Tp2 : Logique des Prédicats

Introduction :

Dans ce qui suit, nous présentons notre utilisation de la logique des prédicats avec la librairie Java Tweety pour modéliser des situations impliquant des personnes, des activités et des relations entre eux. Nous démontrons comment nous avons défini des prédicats, des constantes, des foncteurs, et des formules logiques pour représenter notre exemple, ainsi que la manière dont nous avons utilisé la librairie Tweety pour manipuler ces entités logiques.

Méthodologie :

Choix d'un exemple concret :

Nous explorerons l'utilisation de la librairie Tweety pour modéliser les relations entre différentes entités en utilisant des prédicats et des fonctions, avec comme scénario la représentation de relations entre personnes et activités. Voici une description détaillée de l'exemple :

1- **Définition des entités logiques :** définir les sortes (ou types) pour les entités, telles que "personne" et "activité".

```
FolSignature sig = new FolSignature();
Sort person = new Sort(name:"person");
sig.add(person);
Constant Chafaa = new Constant(name:"Chafaa", person); // Chafaa est une personne
sig.add(Chafaa);
Constant Kenza = new Constant(name:"Kenza", person); // Kenza est une personne
sig.add(Kenza);
Constant david = new Constant(name:"David", person); // David est une personne
sig.add(david);

Sort activity = new Sort(name:"activity");
sig.add(activity);
Constant hiking = new Constant(name:"hiking", activity); // hiking est une activité
sig.add(hiking);
Constant swimming = new Constant(name:"swimming", activity); // swimming est une activité
sig.add(swimming);
Constant painting = new Constant(name:"painting", activity); // painting est une activité
sig.add(painting);
```

2- **Définitions des prédicats et des foncteurs :** pour représenter des relations entre les entités, tels que "Participates" (participation à une activité) et "Organizes" (organisation d'une activité).

Le prédicat "Organizes" est défini avec une arité de 2, indiquant qu'il s'agit d'un prédicat prenant deux termes en argument.

Une fonction "knows est définie avec une arité de 1, indiquant qu'elle prend un terme de type person en argument.

```
Predicate participates = new Predicate(name:"Participates", 1); // création du prédicat Participates d'arité 1
sig.add(participates);
1 = new ArrayList<Sort>();
1.add(person);
1.add(activity);
Predicate organizes = new Predicate(name:"Organizes", 1); // création du prédicat Organizes qui est d'arité 2
sig.add(organizes);
1 = new ArrayList<Sort>();
1.add(person);
// isFriendWith est une fonction qui prend un terme qui est de type personne
Functor knows = new Functor(name:"knows", 1, person);
sig.add(knows);
```

3- **Construction de la base de connaissances :**

En utilisant la classe **FolBeliefSet**, nous avons créé une base de connaissances logique pour stocker nos formules logiques.
Pour l'ajout des formules logiques à la base de connaissances le parser **FolParser est utilisé.**

```
FolBeliefSet b = new FolBeliefSet();
b.setSignature(sig);
FolParser parser = new FolParser();
parser.setSignature(sig);
```

4- Ajout des axiomes :

On ajoute plusieurs axiomes pour exprimer des règles logiques concernant les relations entre les personnes, les activités et les connaissances.
Pour tout individu X, s'il participe à une activité, alors il existe une activité Y qu'il organise.
Si une personne participe à une activité, alors toute personne qui la connaît participe également à cette activité
Kenza organise une randonnée (hiking).
Chafaa participe à une activité.

```
b.add(parser.parseFormula(text:"forall X:(Participates(X) => (exists Y:(Organizes(X,Y))))"));
// si X participe à une activité, alors knows(X) participe également à cette activité
b.add(parser.parseFormula(text:"forall X:(Participates(X) => Participates(knows(X)))"));
b.add(parser.parseFormula(text:"Participates(Chafaa)")); // Chafaa participe à une activité
b.add(parser.parseFormula(text:"Organizes(Kenza, hiking)")); // Kenza organise une randonnée
```

Résultats :
Nous avons affiché les formules logiques de notre base de connaissances pour vérifier qu'elles ont été correctement ajoutées.

```
Organizes(Kenza,hiking)
Participates(Chafaa)
forall X: ((Participates(X)=>exists Y: (Organizes(X,Y))))
forall X: ((Participates(X)=>Participates(knows(X))))
```

Les résultats obtenus ont montré que nos formules logiques étaient cohérentes et conformes aux règles que nous avions définies.

