Faculté des Sciences et Ingénierie - Sorbonne Université

Master Informatique parcours - ANDROIDE / DAC

LRC - Logique et Représentations des Connaissances

Rapport de projet

Ecriture en Prolog d'un démonstrateur basé sur l'algorithme des tableaux pour la logique de description ALC

Réalisé par :

IORDACHE Paul-Tiberiu - M1 ANDROIDE

SAID Faten Racha - M1 DAC

Supervisé par :

Colette Faucher

Décembre 2023

TABLE DES MATIÈRES

INTRODUCTION

La Logique de Description ALC, issue du domaine de l'intelligence artificielle et de la représentation des connaissances, se positionne comme un cadre formel puissant pour la modélisation et la représentation des connaissances complexes.

Notre projet consiste à implémenter en *Prolog* un démonstrateur basé sur l'algorithme des tableaux pour la logique de description ALC.

Ainsi, ce document décrit le travail réalisé dans le cadre d'un projet académique dans le but de concrétiser et améliorer nos connaissances pratiques et théoriques récoltées dans l'Unité d'Enseignement "Logique et Représentations des Connaissances". Ce travail s'articule principalement sur trois axes :

— Le premier consiste en la vérification et la mise en forme de la TBox et de la ABox.

— Le second permet à l'utilisateur la saisie de la proposition à démontrer.

— Le troisième implémente l'algorithme de résolution basé sur la méthode des tableaux et affiche le résultat de la résolution.

De plus, nous présentons dans l'annexe les différents fichiers joints au rapport ainsi que le guide d'utilisation du programme développé et les résultats détaillés de nos jeux de tests.

CHAPITRE 1

ETAPE PRÉLIMINAIRE DE VÉRIFICATION ET DE MISE EN FORME DE LA TBOX ET DE LA ABOX

Les étapes préliminaires que notre démonstrateur doit effectuer consistent à vérifier la cohérence sémantique et syntaxique de la TBox et de la ABox données ainsi que l'absence de l'auto-référence et la mise des box sous une forme normale négative et simplifiée.

Ces vérifications sont réalisées par le prédicat **premiere_etape/3**, qui valide les prédicats **concept**, **traitement_TBox/1**, et **traitement_ABox/1**. Notons que nous faisons également appel à **listeABox-Role/1** afin de vérifier qu'elle contient exactement les instances de rôles définies dans le fichier, en s'assurant que ces derniers respectent bien la syntaxe d'une instanciation de rôle. Il est important de souligner que ces prédicats seront exécutés sur des listes construites à partir de nos box, la mise sous forme de liste étant détaillée dans la partie 1.2.

1.1 Prédicat concept

Le prédicat **concept** définit ci-dessous sert à effectuer la vérification sémantique, syntaxique, ainsi que la simplification et la mise sous forme normale négative de nos listes TBox, ABoxConcept et ABoxRole.

```
concept :- semantique,
           listeTBox(TBox),
           listeABoxConcept(ABoxConcept),
           listeABoxRole(ABoxRole),
           syntaxe(TBox, ABoxConcept, ABoxRole).
```

Un jeu de tests pour ce prédicat est présenté en annexe.

1.1.1 La correction sémantique

En ce qui concerne la correction sémantique, le prédicat **semantique** nous sert à vérifier si nos box sont sémantiquement correctes. Pour ce faire, nous avons initialement commencé par utiliser le prédicat **setof** afin de construire quatre listes distinctes. Ces listes comprennent les concepts atomiques, les concepts non-atomiques, les rôles, et les instances. Ensuite, nous avons concaténé ces quatre listes pour former une unique liste. Cette opération de concaténation est nécessaire car elle nous permet d'appliquer le prédicat **unique**, qui vérifie l'unicité de chaque élément dans la liste résultante. Cette

vérification d'unicité est cruciale pour assurer la cohérence sémantique de nos données car elle permet d'assurer le fait qu'aucun élément introduit n'est défini de deux manières différentes, à titre d'exemple un concept $personne$ ne peut pas être à la fois définit comme un concept atomique et non atomique.

1.1.2 La correction syntaxique

Le prédicat utilisé pour vérifier la correction syntaxique est **syntaxe/3** et il fait appel à deux autres prédicats : **syntaxeTBox/1** et **syntaxeABox/2**.

Le principe de **syntaxeTBox/1** est de vérifier tout d'abord si l'équivalence de notre TBox passée en paramètre est correctement définie. Cela implique d'avoir un concept non atomique à gauche de l'équivalence et une définition à droite qui respecte la syntaxe de la logique ALC. Ensuite, le prédicat procède de manière récursive en parcourant la liste de la TBox en vérifiant cette fois ci l'équivalence avec la définition de la définition si celle ci est non atomique.

