSort, vehicleSort);
        Predicate fasterThan = new Predicate("FasterThan", twoArgs);

        sig.add(drives, fasterThan);

        // Création du parser avec la signature
        FolParser parser = new FolParser();
        parser.setSignature(sig); // Utilise la signature définie précédemment

        // Création de la base de croyances
        FolBeliefSet beliefBase = new FolBeliefSet();

        // Ajout de formules logiques exprimant des faits ou des règles
        FolFormula f1 = (FolFormula) parser.parseFormula("Drives(car)");
        FolFormula f2 = (FolFormula) parser.parseFormula("!Drives(bike)");
        FolFormula f3 = (FolFormula) parser.parseFormula("FasterThan(car, bike)");
        FolFormula f4 = (FolFormula) parser.parseFormula(
            "forall X: (Drives(X) => exists Y: FasterThan(X, Y))"
        );

        // Ajout à la base de croyances
        beliefBase.add(f1);
        beliefBase.add(f2);
        beliefBase.add(f3);
        beliefBase.add(f4);

        // Affichage des formules dans la base
        System.out.println(x:"Base de croyances FOL :");
        for (FolFormula formula : beliefBase) {
            System.out.println(" - " + formula);
        }
    }
}
```

FIGURE 2.2 – exemple 2



### 2.4.3 Explication des formules

- `Drives(car)` : le véhicule `car` est conduit.
- `!Drives(bike)` : le véhicule `bike` n'est pas conduit.
- `FasterThan(car, bike)` : la voiture est plus rapide que le vélo.
- `forall X: (Drives(X) => exists Y: FasterThan(X,Y))` : tout véhicule conduit est plus rapide qu'au moins un autre véhicule.

Ces formules enrichissent la base de croyances qui peut ensuite être utilisée pour des inférences logiques à l'aide d'un raisonneur, ce qui sera présenté dans la section suivante.

## 2.5 Vérification de la vérité des formules

Une fois la base de croyances construite, nous pouvons interroger cette base pour vérifier la validité logique de certaines formules. Cela se fait à l'aide d'un raisonneur logique, tel que `SimpleFolReasoner`.

### 2.5.1 Composants utilisés pour l'inférence

- **FolReasoner** : C'est une interface utilisée pour raisonner sur une base de croyances en logique du premier ordre. On peut y enregistrer un raisonneur par défaut.
- **SimpleFolReasoner** : Implémentation simple d'un raisonneur basé sur une énumération naïve. Utile pour des tests de véracité.
- **query()** : Méthode permettant d'interroger la base à propos d'une formule. Elle retourne `true` ou `false` selon que la formule est une conséquence logique ou non.

### 2.5.2 Exemple d'inférence avec `VehicleFolExample`

Voici un extrait de code Java qui vérifie plusieurs formules logiques à partir de la base de croyances définie précédemment :

```

// Création de la signature
FolSignature sig = new FolSignature(true);
Sort vehicleSort = new Sort("Vehicule");
sig.add(vehicleSort);

Constant car = new Constant("car", vehicleSort);
Constant bike = new Constant("bike", vehicleSort);
sig.add(car, bike);

List<Sort> oneArg = Collections.singletonList(vehicleSort);
Predicate drives = new Predicate("Drives", oneArg);

List<Sort> twoArgs = Arrays.asList(vehicleSort, vehicleSort);
Predicate fasterThan = new Predicate("FasterThan", twoArgs);

sig.add(drives, fasterThan);

// Création du parser
FolParser parser = new FolParser();
parser.setSignature(sig);

// Création de la base de croyances
FolBeliefSet beliefBase = new FolBeliefSet();
beliefBase.add((FolFormula) parser.parseFormula("Drives(car)"));
beliefBase.add((FolFormula) parser.parseFormula("!Drives(bike)"));
beliefBase.add((FolFormula) parser.parseFormula("FasterThan(car, bike)"));
beliefBase.add((FolFormula) parser.parseFormula("forall X: (Drives(X) => exists Y: FasterThan(X, Y))"));

// Création du moteur d'inférence
FolReasoner reasoner = new NaiveReasoner(); // ou un autre moteur compatible

// Formule 1 : Le véhicule "bike" est conduit ?
System.out.println("Vérification 1 : " +
    reasoner.query(beliefBase, (FolFormula) parser.parseFormula("Drives(bike)")));

// Formule 2 : Le véhicule "car" est plus rapide que "bike" ?
System.out.println("Vérification 2 : " +
    reasoner.query(beliefBase, (FolFormula) parser.parseFormula("FasterThan(car, bike)")));

// Formule 3 : Tous les véhicules conduits sont plus rapides que d'autres ?
System.out.println("Vérification 3 : " +
    reasoner.query(beliefBase, (FolFormula) parser.parseFormula(
        "forall X: (Drives(X) => exists Y: FasterThan(X, Y))"
    )));

```

FIGURE 2.3 – exemple 3

### 2.5.3 Résultat obtenu (exécution Java)

Les résultats retournés par le raisonneur s'affichent directement sur la console. Voici une capture d'écran de l'exécution de ce code :

Chaque ligne correspond à une requête logique dont le résultat est soit `true`, soit `false`, selon que la formule est déductible de la base de croyances.

```

public class Main {
    public static void main(String[] args) throws Exception {

        // Formules exemples
        FolFormula f1 = (FolFormula) parser.parseFormula("Drives(car)");
        FolFormula f2 = (FolFormula) parser.parseFormula("Drives(bike)");
        FolFormula f3 = (FolFormula) parser.parseFormula("FasterThan(car, bike)");
        FolFormula f4 = (FolFormula) parser.parseFormula("car == car");
        FolFormula f5 = (FolFormula) parser.parseFormula("bike != car");

        beliefSet.add(f1, f2, f3, f4, f5);

        System.out.println("\nBase de croyances parsée : " + beliefSet);

        // Utilisation d'un raisonneur simple pour interroger la base
        FolReasoner.setDefaultReasoner(new SimpleFolReasoner());
        FolReasoner prover = FolReasoner.getDefaultReasoner();

        System.out.println("Réponse 1 : " + prover.query(beliefSet, (FolFormula) parser.parseFormula("Drives(bike)")));
        System.out.println("Réponse 2 : " + prover.query(beliefSet, (FolFormula) parser.parseFormula("forall X: (exists Y: FasterThan(X, Y))")));
        System.out.println("Réponse 3 : " + prover.query(beliefSet, (FolFormula) parser.parseFormula("bike == bike")));
        System.out.println("Réponse 4 : " + prover.query(beliefSet, (FolFormula) parser.parseFormula("bike != bike")));
        System.out.println("Réponse 5 : " + prover.query(beliefSet, (FolFormula) parser.parseFormula("car != bike")));

    }
}

```

FIGURE 2.4 – résultat exécution

## Interaction avec l'utilisateur

Dans cette dernière étape, nous avons intégré une interface interactive via le terminal permettant à l'utilisateur de saisir dynamiquement une formule en logique du premier ordre (FOL). Le système utilise alors le *parser* et le *reasoner* de Tweety pour déterminer si cette formule peut être déduite de la base de croyances actuelle.

## Principe de fonctionnement

- L'utilisateur saisit une formule au format FOL, par exemple `Drives(car)`.
- Cette formule est analysée à l'aide de la classe `FolParser` de Tweety.
- Ensuite, le `FolReasoner` évalue la véracité de cette formule par rapport à la base de croyances existante.
- Le résultat retourné est un booléen indiquant si la formule est inférable (`true`) ou non (`false`).