Conclusion :

Ce travail démontre l'efficacité de l'utilisation de la logique des prédicats et de la librairie Java Tweety pour modéliser des problèmes impliquant des relations logiques entre différents types d'entités.

Tp3 : Logique Modale

Introduction :

Ce rapport présente une étude pratique de la logique modale à travers deux approches distinctes : la modélisation avec **Modal Logic Playground** et l'implémentation en Java avec **la librairie Tweety**. L'objectif principal de cette étude est de vérifier la valeur de vérité de quelques formules dans le cadre de la logique modale.

Méthodologie :

Modal Logic Playground :

Fonctionnement : Modal Logic Playground offre une interface conviviale permettant aux utilisateurs de saisir des formules de logique modale dans un langage formel et de les vérifier. Il permet de définir des mondes possibles, d'attribuer des valeurs de vérité à des propositions atomiques, et d'appliquer des opérateurs modaux tels que □ (nécessairement) et ◇ (possiblement).
Utilisation pour la modélisation : Dans cette partie de l'expérience, nous avons utilisé Modal Logic Playground pour modéliser différentes situations et vérifier la validité de formules de logique modale. Nous avons saisi les formules pertinentes dans l'outil, défini les mondes possibles et interprété les résultats pour évaluer la vérité des propositions modales.
1-Modelisation :

On choisit le nombre de variables propositionnelles : dans notre exemple = 4 (p, q, r, s) : initialement elle est ainsi :

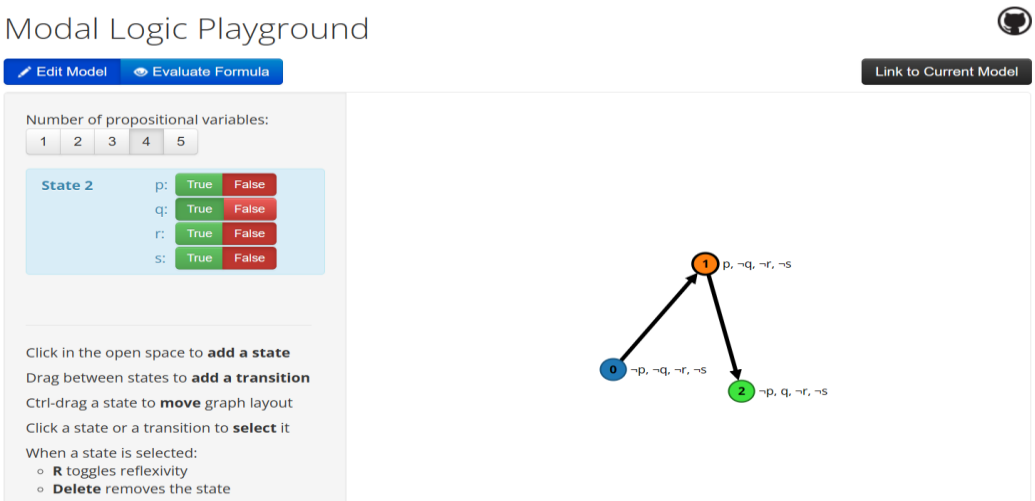


Figure .. : Modal Logic Playground

On prend un exemple de 7 mondes avec 4 variables puis on passe à l'évaluation des formules.