Le prédicat **syntaxeABox/2** utilise lui-même deux prédicats : **syntaxeABoxConcept/1** et **syntaxeABoxRole/1**. **syntaxeABoxConcept/1** sert à vérifier si l'assertion de concept s'effectue entre une instance et un concept, à l'aide du prédicat **verifierInst/1**. Ensuite, le prédicat procède de manière récursive en parcourant la liste ABox contenant les concepts, vérifiant à chaque étape s'il s'agit bien d'une assertion de concept. **syntaxeABoxRole/1** sert à vérifier si les assertions de rôles s'effectuent correctement entre deux instances, ce qui est réalisé par le prédicat **verifierInstR/1**. De même, le prédicat procède de manière récursive en vérifiant si la liste ABox contenant des rôles contient que des assertions de rôle.

1.2 Mise sous forme de liste de la TBox et de la ABox

Dans notre programme, nous représentons les box sous forme de listes afin de pouvoir les manipuler sous *Prolog*. Ainsi, à partir du fichier fourni, nous transformons ces informations en listes à l'aide des prédicats **listeTBox/1**, **listeABoxConcept/1**, et **listeABoxRole/1**.

Pour remplir ces listes, nous utilisons le prédicat **setof**. Lorsque le setof concerne une équivalence, nous ajoutons cela à la liste composant la TBox. Si c'est une assertion de concept, nous l'ajoutons à la liste ABox contenant des concepts, et s'il s'agit d'une assertion de rôle, nous l'ajoutons à la liste ABox contenant les rôles. Ainsi, nous obtenons trois listes distinctes une pour les assertions de concepts, une autre pour les assertions de rôle et une dernière pour la TBox.

1.3 Vérification de l'abscence d'auto-référencement

L'auto-référencement se produit lorsqu'un concept non atomique est présent dans sa propre définition conceptuelle, créant ainsi un risque de boucle infinie dans le développement du programme. Pour remédier à ce problème, nous utilisons un prédicat qui vérifie l'absence d'auto-référencement,

nommé **pas_autoref**. Ce prédicat est évalué comme vrai si et seulement si aucun des éléments de la TBox ne présente d'auto-référencement, et faux dans le cas contraire. Sa définition est la suivante :

```
pas_autoref :- listeTBox(TBox), pasAutoRef_list(TBox).
```

Le prédicat **pasAutoRef_list/1** forme la liste des équivalences présentes dans le fichier à l'aide d'un prédicat de la partie 1.2. Ensuite, à partir de cette liste, il utilise récursivement le prédicat **pasAutoRef_element/2** pour tester l'absence d'auto-référence dans chaque élément de la liste obtenue de notre TBox.

Le prédicat **pasAutoRef_element/2** repose sur l'idée de vérifier l'absence de concept s'auto-référençant, équivalent à vérifier l'absence de cycle par l'opérateur d'unifiabilité entre un concept non atomique et sa définition. Il existe deux cas de figure :

— La définition est un concept atomique : dans ce cas, il n'y a rien à vérifier, et nous sommes assurés qu'il n'y a pas de cycle.

— La définition est un concept non atomique : dans ce cas, il faut vérifier que le concept n'est pas unifiable avec sa définition, puis appliquer une récursivité de manière à simplifier la définition jusqu'à ce qu'elle ne contienne que des concepts atomiques.

Un jeu de tests pour ce prédicat est présenté en annexe.

1.4 Vérification de la simplification et de la mise sous NNF

Tout d'abord, clarifions les définitions de la simplification et de la mise sous forme normale négative. La simplification implique le remplacement récursif des identificateurs de concepts complexes par leur définition. D'autre part, la mise sous forme normale négative consiste à transformer l'expression d'un concept de telle manière que toute négation présente dans cette expression soit directement associée à un concept atomique.

Ces deux traitements seront realises par les predicats suivants :

```
traitement_TBox(NNF_TBox) :-
        listeTBox(TBox),
        remplacer_liste_TBox(TBox, NNF_TBox).

traitement_ABox(NNF_ABox) :-
        listeABoxConcept(ABoxConcept),
        remplacer_liste_ABox(ABoxConcept, NNF_ABox).
```

Dans les deux situations, l'objectif est de substituer tous les concepts complexes présents dans les expressions conceptuelles par des expressions ne comprenant que des identificateurs de concepts atomiques. Cette substitution est réalisée en convertissant l'ensemble en une forme normale négative.