```

Base de croyances parsée ( (carcar), Drives (bike), Faster Than (car,bike), (bike/==car), Drives (car)
Réponse 1: false
Réponse 2: false
Réponse 3: true
Réponse 4: false
Réponse 5: true

```

FIGURE 2.5 – output

## Exemple d'exécution

Voici un exemple d'interaction typique avec l'utilisateur :

```
public class Main {
    public static void main(String[] args) {

        // Requête dynamique par l'utilisateur
        Scanner scanner = new Scanner(System.in);
        System.out.println(x:"\n Entrée utilisateur -");
        System.out.print(s:"Entrez une formule FOL (ex: Drives (car)) ");
        String userInput = scanner.nextLine();
        try {
            FolFormula userFormula = (FolFormula) parser.parseFormula (userInput);
            Boolean userResult = prover.query (beliefset, userFormula);
            System.out.println("Formule: " + userInput + " --> Résultat: " + userResult);
        } catch (Exception e) {
            System.out.println("Erreur lors du parsing ou de la requête : " + e.getMessage());
        }
        scanner.close();
    }
}
```

FIGURE 2.6 – exemple 4

Voici une capture d'écran de l'exécution du programme dans le terminal :

```
Base de croyances: { (car==car), !Drives (bike), FasterThan (car,bike), (bike/=car), Drives (car))

***** Résultats des requêtes prédéfinies *****
Formule Drives (bike) --> Résultat: false
Formule forall X: (exists Y: (Drives (X) && FasterThan (X, Y))) --> Résultat: false
Formule: bike bike --> Résultat: true
Formule bike/bike--> Résultat: false
Formule: car / bike--> Résultat: true