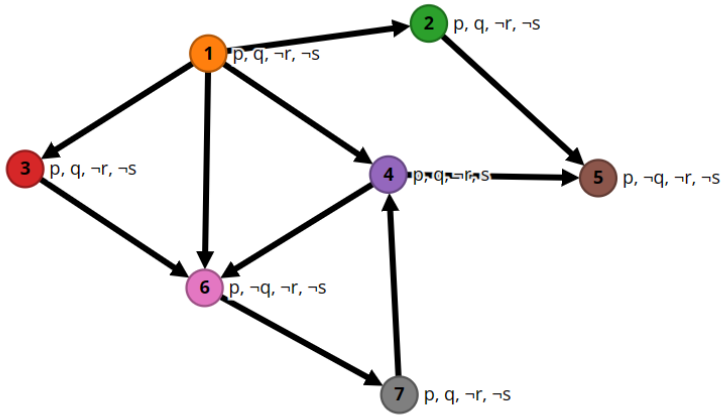
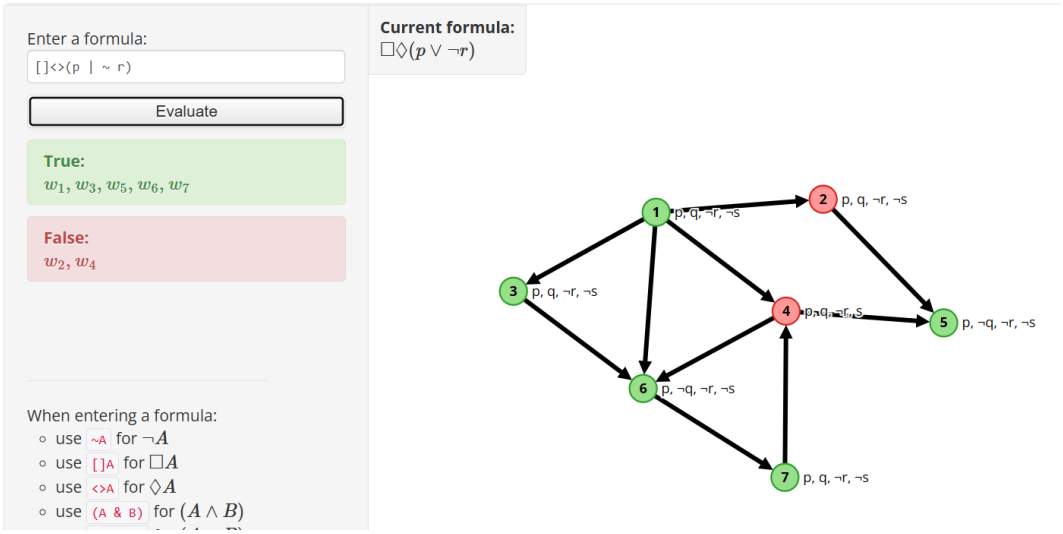


Figure : Exemple de modélisation d'un modèle modale

2-Evaluation :

[] <> (p | ~ r)



<> (p & q)

Enter a formula:

```
<> (p & q)
```

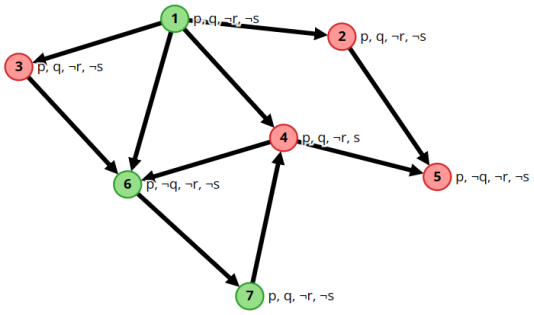
Evaluate

True:
w_1, w_6, w_7

False:
w_2, w_3, w_4, w_5

When entering a formula:
- use `~A` for $\neg A$
- use `[]A` for $\Box A$
- use `<>A` for $\Diamond A$
- use `(A & B)` for $(A \wedge B)$

Current formula:
$\Diamond(p \wedge q)$



~[]<>p | ~s

Enter a formula:

```
~[]<>p | ~s
```

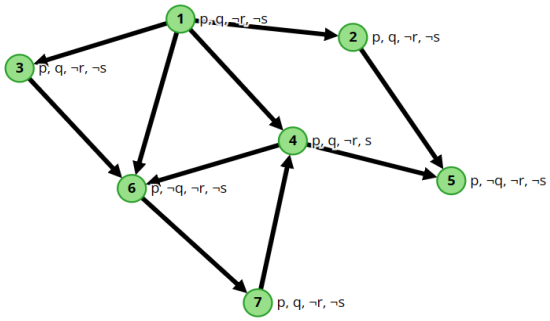
Evaluate

True:
$w_1, w_2, w_3, w_4, w_5, w_6, w_7$

False:
$\varnothing$

When entering a formula:
- use `~A` for $\neg A$
- use `[]A` for $\Box A$
- use `<>A` for $\Diamond A$
- use `(A & B)` for $(A \wedge B)$