Ce processus est effectué par les prédicats **remplacer_liste_TBox/2** et **remplacer_liste_ABox/2**, en utilisant les expressions conceptuelles présentes dans la TBox.

Un jeu de tests pour ces deux prédicats est présenté en annexe.

CHAPITRE 2

SAISIE DE LA PROPOSITION À DÉMONTRER

L'objectif de cette partie est d'obtenir la proposition que l'utilisateur souhaite prouver à l'aide du démonstrateur. Afin de démontrer la proposition, le programme applique la méthode des tableaux sémantiques qui, pour rappel, ajoute la négation de la proposition aux assertions de la ABox et démontre que l'ensemble est insatisfiable. La proposition à démontrer peut être formulée de deux manières, comme nous le décrivons dans ce qui suit.

2.1 Proposition de type : $I : C$

Pour ce type de proposition, il faut ajouter $I : \neg C$ à la liste des assertions de concepts qu'on nomme Abi, le résultat de l'opération sera contenu dans $Abi1$. Le prédicat associé a cette exécution est **acquisition_prop_type1 /3** défini comme suit :

```
acquisition_prop_type1(Abi, Abi1, TBox) :-
            get_prop_type1(Instance, Concept),
            traitement_prop_type1((Instance, Concept),
            (Instance, New_Concept)),
            concat(Abi,[(Instance, New_Concept)], Abi1).
```

Ce prédicat permet de faire appel à **get_prop_type1 /2** qui récupère l'instance et le concept que l'utilisateur veut tester et retourne une erreur si l'une des deux entrées ne vérifie pas la syntaxe exigée dans la première partie du rapport.

Une fois la proposition acquise, il faut la simplifier et la mettre sous forme normale négative (NFF) grâce au prédicat **traitement_prop_type1 /2**.

Une fois que toutes ces conditions sont vérifiées, nous ajoutons la négation de la proposition simplifiée et mise sous NNF dans la liste des assertions de concepts.

2.2 Proposition de type : $C1 \sqcap C2 \sqsubseteq \bot$

La négation de ce type de proposition est de la forme $\exists \text{inst}, \text{inst} : C1 \sqcap C2$, ce que nous ajoutons à Abi. Le prédicat associé a cette exécution est **acquisition_prop_type2 /3** défini comme suit :

```
acquisition_prop_type2(Abi,Abi1,Tbox) :-
     get_prop_type2(C1, C2),
```

```
genere(I),
traitement_prop_type2(and(C1, C2),(I, and(New_C1, New_C2))),
concat(Abi,[(I, and(New_C1, New_C2))], Abi1).
```

Ce prédicat permet de faire appel à **get_prop_type2 /2** qui récupère les deux concepts que l'utilisateur veut tester et retourne une erreur si au moins une des deux entrées ne vérifie pas la syntaxe exigée dans la première partie du rapport.

Après avoir récupéré la proposition, le programme se charge de la simplifier et de la mettre sous forme normale négative (NFF) puis de l'ajouter a une nouvelle instance grâce aux prédicats **traitement_prop_type2 /2** et **genere /1**.

Enfin, après avoir vérifié de toutes ces conditions, nous ajoutons la négation de la proposition simplifiée et mise sous NNF dans la liste des assertions de concepts, la Abox étendue est représenté par $Abi1$.

Un jeu de tests pour ces deux prédicats **acquisition_prop_type1 /3** et **acquisition_prop_type2 /3** est présenté en annexe.

CHAPITRE 3

DÉMONSTRATION DE LA PROPOSITION

Le but dans cette partie est d'implémenter l'algorithme de résolution basé sur la méthode des tableaux et permettre au programme d'afficher le résultat de la résolution.

L'idée est qu'à partir des assertions de Abe la Abox étendue construite dans la partie précédente, nous construisons un arbre de résolution que nous déroulons progressivement, et en fonction de cette exécution, affirmer si la proposition est démontrée ou non. Notons qu'une proposition n'est démontrée que toutes les branches de l'arbre de résolution sont fermées ; soit que Abe est insatisfiable.