***** Entrée utilisateur *****
Entrez une formule FOL (ex: Drives (car)) : FasterThan (car, bike)
Formule Faster Than (car, bike) --> Résultat true
```

FIGURE 2.7 – exécution exemple 4

## Exploration via Modal Logic Playground

L'outil **Modal Logic Playground** est une plateforme interactive permettant de visualiser des modèles et de tester des formules modales sur ces modèles. Elle est particulièrement adaptée pour explorer la logique modale temporelle.

Lien : <https://modal-logic.github.io/Playground/>

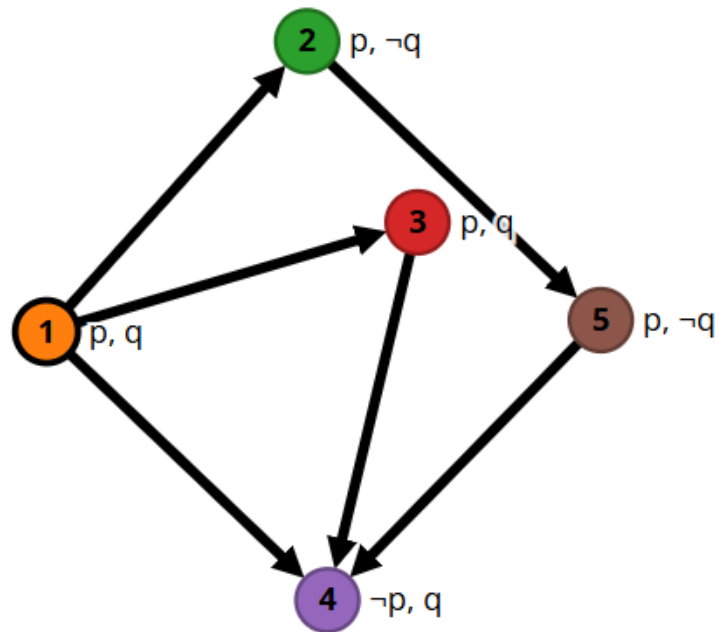


FIGURE 2.8 – Modèle logique

### 2.5.4 Étapes réalisées

1. **Définition du modèle** : Nous avons construit un modèle temporel avec différents états (ou mondes) liés entre eux par des transitions. Chaque état contient un ensemble de propositions atomiques vraies à ce moment.
2. **Formulation des propriétés** : Nous avons écrit plusieurs formules en logique temporelle (utilisant  $G, F$ , etc.) pour vérifier certaines propriétés du système.
3. **Évaluation des formules** : L'outil a permis de tester la validité de ces formules dans différents états du modèle.

### Formules temporelles testées

Nous avons testé deux formules en logique modale temporelle dans le modèle défini :

**Formule 1** :

$$\Box(\Diamond p \vee \neg q)$$

**Interprétation** : Dans tous les mondes accessibles, il est toujours vrai que soit  $p$  sera éventuellement vrai, soit  $q$  est faux.

**Résultat** : Cette formule est **vraie** dans les mondes :  $w_2, w_4$

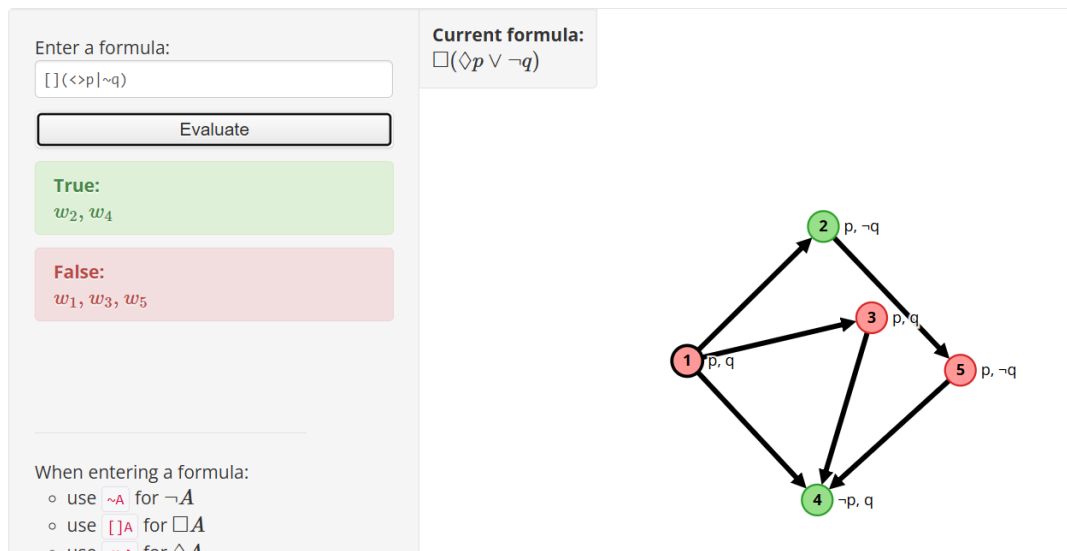


FIGURE 2.9 – test formule 1

**Formule 2 :**

$$\Box(q \wedge \Diamond \neg p)$$

**Interprétation :** Dans tous les mondes accessibles,  $q$  est toujours vrai et il y aura un monde accessible où  $p$  est faux.

**Résultat :** Cette formule est **vraie** uniquement dans le monde :  $w_4$

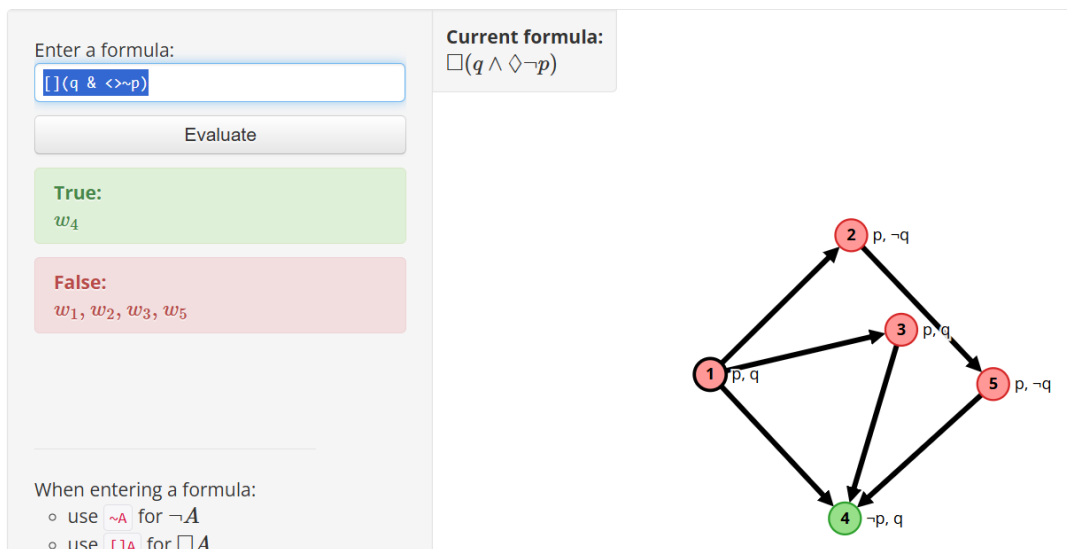


FIGURE 2.10 – test formule 2

# Chapitre 3

## TP3 : Logique Modale

### 3.1 Introduction à la Logique Modale

La logique modale est une extension de la logique classique qui permet de raisonner sur des concepts tels que la nécessité, la possibilité, la connaissance, le temps, ou l'obligation. Contrairement à la logique propositionnelle qui traite des propositions simplement vraies ou fausses, la logique modale introduit des opérateurs modaux qui modifient la signification des propositions. Les opérateurs les plus connus sont  $\Box$  (box, pour la nécessité ou "il est nécessaire que") et  $\Diamond$  (diamond, pour la possibilité ou "il est possible que").

Ces opérateurs sont interprétés dans le cadre de la sémantique des mondes possibles, également connue sous le nom de sémantique de Kripke. Un modèle de Kripke est un triplet  $(W, R, V)$ , où :

- $W$  est un ensemble de "mondes possibles" (ou états, instants, situations).
- $R$  est une "relation d'accessibilité" entre ces mondes, indiquant quels mondes sont "visibles" ou "atteignables" depuis un autre monde. Cette relation peut varier en fonction du type de logique modale (par exemple, réflexive, transitive, euclidienne).
- $V$  est une "fonction de valuation" qui assigne une valeur de vérité (vrai ou faux) à chaque proposition atomique dans chaque monde.

Dans ce cadre, la formule  $\Box\phi$  est vraie dans un monde  $w$  si  $\phi$  est vraie dans tous les mondes accessibles depuis  $w$ . De même,  $\Diamond\phi$  est vraie dans un monde  $w$  si  $\phi$  est vraie dans au moins un monde accessible depuis  $w$ .

### 3.2 Choix de Modélisation : Pourquoi Python ?

Initialement, la modélisation de systèmes logiques complexes peut s'avérer ardue avec des outils ou des approches spécifiques. Par exemple, des problèmes comme le "problème de Tweety" en logique non monotone, qui traite de l'inférence par défaut (les oiseaux volent, mais Tweety est un pingouin et ne vole pas), illustrent la difficulté de capturer

des raisonnements nuancés avec des formalismes rigides. La complexité de représenter de telles exceptions et des connaissances évolutives dans un cadre logique purement formel peut être considérable.

Face à ces défis, nous avons choisi Python comme langage de modélisation. Python offre une flexibilité et une richesse de bibliothèques (comme `networkx` pour la manipulation de graphes et `sympy` pour le calcul symbolique) qui facilitent grandement la construction et l'expérimentation avec des modèles de Kripke. Sa syntaxe claire et sa capacité à gérer des structures de données complexes permettent une implémentation intuitive des mondes, des relations d'accessibilité et des fonctions de valuation, rendant l'exploration de la logique modale plus accessible et interactive.

### 3.3 Description de la Modélisation Réalisée

Le projet Python développé vise à illustrer des exemples concrets de logique modale à travers trois scénarios principaux : la logique épistémique (connaissance), la logique temporelle (possibilités futures) et la logique déontique (obligation/perméssibilité).

Le cœur de la modélisation repose sur les éléments suivants :

- **Modèle de Kripke** : Pour chaque scénario, un modèle de Kripke spécifique est défini avec son propre ensemble de mondes ( $W$ ), sa relation d'accessibilité ( $R$ ) et sa fonction de valuation ( $V$ ).
- **Graphes avec `networkx`** : La relation d'accessibilité  $R$  est représentée comme un graphe orienté à l'aide de la bibliothèque `networkx`. Cela permet de naviguer facilement entre les mondes accessibles.
- **Opérateurs Modaux** : Deux fonctions principales, `box` et `dia`, implémentent les opérateurs  $\Box$  et  $\Diamond$ . Ces fonctions prennent en entrée une formule (représentée par une fonction lambda) et un monde, puis évaluent la vérité de la formule modale en parcourant les mondes accessibles.
- **Variables Propositionnelles et Logique avec `sympy`** : La bibliothèque `sympy` est utilisée pour définir les variables propositionnelles (par exemple, `R`, `S` pour le scénario épistémique) et pour construire des formules logiques complexes à l'aide d'opérateurs comme `Not` ( $\neg$ ), `Or` ( $\vee$ ), `Implies` ( $\rightarrow$ ), et `And` ( $\wedge$ ). Cela permet une évaluation symbolique et une manipulation aisée des expressions logiques.

Chaque scénario est encapsulé dans une fonction Python dédiée, ce qui permet de maintenir une structure modulaire et de faciliter l'ajout de nouveaux exemples. Un menu principal interactif guide l'utilisateur à travers les différents scénarios.

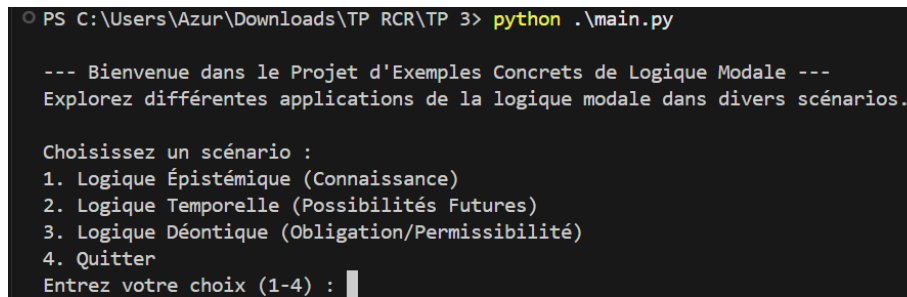


## 3.4 Exécution du Projet et Exemples

Le projet s'exécute via un menu interactif en ligne de commande. L'utilisateur est invité à choisir l'un des scénarios proposés ou à quitter l'application.

### 3.4.1 Menu Principal

Voici un aperçu du menu principal tel qu'il apparaît lors de l'exécution du script :