Current formula:
$(\neg\Box\Diamond p \vee \neg s)$



~<> <> (p -> q | r)

Enter a formula:

```
~<> <> (p -> q | r)
```

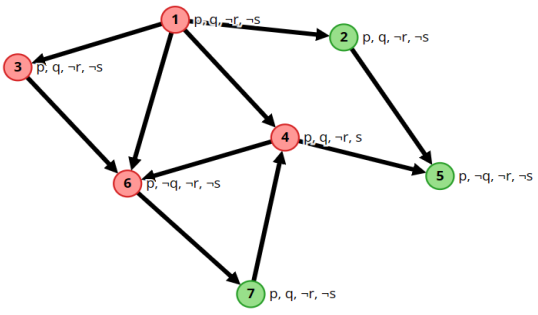
Evaluate

True:
w_2, w_5, w_7

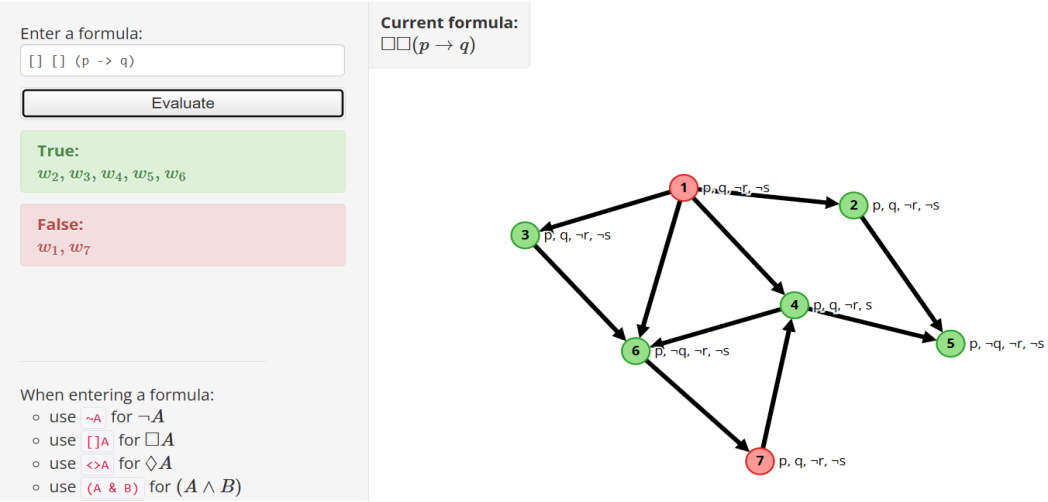
False:
w_1, w_3, w_4, w_6

When entering a formula:
- use `~A` for $\neg A$
- use `[]A` for $\Box A$
- use `<>A` for $\Diamond A$
- use `(A & B)` for $(A \wedge B)$

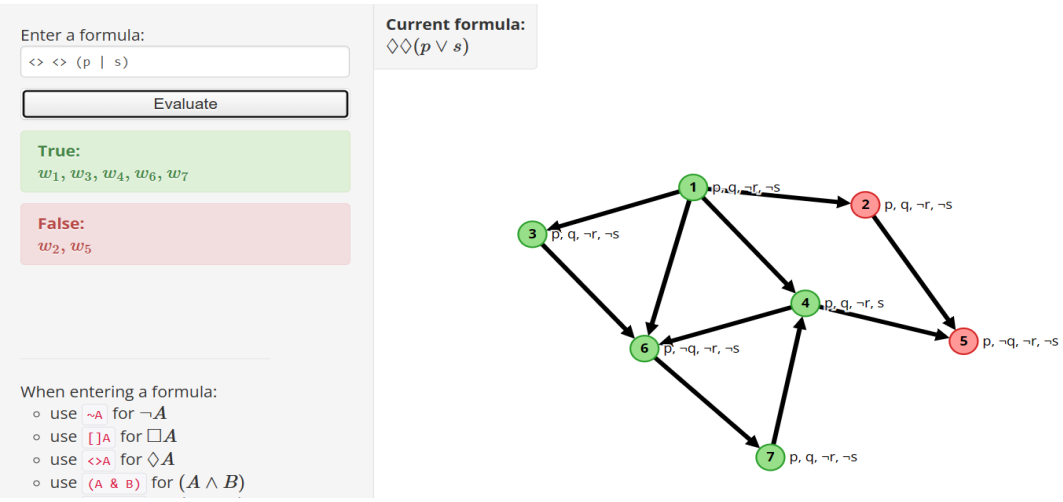
Current formula:
$\neg\Diamond\Diamond(p \rightarrow (q \vee r))$