3.1 Prédicat tri_Abox

Nous commençons tout d'abord par introduire le prédicat **tri_Abox /6** qui permet de trier Abe en cinq listes, une pour chacun des quatre types de règles et une autre pour les assertions simples. Cela a pour objectif de facilité l'implémentation du programme et accélérer le déroulement de l'arbre de résolution. Ci-dessous un exemple dans le cas d'une assertion de concept de type "Inter" :

```
tri_Abox([(Instance, and(Concept1,Concept2))|Abi_suite],
        Lie,Lpt,Li,Lu,Ls) :-
        concat([(Instance, and(Concept1,Concept2))],Li_suite,Li),
        tri_Abox(Abi_suite,Lie,Lpt,Li_suite,Lu,Ls), !.
```

Les cinq listes définies correspondent à ce qui suit :

- Lie : la liste des assertions du type (I,some(R,C)).
- Lpt : la liste des assertions du type (I,all(R,C)).
- Li : la liste des assertions du type (I,and(C1,C2)).
- Lu : la liste des assertions du type (I,or(C1,C2)).
- Ls : la liste type (I,C) ou (I,not(C)), avec C un concept atomique.

Ainsi, le prédicat **tri_Abox /6** prend récursivement une assertion de Abe et l'ajoute à une des cinq listes en fonction de la nature de l'assertion.

3.2 Prédicat resolution

Nous passons au prédicat **resolution /6**, le plus important de cette partie étant donné qu'il détermine le résultat de la démonstration. En effet, ce prédicat déroule l'algorithme des tableaux séman-

tiques et renvoie $True$ si toutes les branches de l'arbre de résolution sont fermées, soit que Abe est insatisfiable, ce qui nous permet d'affirmer que la proposition initiale est démontrée, et $False$ sinon.

```
resolution(Lie, Lpt, Li, Lu, Ls, Abr) :-
            Lie\==[],
            not_clash(Ls),
            complete_some(Lie, Lpt, Li, Lu, Ls, Abr).
resolution([], Lpt, Li, Lu, Ls, Abr) :-
            Li\==[],
            not_clash(Ls),
            transformation_and([], Lpt, Li, Lu, Ls, Abr).
resolution([], Lpt, [], Lu, Ls, Abr) :-
            Lpt\==[],
            not_clash(Ls),
            deduction_all([], Lpt, [], Lu, Ls, Abr).
resolution([], [], [], Lu, Ls, Abr) :-
            Lu\==[],
            not_clash(Ls),
            transformation_or([], [], [], Lu, Ls, Abr).
resolution([], [], [], [], Ls, Abr) :-
            not(not_clash(Ls)),!.
```

Ce prédicat permet d'implémenter la boucle de contrôle du processus de développement de l'arbre de démonstration en se basant sur l'odre de priorité des règles défini dans l'énoncé du projet. A cet effet, notre programme commence par traiter les règles du type "Existe" suivi de "Inter", "Pour tout", et enfin "Union".

En plus de vérifier l'ordre de priorité, il est nécessaire de s'assurer qu'aucun $clash$ n'est détecté au niveau de la liste des concepts atomiques car si un $clash$ est détecté le noeud peut être élagué. Le prédicat **not_clash /1** vérifie donc que Ls ne contient pas à la fois une assertion de concept et sa négation.

De plus, ce prédicat utilise les prédicats **complete_some /6**, **transformation_and /6**, **deduction_all /6**, **transformation_or /6**, qui traduisent l'état de la Abox étendue à un instant donné, nous détaillons chacun d'eux dans les prochaines sections.

3.3 Prédicat evolue

Le prédicat **evolue /11** assigne une nouvelle assertion de concept à une des cinq listes définies dans 3.1 en faisant attention à ce qu'il n'y ait pas de doublons. Il traite ainsi le cas des règles du

type "Existe", "Inter", "Pour tout", et "Union". Ci-dessous un exemple dans le cas d'une assertion de concept de type "Inter" :

```
evolue((I,and(C1,C2)),  Lie, Lpt, Li, Lu, Ls,
                        Lie, Lpt, Li, Lu, Ls) :-
                        member((I,and(C1,C2)),Li).
evolue((I,and(C1,C2)),  Lie, Lpt, Li, Lu, Ls,
                        Lie, Lpt, Li1, Lu, Ls) :-
                        not(member((I,and(C1,C2)),Li)),
                        concat([(I,and(C1,C2))], Li, Li1).
```

Suivant le même principe, nous avons introduit **evolue_L /11** qui ajoute une liste d'assertions de concepts en appliquant une récursivité sur **evolue /11**.

Ces deux prédicats servent à mettre à jour les listes définies dans 3.1 après qu'un ou plusieurs éléments aient été générés lors de l'application d'une des règles de réécriture en ALC.