```
PS C:\Users\Azur\Downloads\TP_RCR\TP 3> python .\main.py

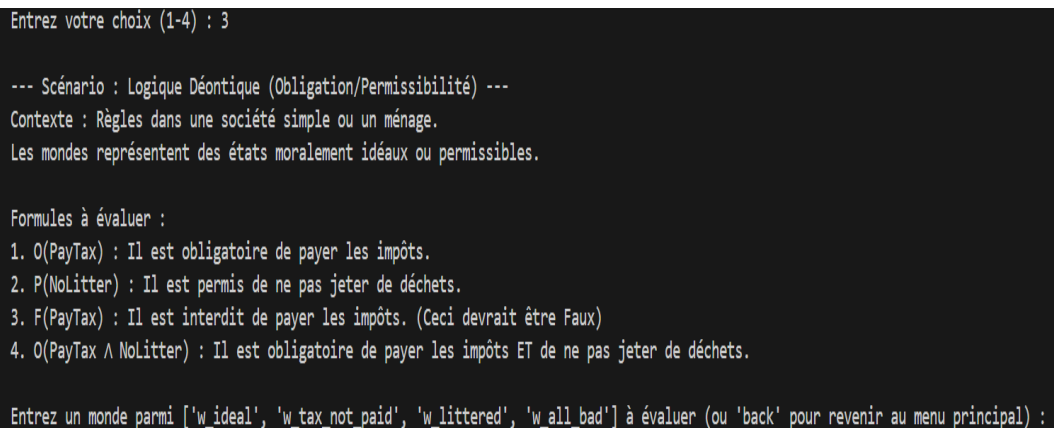
--- Bienvenue dans le Projet d'Exemples Concrets de Logique Modale ---
Explorez différentes applications de la logique modale dans divers scénarios.

Choisissez un scénario :
1. Logique Épistémique (Connaissance)
2. Logique Temporelle (Possibilités Futures)
3. Logique Déontique (Obligation/Permissibilité)
4. Quitter
Entrez votre choix (1-4) : █
```

FIGURE 3.1 – menu principal du projet.

### 3.4.2 Choix du Scénario 3 : Logique Déontique

Pour illustrer l'utilisation, nous choisissons le scénario 3, "Logique Déontique (Obligation/Permissibilité)". Après avoir entré "3", le programme affiche le contexte de ce scénario et les mondes disponibles.



```
Entrez votre choix (1-4) : 3

--- Scénario : Logique Déontique (Obligation/Permissibilité) ---
Contexte : Règles dans une société simple ou un ménage.
Les mondes représentent des états moralement idéaux ou permissibles.

Formules à évaluer :
1. O(PayTax) : Il est obligatoire de payer les impôts.
2. P(NoLitter) : Il est permis de ne pas jeter de déchets.
3. F(PayTax) : Il est interdit de payer les impôts. (Ceci devrait être Faux)
4. O(PayTax ∧ NoLitter) : Il est obligatoire de payer les impôts ET de ne pas jeter de déchets.

Entrez un monde parmi ['w_ideal', 'w_tax_not_paid', 'w_littered', 'w_all_bad'] à évaluer (ou 'back' pour revenir au menu principal) : █
```

FIGURE 3.2 – choix du scénario 3 et de son introduction.

### 3.4.3 Exécution et Résultats pour un Monde Spécifique (Scénario 3)

Une fois dans le scénario 3, l'utilisateur est invité à entrer un monde pour évaluer les formules. Choisissons par exemple le monde "w\_tax\_not\_paid" ("Impôts non payés"). Le programme affichera alors la valuation des propositions atomiques dans ce monde et les résultats de l'évaluation des formules déontiques clés.

```
Entrez un monde parmi ['w_ideal', 'w_tax_not_paid', 'w_littered', 'w_all_bad'] à évaluer (ou 'back' pour revenir au menu principal) : w_tax_not_paid

--- Évaluation dans le Monde : w_tax_not_paid ---
Impôts payés ? (PayTax): False
Pas de déchets ? (NoLitter): True
Il est obligatoire de payer les impôts (O(PayTax)): True
Il est permis de ne pas jeter de déchets (P(NoLitter)): True
Il est interdit de payer les impôts (F(PayTax)): False
Il est obligatoire de payer les impôts ET de ne pas jeter de déchets (O(PayTax ∧ NoLitter)): True
```

FIGURE 3.3 – évaluation des formules dans le monde "w\_tax\_not\_paid" du scénario 3.

Ces captures d'écran représentent le flux d'interaction avec le programme, montrant comment l'utilisateur peut explorer les différentes applications de la logique modale et observer les résultats des évaluations de formules dans des contextes variés.

# Chapitre 4

## Partie 4 – Logique des défauts

### Introduction

La logique des défauts permet de raisonner dans un contexte où l'information est incomplète ou partiellement fiable. Elle s'appuie sur des règles par défaut qui s'appliquent tant qu'aucune preuve ne vient les contredire. Cette approche est utile dans de nombreuses situations du raisonnement non-monotone, où les conclusions peuvent changer en fonction de l'arrivée de nouvelles données.

Dans ce chapitre, nous allons explorer les principes de la logique des défauts à travers des exercices concrets. Nous utiliserons un outil spécialisé qui simule des modèles par défaut pour observer le comportement des systèmes en fonction de différentes hypothèses.

#### **Outil utilisé :** `DefaultLogicModelCheck`

Cet outil est disponible sur le dépôt GitHub suivant :

<https://github.com/edm92/defaultlogic>

**Objectif :** Tester et vérifier le fonctionnement de modèles à défauts sur les exercices 1, 2, 3 et 6. Nous avons volontairement écarté les exercices plus avancés, car l'outil utilisé ne prend pas en charge :

- la gestion des priorités entre défauts,
- les prédicats complexes ou paramétrés.

Pour chaque exercice, nous allons :

- décrire le monde et les règles par défaut proposées,
- tester les différents états possibles avec et sans défaut,
- analyser les cas où les règles s'appliquent ou sont bloquées.

## Implémentation des composants principaux en Java

Avant de présenter les exercices, nous décrivons les éléments fondamentaux de notre simulateur Java pour le raisonnement par défaut. L'approche est 100% manuelle et ne repose sur aucune bibliothèque externe comme Tweety.

```
public class Monde {
    private Set < String > faits ;

    public Monde ( Set < String > faits ) {
        this.faits = new HashSet < >( faits ) ;
    }

    public boolean contient ( String formule ) {
        return faits.contains ( formule ) ;
    }

    public void ajouter ( String formule ) {
        faits.add ( formule ) ;
    }

    public Set < String > getFaits () {
        return faits ;
    }

    @Override
    public String toString () {
        return faits.toString () ;
    }
}
```

FIGURE 4.1 – classe Monde

1. **Classe Monde** : représente un ensemble de formules initiales  $W$ .
2. **Classe Default** : modélise une règle par défaut  $\frac{\alpha:\beta}{\gamma}$ .
3. **Méthode computeExtensions** : applique récursivement les défauts applicables.