[] [] (p -> q)



<> <> (p | s)



Librairie Java Tweety :

Fonctionnalités : Tweety offre un ensemble de classes et de méthodes pour représenter des modèles logiques, définir des opérateurs modaux, et effectuer des opérations telles que la vérification de la satisfiabilité et la déduction.

Utilisation pour l'implémentation en Java :

Voici un exemple démontrant l'utilisation de la librairie Java Tweety pour travailler avec des formules modales et des formules de logique du premier ordre. L'exemple montre comment créer une base de connaissances modale, ajouter des formules, utiliser un analyseur syntaxique (parseur) pour interpréter les formules, et utiliser un moteur d'inférence (raisonneur) pour répondre aux requêtes sur la base de connaissances modale.

1-Creation de la base de connaissance modale :

```
MlBeliefSet bs = new MlBeliefSet(); // Mlbeliefset est utilise pour stocker les formules modales
```

2- Initialisation du parseur :

```
MlParser parser = new MlParser(); // Mlparser est utilise pour parser les formules modales
```

3- Définition des symboles de la logique du premier ordre :

```
FolSignature sig = new FolSignature();  // FolSignature est utilise pour stocker les symboles de la logique du premier ordre
// On ajoute les symboles de la logique du premier ordre dans la signature de la logique modale
sig.add(new Predicate(name:"fenetres_fermees", arity:0));
sig.add(new Predicate(name:"porte_verrouillee", arity:0));
// On ajoute la signature de la logique du premier ordre dans le parser de la logique modale
parser.setSignature(sig);
```

4-Ajout des formules modales a la base de connaissances :

```
// On ajoute les formules modales dans la base de connaissances modale
bs.add((RelationalFormula)parser.parseFormula(text:"<>(fenetres_fermees && porte_verrouillee)"));
bs.add((RelationalFormula)parser.parseFormula(text:"[](!fenetres_fermees || porte_verrouillee)"));
bs.add((RelationalFormula)parser.parseFormula(text:"[](porte_verrouillee && <>(!porte_verrouillee))"));
```

La base de connaissances modale que nous avons créée dans l'exemple comprend trois formules modales :

La première formule signifie "Il est possible que les fenêtres soient fermées et la porte soit verrouillée".

La deuxième formule signifie "Il est nécessaire que si les fenêtres ne sont pas fermées, alors la porte est verrouillée".

La troisième formule signifie "Il est nécessaire que si la porte est verrouillée, alors il existe un monde où la porte n'est pas verrouillée".

5-Utilisation du moteur d'inférence pour répondre aux requêtes :

```
//reasoner est utilise pour repondre aux requetes sur la base de connaissances modale par la methode query
SimpleMlReasoner reasoner = new SimpleMlReasoner();
System.out.println("[](!fenetres_fermees)      " + reasoner.query(bs, (FolFormula)parser.parseFormula(text:"[](!fenetres_fermees)")) + "\n");
System.out.println("<>(fenetres_fermees && porte_verrouillee)  " + reasoner.query(bs, (FolFormula)parser.parseFormula(text:"<>(fenetres_fermees && porte_verrouillee)")) + "\n");
System.out.println("!(porte_verrouillee)  " + reasoner.query(bs, (FolFormula)parser.parseFormula(text:"!(porte_verrouillee)")) + "\n");
```

Resultats d'exécution :

```
Modal knowledge base: { <>(fenetres_fermees&&porte_verrouillee), [](!fenetres_fermees||porte_verrouillee), [](porte_verrouillee&&<>(!porte_verrouillee)) }
[](!fenetres_fermees)      true

<>(fenetres_fermees && porte_verrouillee)  true

!(porte_verrouillee)  true
```