3.4 Prédicat complete_some

Le programme fait appel au prédicat **complete_some /6** pour traiter une assertion de la forme a : ∃ R.C présente dans la *Abe*.

```
complete_some([(I, some(R,C)|Lie_suite)], Lpt,Li,Lu,Ls,Abr) :-
    genere(I2),
    evolue((I2,C), Lie_suite, Lpt,Li,Lu,Ls,
    Lie1,Lpt1,Li1,Lu1,Ls1),
    concat([(I,I2,R)], Abr, Abr1), nl,
    affiche_evolution_Abox(Ls, [(I,some(R,C))|Lie_suite],
    Lpt,Li,Lu,Abr, Ls1,Lie1,Lpt1,Li1,Lu1, Abr1),
    resolution(Lie1,Lpt1,Li1,Lu1,Ls1,Abr1).
```

Ce prédicat permet de générer une nouvelle instance b telle que b : C et <a, b> : R. Il utilise pour cela le prédicat **evolue /11** pour l'assertion de concept à *Abe*, puis ajoute la nouvelle assertion de rôle à *Abr*, et génère un nouveau noeud de l'arbre de résolution sur cette nouvelle Abox.

3.5 Prédicat transformation_and

Le programme fait appel au prédicat **transformation_and /6** pour traiter une assertion de la forme a : C ⊓ D présente dans la *Abe*.

```
transformation_and(Lie, Lpt, [(I, and(C1, C2))|Li_suite],
    Lu, Ls, Abr) :-
    evolue_L([(I,C1),(I,C2)], Lie, Lpt, Li_suite, Lu, Ls,
    Lie1, Lpt1, Li1, Lu1, Ls1),
    affiche_evolution_Abox(Ls, Lie, Lpt, [(I,and(C1,C2))|Li_suite],
    Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
    resolution(Lie1,Lpt1,Li1,Lu1,Ls1,Abr).
```

Ce prédicat permet d'ajouter les assertions de concepts a : C et a : D à Abe, il utilise pour cela le prédicat **evolue_L /11**, et génère un nouveau noeud de l'arbre de résolution sur cette nouvelle Abox.

3.6 Prédicat deduction_all

Le programme fait appel au prédicat **deduction_all /6** pour traiter une assertion de la forme $a : \forall R.C$ présente dans la Abe.

```
deduction_all(Lie, [(I, all(R, C))|Lpt_suite],
    Li, Lu, Ls, Abr) :-
    setof((I2,C), member((I,I2,R), Abr), L),
    evolue_L(L, Lie, Lpt_suite, Li, Lu, Ls,
    Lie1, Lpt1, Li1, Lu1, Ls1),
    affiche_evolution_Abox(Ls, Lie, [(I, all(R, C))|Lpt_suite],
    Li, Lu, Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
    resolution(Lie1,Lpt1,Li1,Lu1,Ls1,Abr).
```

L'idée est de retrouver toutes instance b qui vérifient une assertion de concept du type <a,b> : R ; soit les instances b pour lesquelles a est en relation R. Pour ce faire le prédicat **deduction_all /6** utilise le prédicat **setof /3**.

Dans le cas où au moins une instance b a été trouvée, le programme ajoute une assertion de concept du type b : C grâce au prédicat **evolue_L /11**. Nous utilisons ce dernier prédicat car il se peut que plusieurs instances vérifient la relation R avec a.

Le programme génère par la suite un nouveau noeud de l'arbre de résolution sur cette nouvelle Abox.

3.7 Prédicat transformation_or

Le programme fait appel au prédicat **transformation_or /6** pour traiter une assertion de la forme a : C⊔D présente dans la Abe.

```
transformation_or(Lie, Lpt, Li, [(I, or(C1, C2))|Lu_suite], Ls, Abr) :-
        evolue((I,C1), Lie, Lpt, Li, Lu_suite, Ls, Lie1, Lpt1,
        Li1, Lu1, Ls1),
        affiche_evolution_Abox(Ls, Lie, Lpt, Li[(I,or(C1,C2))|Lu_suite],
        Abr, Ls1, Lie1, Lpt1, Li1, Lu1, Abr),
        resolution(Lie1,Lpt1,Li1,Lu1,Ls1,Abr),
        evolue((I,C2),Lie, Lpt, Li, Lu_suite, Ls,
        Lie2, Lpt2, Li2, Lu2, Ls2),
        affiche_evolution_Abox(Ls, Lie, Lpt, Li,
        [(I,or(C1,C2))|Lu_suite], Abr, Ls2, Lie2, Lpt2, Li2, Lu2, Abr),
        resolution(Lie2,Lpt2,Li2,Lu2,Ls2,Abr).
```

Ce prédicat permet de créer deux noeuds frères de l'arbre de résolution, tel qu'au premier noeud nous ajoutons l'assertion a : C, et dans le second l'assertion a : D. Ceci se traduit par deux appels aux prédicats **evolue /11** et **resolution /6**. Le résultat étant deux nouveaux noeuds de l'arbre de résolution.