```

public class Default {
    private String prerequis ;
    private String justification ;
    private String conclusion ;

    public Default ( String prerequis , String justification , String conclusion ) {
        this.prerequis = prerequis ;
        this.justification = justification ;
        this.conclusion = conclusion ;
    }

    public String getPrerequis () { return prerequis ; }
    public String getJustification () { return justification ; }
    public String getConclusion () { return conclusion ; }
}

```

FIGURE 4.2 – classe Default

```

public class TheorieParDefault {
    private List<Default> defaults;
    private Set<String> mondeInitial;

    public TheorieParDefault(List<Default> defaults, Set<String> mondeInitial) {
        this.defaults = defaults;
        this.mondeInitial = new HashSet<>(mondeInitial);
    }

    public void executerTheorie() {
        Set<String> extension = new HashSet<>(mondeInitial);
        boolean changed;

        do {
            changed = false;
            for (Default d : defaults) {
                boolean applicable = (d.getPrerequis().isEmpty() || extension.contains(d.getPrerequis()))
                    && !extension.contains(negate(d.getJustification()));

                if (applicable && !extension.contains(d.getConclusion())) {
                    extension.add(d.getConclusion());
                    changed = true;
                }
            }
        } while (changed);

        System.out.println("Extension trouvée : " + extension);
    }

    private String negate(String f) {
        if (f.startsWith(prefix:"-") || f.startsWith(prefix:"!")) {
            return f.substring(beginIndex:1); // supprime le symbole de négation
        }
        return "-" + f;
    }
}

```

FIGURE 4.3 – classe TheorieParDefault

## Exercice 1 : Extensions en logique des défauts

Dans cet exercice, nous étudions un ensemble de défauts  $D = \{d_1, d_2\}$  où :

- $d_1 = \frac{A:B}{C}$
- $d_2 = \frac{A:\neg C}{D}$

4.  $W = \{A, \neg B \wedge C\}$ 

- Le défaut  $d_1$  est bloqué car  $B$  est contredit.
- Le défaut  $d_2$  est aussi bloqué puisque sa justification  $\neg C$  est fausse.
- **Extension finale** :  $\{A, \neg B, C\}$

**Implémentation Java – Composants principaux**

Pour modéliser les défauts et les mondes possibles, nous avons implémenté les classes suivantes :

- **Default** : représente un défaut avec un antécédent, une justification et une conclusion.
- **Monde** : permet de représenter un ensemble de formules (par exemple  $W = \{A, \neg B\}$ ).
- **TheorieParDefault** : calcule les extensions possibles à partir d'un ensemble de défauts et d'un monde donné.

Extrait du code utilisé pour l'exercice :

```
Default d1 = new Default("A", "B", "C");
Default d2 = new Default("A", "C", "D");

List<Default> defaults = Arrays.asList(d1, d2);

Set<String> mondeW = new HashSet<>(Arrays.asList(...a:"A", "B", "C"));

TheorieParDefault theorie = new TheorieParDefault(defaults, mondeW);
theorie.executerTheorie();
```

FIGURE 4.4 – extrait du code

Capture d'écran de l'exécution :

```

===== World W1 =====
Input: [¬A]
Extension: [¬A]

===== World W2 =====
Input: [A, ¬B]
Extension: [A, D, ¬B]

===== World W3 =====
Input: [A, ¬C ∨ ¬D]
Extension: [A, C, ¬C ∨ ¬D]

===== World W4 =====
Input: [A, ¬B ∧ C]
Extension: [A, ¬B ∧ C, C]

```

## Exercice 6 : Raisonnement logique avec Tweety et Java

Dans cet exercice, nous avons implémenté un moteur de raisonnement par défaut en Java pour simuler les extensions d'une théorie contenant des défauts.

Défauts considérés :

$$\begin{cases} d_1 = \frac{a:b}{b} \\ d_2 = \frac{:\neg a}{\neg a} \\ d_3 = \frac{:\neg a}{a} \end{cases}$$

avec la relation de priorité suivante :  $d_1 \prec d_3 \prec d_2$

### Implémentation Java – Composants principaux

Nous avons utilisé les composants suivants pour modéliser et exécuter cette théorie :

- Default : pour modéliser chaque défaut logique.
- TheorieParDefaut : qui contient la logique de construction des extensions par priorités.

- `FormuleUtil` : utilitaire de traitement de chaînes de caractères représentant les formules.

Voici un extrait du code utilisé :

```
Default d1 = new Default ( " a " , " b " , " b " ) ;
Default d2 = new Default ( " " , " a " , " a " ) ;
Default d3 = new Default ( " " , " a " , " a " ) ;
List < Default > defaults = Arrays . asList ( d1 , d3 , d2 ) ;
TheorieParDefault theorie = new TheorieParDefault ( defaults , new ArrayList
() ) ;
theorie . executerTheorie () ;
```

FIGURE 4.5 – extrait du code

Capture d'écran de l'exécution :

```
/****** Execution de la théorie priorisée *****/
W =
D = {
    d1: [(a):(b) ==> (b)] ;
    d2: [([]):(~a) ==> (~a)] ;
    d3: [([]):(a) ==> (a)]
}
Priorités: d1 < d3 < d2

Trying eeee & eeee[
Trying eeee & eeee
Trying eeee & eeee
Trying eeee & eeee
Trying eeee & eeee
Trying eeee & eeee
Extensions classiques possibles :

E: Th(W ∪ (~a))
    = eeee & ~a

E: Th(W ∪ (b & a))
```



# Chapitre 5

## TP5 : Les Réseaux Sémantiques

Ce TP se concentre sur l'implémentation de réseaux sémantiques, y compris différents algorithmes associés. Un réseau sémantique est un graphe marqué utilisé pour représenter les connaissances. Dans ce TP, nous utiliserons un fichier JSON pour définir un réseau sémantique, où chaque nœud est identifié par un label et un ID, permettant ainsi de représenter les relations entre les nœuds. Les algorithmes seront implémentés en utilisant le langage Python.

### 5.1 Partie 1 : Implémenter l'algorithme de propagation de marqueurs dans les réseaux sémantiques

Pour cet algorithme, nous allons utiliser le réseau sémantique fourni par le lien donné dans la série de TP, détaillé dans la figure suivante :



FIGURE 5.1 – Réseau sémantique – Partie 1

### 5.1.1 Code

Le code de propagation de marqueurs prend en entrée deux nœuds et une relation pour déterminer s'il existe un chemin entre ces nœuds qui respecte cette relation spécifique. Par exemple, si on demande si la logique d'ordre 0 est une logique classique à travers la relation "is\_a", le code parcourt le graphe en partant de 'Logique D'ordre 0', remonte jusqu'à 'Logique Classiques' en suivant les relations "is\_a" et confirme cette relation.

## 5.1. PARTIE 1 : IMPLÉMENTER L'ALGORITHME DE PROPAGATION DE MARQUEURS DANS I

```
def propagation_de_marqueurs(reseau_semantique, requetes):
    nodes = reseau_semantique["nodes"]
    solutions_found = []

    for req in requetes:
        node1, node2, relation = req
        solution_found = False

        try:
            M1 = [node for node in nodes if node["label"] == node1][0]
            M2 = [node for node in nodes if node["label"] == node2][0]

            edges = reseau_semantique["edges"]

            # Vérification directe de la relation recherchée
            direct_relation = any(edge for edge in edges if edge["from"] == M1["id"] and edge["to"] == M2["id"] and edge["label"] == relation)
            if direct_relation:
                solution_found = True
            else:
                propagation_edges = [edge for edge in edges if (edge["to"] == M1["id"] and edge["label"] == "is a")]

                while len(propagation_edges) != 0 and not solution_found:
                    temp_node = propagation_edges.pop()
                    temp_node_contient_edges = [edge for edge in edges if (edge["from"] == temp_node["from"] and edge["label"] == relation)]
                    solution_found = any(d["to"] == M2["id"] for d in temp_node_contient_edges)
                    if not solution_found:
                        temp_node_is_a_edges = [edge for edge in edges if (edge["to"] == temp_node["from"] and edge["label"] == "is a")]
                        propagation_edges.extend(temp_node_is_a_edges)

                solutions_found.append(get_label(reseau_semantique, M2, relation) if solution_found else "il n'y a pas de lien entre les 2 noeuds")

        except IndexError:
            solutions_found.append("Aucune reponse n'est fournie par manque de connaissances.")