Remarque : la résolution des deux noeuds doit être à $True$ pour que le prédicat **transformation_or /6** soit à $True$; soit il faut que les deux nouvelles Abe soient insatisfiables pour démontrer la proposition.

3.8 Prédicat affiche_evolution_Abox

Le prédicat **affiche_evolution_Abox /12** permet d'afficher l'évolution d'un état de la Abe vers un état suivant. Il prend ainsi en paramètre les six listes décrivant l'état de départ et les six autres décrivant l'état d'arrivée, cela comprend à la fois les assertions de concepts et de rôles de la ABox étendue. L'objectif est de permettre à l'utilisateur de suivre facilement le développement de l'arbre de démonstration.

Une exemple d'exécution de notre programme est présenté en Annexe.

CONCLUSION

La réalisation de ce projet nous a permis de développer un programme qui se base sur l'algorithme des tableaux pour démontrer une proposition en logique de description ALC. Nous avons pour cela suivi une méthodologie en trois étapes, l'objectif étant d'assurer la cohérence et la validité des informations traitées par notre programme.

Comme perspectives futures, il serait pertinent d'implémenter également le traitement d'une TBox simple et d'une TBox générale.

ANNEXE

Guide d'utilisation du programme

Les annexes de ce document incluent le code de notre programme, ainsi que les définitions de la TBox et de la ABox qui sont contenues dans le fichier **programme.pl**.

Note : Après chaque instruction entrée dans l'interpréteur Prolog, il faut ajouter un "**.**".

Pour lancer le programme, il suffit d'exécuter la commande **["programme.pl"].** dans un interpréteur Prolog tel que SWI-Prolog pour charger le fichier, puis d'exécuter **programme.**

Ensuite, l'utilisateur est invité à entrer la proposition à prouver. Conformément à la partie 2, nous avons 2 propositions à prouver.

Si l'utilisateur choisit de prouver une proposition comme celle de la section 2.1, il doit taper **"1."**. Ensuite, le programme demande d'introduire l'identificateur de l'instance (ex : vinci.). Après quoi, il est demandé d'introduire l'identificateur du concept ou sa définition (ex : and(auteur, sculpteur)).

Si l'utilisateur choisit de prouver une proposition comme celle de la section 2.2, il doit taper **"2."**. Puis, le programme demande d'introduire l'identificateur du premier concept ou sa définition (ex : auteur.) Une fois cela introduit, il est demandé d'introduire l'identificateur du deuxième concept ou sa définition (ex : éditeur.).

Un exemple du déroulement de notre programme est présenté dans la section "Test sur les prédicats de résolution".

Résultats du jeux de tests

Tests sur le prédicat concept

Avec les définitions de la ABox et de la TBox données dans le sujet du projet, notre prédicat nous affiche trois messages, chacun correspondant à la vérification de la correction sémantique, syntaxique et le test d'auto-référence.

Erreur sémantique

Lors de l'ajout de "cnamea(animal)." et "cnamena(animal)." dans les définitions de la ABox et de la TBox initiales, une erreur de correction sémantique apparaît car, pour respecter le principe d'unicité défini dans 1.1.1, nous ne pouvons pas avoir deux concepts, un atomique et un non atomique, avec le même nom.

Erreur syntaxique

Si nous ajoutons "inst(david,etudiant)." dans les définitions de la ABox et de la TBox initiales, une erreur de syntaxe se produira, car le concept "etudiant" n'est pas défini.

Test sur le prédicat pas_autoref

Pour avoir une auto-référence, nous ajoutons dans les définitions initiales "rname(creePar).", "cnamena(sculpture)." et "equiv(sculpture,and(objet,all(creePar,sculpteur)))." . Ensuite, nous supprimons "cnamea(sculpture).". Le résutat de l'exécution est à $false$, et le programme nous affiche un message d'erreur, car une auto-référence a été détectée.

Tests sur les prédicats traitement_TBox/1 et traitement_ABox/1

Pour vérifier si la simplification et la mise sous forme normale négative ont été effectuées, nous allons tester cela par rapport aux définitions de la ABox donnée dans le sujet du projet. Pour ce qui est de la TBox, nous avons changé les définitions suivantes :

equiv(sculpteur,and(personne,not(all(aCree,not(sculpture))))).

equiv(editeur_parent,and(parent,and(not(some(aEcrit,livre)),some(aEdite,livre)))).

Résultat :

TBox avant simplification et mise sous NNF :

[(auteur,and(personne,some(aEcrit,livre))),

(editeur_parent,and(parent,and(not(some(aEcrit,livre)),some(aEdite,livre)))),

(parent,and(personne,some(aEnfant,anything))),

(sculpteur,and(personne,not(all(aCree,not(sculpture)))))]

TBox après simplification et mise sous NNF :

[(auteur,and(personne,some(aEcrit,livre))),

(editeur_parent,and(and(personne,some(aEnfant,anything)),and(all(aEcrit,not(livre)),some(aEdite,livre)))),

(parent,and(personne,some(aEnfant,anything))),

(sculpteur,and(personne,some(aCree,sculpture)))]

ABox contenant les instances avant simplification et mise sous NNF :

[(david,sculpture),(joconde,objet),(michelAnge,sculpteur),(sonnets,livre),(vinci,personne)]

ABox contenant les instances après simplification et mise sous NNF :

[david,sculpture),(joconde,objet),(michelAnge,and(personne,some(aCree,sculpture))),

(sonnets,livre),(vinci,personne)]

ABox contenant les rôles :

[(michelAnge,david,aCree),(michelAnge,sonnets,aEcrit),(vinci,joconde,aCree)]

Comme nous pouvons le constater à travers l'affichage des listes en rouge, les simplifications ainsi que la mise sous forme normale négative ont été correctement effectuées.

Tests sur les prédicats acquisition_prop_type1/3 et acquisition_prop_type2/3

Pour tester le prédicat **acquisition_prop_type1/3**, toujours en suivant les définitions fournies, nous avons d'abord suivi les instructions du guide d'utilisation pour la saisie au clavier. Nous avons entré "1." afin de vérifier le prédicat. Pour notre test, nous avons ajouté l'instance "michelAnge." avec la définition "sculpteur.". Le résultat obtenu est le suivant :

ABox contenant les intances :

[(david,sculpture),(joconde,objet),(michelAnge,personne),(sonnets,livre),(vinci,personne),
(michelAnge,or(not(personne),all(aCree,not(sculpture))))]

Comme nous pouvons le constater, l'instance "michelAnge : not(sculpteur)" a été ajoutée, avec la simplification de "not(sculpteur)" également effectuée et mise sous forme normale négative.

Pour **acquisition_prop_type2/3**, nous avons suivi la même procédure en entrant "2." et en utilisant les mêmes données. Cette fois-ci, comme test, nous avons saisi "auteur." puis "editeur.". Le résultat obtenu est le suivant :

ABox contenant les intances :

[(david,sculpture),(joconde,objet),(michelAnge,personne),(sonnets,livre),(vinci,personne),
(inst1,and(and(personne,some(aEcrit,livre)),and(personne,and(all(aEcrit,not(livre)),some(aEdite,livre)))))]

Nous constatons qu'une nouvelle instance, "inst1", a été crée. Cette instance vérifie l'intersection entre les deux concepts non atomiques "auteur" et "éditeur", qui ont été simplifiés et mis sous forme normale négative.

Tests sur le prédicat tri_Abox/6

Pour pouvoir tester ce prédicat, nous avons choisi d'exécuter l'instruction suivante :

:- tri_Abox([(david,sculpture),(sonnets,livre),(michelAnge,or(not(objet),personne)),
(michelAnge,all(aCree,(sculpture))),(michelAnge,some(aEcrit,(livre))),
(michelAnge,and(auteur,sculpteur))],Lie,Lpt,Li,Lu,Ls).

Dans ce cas, nous avons choisi le contenu de notre ABox (la première liste passée en paramètre) de manière à vérifier le bon fonctionnement du prédicat sans avoir de logique particulière à l'intérieur de la définition de notre ABox.

Comme les résultats ci-dessous le montrent, notre prédicat a correctement associé chaque type d'instance à la liste correspondante. Par exemple, la liste "Lie" correspond aux assertions du type (I,some(R,C)), et l'unique assertion de ce type présente dans l'ABox de base a été correctement placée dans cette liste. De manière équivalente, nous observons cela pour les autres listes.

Résultat :

Lie = [(michelAnge, some(aEcrit, livre))],

Lpt = [(michelAnge, all(aCree, sculpture))],

Li = [(michelAnge, and(auteur, sculpteur))],

Lu = [(michelAnge, or(not(objet), personne))],

Ls = [(david, sculpture), (sonnets, livre)].