    return solutions_found
```

FIGURE 5.2 – Code Partie 1

### 5.1.2 Result

```
C:\Users\mehdi\OneDrive\Bureau\RCR Project\TP 5>python main.py

Menu
-----
1) Partie 1 : L'algorithme de propagation de marqueurs
2) Partie 2 : L'algorithme d'héritage
3) Partie 3 : L'algorithme de propagation de marqueurs avec exception
4) Quitter
-----

Entrer le numero de la partie : 1
***** Partie 1 : L'algorithme de propagation de marqueurs *****
Modes Logiques is a Modes de Representations des connaissances --> il y a un lien entre les 2 noeuds : Modes Logiques, Modes Graphiques
Logique d'ordre 1 is a Logiques Classiques --> il y a un lien entre les 2 noeuds : Logique d'ordre 1, Logique d'ordre 0
Reseaux Semantique is a Modes Graphiques --> il y a un lien entre les 2 noeuds : Reseaux Semantique, Reseaux Bayesiens
Modes de Representations des connaissances is a Axiome A9 --> Aucune reponse n'est fournie par manque de connaissances.
```

FIGURE 5.3 – Exécution Partie 1

## 5.2 Partie 2 : Implémenter l'algorithme d'héritage

Pour cet algorithme, nous allons utiliser un autre réseau sémantique exploitant mieux les propriétés de cet algorithme et qui a, lui aussi, été vu en cours et qui est détaillé dans la figure suivante :

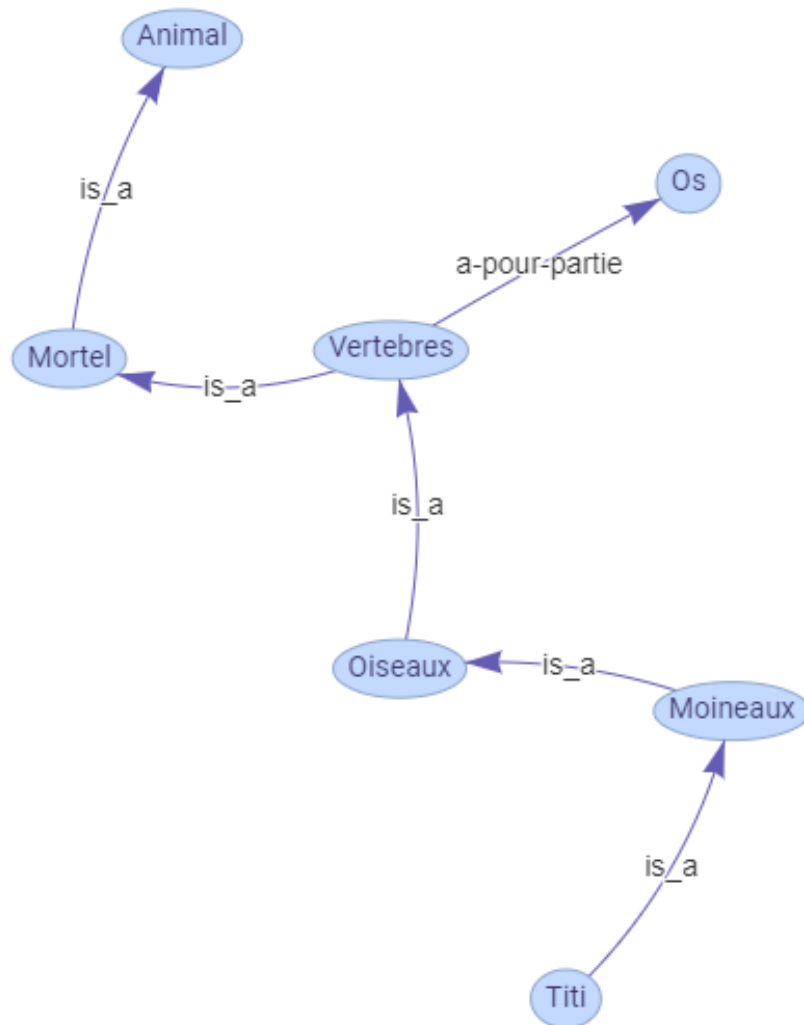


FIGURE 5.4 – Réseau sémantique Partie 2

### 5.2.1 Code

L'algorithme d'héritage extrait et compile toutes les propriétés héritées le long d'un chemin spécifié par des relations `is_a`. Si on cherche les héritages de Logique Modale, le code identifiera les nœuds de Logique d'Ordre et Logique Classique (en remontant par `is_a`) comme des héritages, listant toutes leurs propriétés transmises à Logique Modale.

```
def heritage(reseau_semantique, starting_node_name):
    nodes = reseau_semantique["nodes"]
    edges = reseau_semantique["edges"]
    all_edges = []
    properties = []

    #get node where given name
    node = [node for node in nodes if node["label"] == starting_node_name][0]

    # get all inherited nodes IDs
    direct_edges = [edge["to"] for edge in edges if (edge["from"] == node["id"] and edge["label"] == "is-a")]

    while direct_edges:
        n = direct_edges.pop()

        # get inherited nodes Label
        all_edges.append(get_label(reseau_semantique, n))

        # if the inherited are also inherited by other nodes
        direct_edges.extend([edge["to"] for edge in edges if (edge["from"] == n and edge["label"] == "is-a")])

        properties_nodes = [edge for edge in edges if ((edge["from"] == n or edge["from"] == node["id"]) and edge["label"] != "is-a")]

        for pn in properties_nodes:
            properties.append(":".join([pn["label"], get_label(reseau_semantique, pn["to"])]))

    return all_edges, properties
```

FIGURE 5.5 – Code Partie 2

### 5.2.2 Result

```
Entrer le numero de la partie : 2
***** Partie 2 : L'algorithme d'heritage *****
Noeuds disponibles : Titi, Moineaux, Oiseaux, Vertebres, Animal, Mortel, Os
Entrer le nom du noeud de départ : Titi
Inférence à partir de 'Titi' :
- Moineaux
- Oiseaux
- Vertebres
- Mortel
- Animal
Propriétés déduites :
- a-pour-partie: Os
```

FIGURE 5.6 – Exécution Partie 2

## 5.3 Partie 3 : Implémenter l'algorithme d'exception

Pour cette partie, on reprend le réseau sémantique vu dans la première partie.

### 5.3.1 Code

L'algorithme d'exception est similaire à la propagation de marqueurs mais prend en compte les exceptions spécifiées. Par exemple, si certains liens "is\_a" sont marqués comme des exceptions, le code les évitera lors de la propagation. Si on demande si 'Titi' est un 'Animal', mais le lien entre 'Oiseaux' et 'Vertébrés' est une exception, le code ne reconnaîtra pas 'Titi' comme un 'Animal' en utilisant cette route spécifique.

```
def propagation_de_marqueurs(reseau_semantique, requetes):
    nodes = reseau_semantique["nodes"]
    edges = reseau_semantique["edges"]
    solutions_found = []

    for req in requetes:
        node1, node2, relation = req
        solution_found = False

        try:
            M1 = [node for node in nodes if node["label"] == node1][0]
            M2 = [node for node in nodes if node["label"] == node2][0]

            # Vérification directe de la relation recherchée (hors exception)
            direct_relation = any(
                edge for edge in edges
                if edge["from"] == M1["id"] and edge["to"] == M2["id"] and edge["label"] == relation and edge.get("edge_type") != "exception"
            )
            if direct_relation:
                solution_found = True
            else:
                propagation_edges = [
                    edge for edge in edges
                    if (edge["to"] == M1["id"] and edge["label"] == "is a" and edge.get("edge_type") != "exception")
                ]
                while propagation_edges and not solution_found:
                    temp_edge = propagation_edges.pop()
                    temp_node_contient_edges = [
                        edge for edge in edges
                        if (edge["from"] == temp_edge["from"] and edge["label"] == relation and edge.get("edge_type") != "exception")
                    ]
                    solution_found = any(d["to"] == M2["id"] for d in temp_node_contient_edges)
                    if not solution_found:
                        temp_node_is_a_edges = [
                            edge for edge in edges
                            if (edge["to"] == temp_edge["from"] and edge["label"] == "is a" and edge.get("edge_type") != "exception")
                        ]
                        propagation_edges.extend(temp_node_is_a_edges)

                solutions_found.append(get_label(reseau_semantique, M2, relation) if solution_found else "il n'y a pas de lien entre les 2 noeuds")
        except IndexError:
            solutions_found.append("Aucune reponse n'est fournie par manque de connaissances.")

    return solutions_found
```

FIGURE 5.7 – Code Partie 3

### 5.3.2 Result

```
Entrer le numero de la partie : 3
***** Partie 3 : L'algorithme de propagation de marqueurs avec exception *****
Logique d'ordre 1 is a Logiques Classiques --> Aucune reponse n'est fournie par manque de connaissances.
Modes de Representations des connaissances contient Axiome A7 --> il y a un lien entre les 2 noeuds : Systeme T, Systeme S5
Reseaux Semantique is a Modes Graphiques --> il y a un lien entre les 2 noeuds : Reseaux Semantique, Reseaux Bayesiens
Modes de Representations des connaissances is a Axiome A9 --> Aucune reponse n'est fournie par manque de connaissances.
```

FIGURE 5.8 – Exécution Partie 3

# Chapitre 6

## TP6 Langage de Description

### 6.1 Choix de l'outil

Pour ce TP, nous avons choisi **Owlready2**, une bibliothèque Python permettant de manipuler des ontologies au format OWL et d'effectuer du raisonnement en logique des descriptions. Les principaux avantages sont :

- **Simplicité d'utilisation** grâce à la syntaxe Python.
- Intégration facile avec des raisonneurs tels que *Pellet* pour l'inférence automatique.
- Support complet des concepts OWL : classes, propriétés, restrictions, individus, axiomes.

### 6.2 Fonctionnement de Owlready2

Owlready2 permet :

- De charger et sauvegarder des ontologies OWL (formats RDF/XML, OWL/XML, etc.).
- De créer et manipuler dynamiquement des classes (concepts), propriétés (rôles) et individus (instances).
- De définir des axiomes en logique des descriptions, tels que inclusions, disjonctions, équivalences et restrictions.
- D'exécuter un raisonneur (Pellet intégré) pour inférer des connaissances, classifier les concepts, détecter des incohérences, etc.

## 6.3 Composants de la TBox en logique des descriptions

La **TBox** (terminological box) contient la partie conceptuelle de la connaissance, c'est-à-dire les définitions et relations entre concepts et rôles :

- **Concepts** : classes comme *Personne*, *Aliment*, *University*.
- **Inclusions** : par exemple, *Car* est un sous-concept de *Vehicle*, écrit  $Car \sqsubseteq Vehicle$ .
- **Restrictions** : par exemple, "Une personne enseigne seulement à des étudiants" est modélisée par une restriction sur un rôle.
- **Disjonctions et équivalences** entre concepts, définissant la sémantique précise des classes.

Les **rôles** (ou propriétés) représentent les relations entre individus, comme *enseigne*.

## 6.4 Exemple de code avec Owlready2

Voici un extrait de code illustrant la définition de concepts, rôles, individus, ainsi que l'utilisation du raisonneur Pellet en Python avec Owlready2 :

```
from owlready2 import *

onto = get_ontology("http://testxyz.org/onto.owl")

with onto:

    # Définition des concepts (classes)
    class Personne(Thing): pass
    class Aliment(Thing): pass
    class University(Thing): pass

    AllDisjoint([Personne, Aliment, University]) # Disjonction stricte

    # Définition des propriétés (rôles)
    class mange(Personne >> Thing): pass
    class enseigne(Personne >> Thing): pass
    class enseigne_par(ObjectProperty): inverse_property = enseigne
    class mange_par(ObjectProperty): inverse_property = mange
    class PartieDe(Thing >> Thing): pass

    # Définition des entités composées
    class Faculty(Thing): equivalent_to = [Thing & PartieDe.some(University)]
    class Departement(Thing): equivalent_to = [Thing & PartieDe.some(Faculty)]
    class Enseignant(Personne): equivalent_to = [Personne & enseigne.only(Personne)]
    class Etudiant(Personne): equivalent_to = [Personne & enseigne_par.only(Enseignant)]

    # Définition d'instances génériques (ABox partielle)
    class Mohamed(Thing): equivalent_to = [Personne & mange.only(Aliment)]
    class Meriem(Personne): equivalent_to = [Enseignant & mange.some(Aliment) & enseigne.only(Etudiant)]
    class MalBouffe(Thing): equivalent_to = [Aliment & mange_par.some(Personne)]
```

FIGURE 6.1 – Code Partie 1

Ce code permet de construire une ontologie, définir des axiomes complexes, ajouter des individus, puis d'exécuter un raisonnement automatique pour inférer de nouvelles connaissances à partir des axiomes définis.



```

AllDisjoint([Etudiant, Enseignant])
AllDisjoint([Meriem, Mohamed])
AllDisjoint([MalBouffe, Departement, Faculty, University])

sync_reasoner_pellet(infer_property_values=True)
onto.save(file="tp_rc1.owl", format="rdxml")

with onto:
    USTHB = onto.University()
    Sidali = onto.Etudiant()
    Chocolat = onto.Aliment()
    Belhadi = onto.Personne()

    SI = Thing() # Département
    INFO = Thing() # Faculté

    INFO.PartieDe.append(USTHB)
    SI.PartieDe.append(INFO)

    Sidali.mange = [Chocolat]
    Belhadi.enseigne = [Sidali]

sync_reasoner_pellet(infer_property_values=True)
onto.save(file="tp_rc2.owl", format="rdxml")

```

FIGURE 6.2 – Code Partie 2

```

* Owlready2 * Pellet took 1.0793547630310059 seconds
* Owlready * Reparenting onto.aliment1: {onto.Aliment} => {onto.MalBouffe}
* Owlready * Reparenting onto.thing1: {owl.Thing} => {onto.Departement}
* Owlready * Reparenting onto.thing2: {owl.Thing} => {onto.Faculty}
* Owlready * Reparenting onto.personne1: {onto.Personne} => {onto.Enseignant}
* Owlready * (NB: only changes on entities loaded in Python are shown, other changes are done but not listed)

```

FIGURE 6.3 – Terminal output

## 6.5 Rôle des fichiers .owl

Les fichiers `.owl` représentent des ontologies selon le standard OWL (Web Ontology Language), basé sur la logique de description. Ces fichiers jouent un rôle fondamental dans la modélisation et l'échange de connaissances structurées. Ils permettent notamment :

- la définition de concepts (classes), de rôles (propriétés), et d'individus (instances) ;
- la sauvegarde de la base de connaissances ;
- le raisonnement automatique via un moteur d'inférence (comme HermiT, Pellet, ou en Python via Owlready2) ;
- l'interopérabilité avec des outils comme Protégé, Owlready2, ou les APIs OWL en Java.

## 6.6 Affichage d'une ontologie OWL

```

<owl:ObjectProperty rdf:about="#mange">
  <rdfs:range rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
  <rdfs:domain rdf:resource="#Personne"/>
  <owl:inverseOf rdf:resource="#mange_par"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#enseigne">
  <rdfs:range rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
  <rdfs:domain rdf:resource="#Personne"/>
  <owl:inverseOf rdf:resource="#enseigne_par"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#enseigne_par">
  <owl:inverseOf rdf:resource="#enseigne"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#mange_par">
  <owl:inverseOf rdf:resource="#mange"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#PartieDe">
  <rdfs:domain rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
  <rdfs:range rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
</owl:ObjectProperty>

```

FIGURE 6.4 – Fichier output

## 6.7 Modélisation - Exercice 3

Voici la modélisation des assertions TBox et ABox en langage de description, puis leur implémentation via Owlready2.

**TBox :**

$$\text{VILLEINT} \equiv \text{VILLE} \sqcap \exists \text{utilise.TIC} \sqcap (\exists \text{améliore.SERVICE} \sqcup \exists \text{réduit.COÛTS})$$

$$\text{CYBERVILLE} \sqsubseteq \text{VILLEINT}$$

$$\text{VILLEINT} \sqsubseteq \forall \text{intègre.TECHCAPTSF}$$

$$\text{URBANRESP} \sqsubseteq \neg(\exists \text{développe.VILLEINT})$$

ABox :

VILLEINT(Amsterdam)

## 6.8 Code Python Owlready2

```
from owlready2 import *

# Création d'une ontologie
onto = get_ontology("http://smartcity.org/onto.owl")

with onto:
    # Concepts atomiques (classes de base)
    class Ville(Thing): pass
    class VilleIntelligente(Ville): pass
    class CyberVille(VilleIntelligente): pass
    class TIC(Thing): pass
    class Service(Thing): pass
    class CoutS(Thing): pass
    class UrbanisationResponsable(Thing): pass
    class TechnologieCapteursSF(Thing): pass
    class ChangementClimatique(Thing): pass
    class RestructurationEco(Thing): pass
    class DeveloppementDurable(Thing): pass
    class MobiliteIntelligente(Thing): pass

    # Rôles (propriétés)
    class utilise(Ville >> TIC): pass
    class améliore(Ville >> Service): pass
    class réduire(Ville >> CoutS): pass
    class integre(Ville >> TechnologieCapteursSF): pass
    class developpe(Thing >> VilleIntelligente): pass
    class repond_a(Ville >> Thing): pass # générique pour les changements
    class doit_developper(Ville >> Thing): pass
```

FIGURE 6.5 – Code exo 3 Partie 1

```

from owlready2 import *

onto = get_ontology("http://testxyz.org/onto.owl")

with onto:

    # Définition des concepts (classes)
    class Personne(Thing): pass
    class Aliment(Thing): pass
    class University(Thing): pass

    AllDisjoint([Personne, Aliment, University]) # Disjonction stricte

    # Définition des propriétés (rôles)
    class mange(Personne >> Thing): pass
    class enseigne(Personne >> Thing): pass
    class enseigne_par(ObjectProperty): inverse_property = enseigne
    class mange_par(ObjectProperty): inverse_property = mange
    class PartieDe(Thing >> Thing): pass

    # Définition des entités composées
    class Faculty(Thing): equivalent_to = [Thing & PartieDe.some(University)]
    class Departement(Thing): equivalent_to = [Thing & PartieDe.some(Faculty)]
    class Enseignant(Personne): equivalent_to = [Personne & enseigne.only(Personne)]
    class Etudiant(Personne): equivalent_to = [Personne & enseigne_par.only(Enseignant)]

    # Définition d'instances génériques (ABox partielle)
    class Mohamed(Thing): equivalent_to = [Personne & mange.only(Aliment)]
    class Meriem(Personne): equivalent_to = [Enseignant & mange.some(Aliment) & enseigne.only(Etudiant)]
    class MalBouffe(Thing): equivalent_to = [Aliment & mange_par.some(Personne)]

```

FIGURE 6.6 – Code exo 3 Partie 2

## 6.9 Capture d'écran du fichier owl

```

from owlready2 import *

onto = get_ontology("http://testxyz.org/onto.owl")

with onto:

    # Définition des concepts (classes)
    class Personne(Thing): pass
    class Aliment(Thing): pass
    class University(Thing): pass

    AllDisjoint([Personne, Aliment, University]) # Disjonction stricte

    # Définition des propriétés (rôles)
    class mange(Personne >> Thing): pass
    class enseigne(Personne >> Thing): pass
    class enseigne_par(ObjectProperty): inverse_property = enseigne
    class mange_par(ObjectProperty): inverse_property = mange
    class PartieDe(Thing >> Thing): pass

    # Définition des entités composées
    class Faculty(Thing): equivalent_to = [Thing & PartieDe.some(University)]
    class Departement(Thing): equivalent_to = [Thing & PartieDe.some(Faculty)]
    class Enseignant(Personne): equivalent_to = [Personne & enseigne.only(Personne)]
    class Etudiant(Personne): equivalent_to = [Personne & enseigne_par.only(Enseignant)]

    # Définition d'instances génériques (ABox partielle)
    class Mohamed(Thing): equivalent_to = [Personne & mange.only(Aliment)]
    class Meriem(Personne): equivalent_to = [Enseignant & mange.some(Aliment) & enseigne.only(Etudiant)]
    class MalBouffe(Thing): equivalent_to = [Aliment & mange_par.some(Personne)]

```

FIGURE 6.7 – Code exo 3 Partie 3

## Conclusion

Au fil de ces travaux pratiques, nous avons exploré plusieurs formes de logique utilisées en intelligence artificielle et en informatique : la logique propositionnelle, la logique du premier ordre, la logique des défauts, ainsi que la logique de description.

Ces différentes logiques nous ont permis de mieux comprendre comment un système peut modéliser et raisonner sur des connaissances issues du monde réel. Nous avons appris à :

- modéliser des situations complexes en utilisant des langages logiques adaptés ;
- utiliser des outils et des bibliothèques pour manipuler des bases de connaissances ;
- appliquer des raisonneurs pour générer automatiquement des inférences ;
- étudier la véracité de formules logiques dans des contextes donnés ;
- représenter visuellement des informations et déduire de nouveaux faits.

Ces compétences constituent une base essentielle pour aborder des domaines avancés tels que le Web sémantique, les systèmes experts, les systèmes à base de règles et les ontologies. Ce TP nous a offert une vue d'ensemble concrète de l'utilisation de la logique comme moteur d'intelligence dans les systèmes informatiques.