Tests sur le prédicat evolue/11

Afin de tester ce prédicat, nous avons exécuté les instructions suivantes :

:- evolue((david,and(objet,sculpture)),[],[],[],[],[],[],[],[],Li,[],[]).

:- evolue((david,and(objet,sculpture)),[],[],[(david,and(objet,sculpture)),[],[],[],[],Li,[],[]).

Le résultat obtenu est : Li = [(david, and(objet, sculpture))] dans les deux cas. Cela montre que si un concept n'existe pas déjà dans la liste correspondante à son type, alors il l'ajoute, sinon ne fait rien.

Ce prédicat fonctionne de la même façon pour les trois autres types de règles ; à savoir "Existe", "Pour tout", et "Union".

Test sur les prédicats de résolution

Dans cette partie nous allons tester les prédicats **resolution /6**, **deduction_all /6**, **transformation_and /6**, **complete_some /6**, **transformation_or /6** et **affiche_evolution_Abox /6**, étant donné qu'ils sont tous imbriqués les uns dans les autres.

Ainsi, nous affichons dans ce qui suit un déroulement de notre programme dans le cas où la démonstration est vérifiée et un autre dans le cas où la résolution échoue.

Notons que ces deux tests sont effectués sur les TBox et ABox données dans l'énoncé du projet.

Réussite de la résolution

En appliquant la méthode des tableaux nous vérifions si Michel-Ange est un sculpteur (proposition de type 1).

Remarque : Cet exemple utilise les prédicats **deduction_all /6**, **transformation_or /6**, **resolution /6** et **affiche_evolution_Abox /6**, ce qui nous permet de vérifier leur bon fonctionnement.

```
_____________________________________|            Etat de depart            |_____________________________________

david:sculpture
joconde:objet
michelAnge:personne
sonnets:livre
vinci:personne
michelAnge:¬personne⊔∀aCree.¬sculpture

<michelAnge,david>:aCree
<michelAnge,sonnets>:aEcrit
<vinci,joconde>:aCree


_____________________________________|            Etat d'arrivee            |_____________________________________

michelAnge:¬personne
david:sculpture
joconde:objet
michelAnge:personne
sonnets:livre
vinci:personne

<michelAnge,david>:aCree
<michelAnge,sonnets>:aEcrit
<vinci,joconde>:aCree
APPLICATION TRANSFORMATION OR => SEPARATION DANS L'ARBORESCENCE
_____________________________________|            Etat de depart            |_____________________________________

david:sculpture
joconde:objet
michelAnge:personne
sonnets:livre
vinci:personne
michelAnge:¬personne⊔∀aCree.¬sculpture

<michelAnge,david>:aCree
<michelAnge,sonnets>:aEcrit
<vinci,joconde>:aCree
_____________________________________|            Etat d'arrivee            |_____________________________________

david:sculpture
joconde:objet
michelAnge:personne
sonnets:livre
vinci:personne
michelAnge:∀aCree.¬sculpture

<michelAnge,david>:aCree
<michelAnge,sonnets>:aEcrit
<vinci,joconde>:aCree
_____________________________________|            Etat de depart            |_____________________________________

david:sculpture
joconde:objet
michelAnge:personne
sonnets:livre
vinci:personne
michelAnge:∀aCree.¬sculpture

<michelAnge,david>:aCree
<michelAnge,sonnets>:aEcrit
<vinci,joconde>:aCree


_____________________________________|            Etat d'arrivee            |_____________________________________

david:¬sculpture
david:sculpture
joconde:objet
michelAnge:personne
sonnets:livre
vinci:personne

<michelAnge,david>:aCree
<michelAnge,sonnets>:aEcrit
<vinci,joconde>:aCree

Youpiiiiii, on a demontre la proposition initiale !!!
true .
```

Le démonstrateur retourne une réponse positive car toutes les branches de l'arbre de résolution sont fermées, *Abe* est insatisfiable ce qui permet au programme d'affirmer que la proposition initiale est démontrée.

Echec de la résolution

En appliquant la méthode des tableaux nous vérifions si un sculpteur est un auteur (proposition de type 2).

Remarque : Cet exemple utilise les prédicats **transformation_and /6**, **complete_some /6**, **resolution /6** et **affiche_evolution_Abox /6**, ce qui nous permet de vérifier leur bon fonctionnement.

```
_____________________________________|            Etat de depart           |_____________________________________

david:sculpture
joconde:objet
michelÀnge:personne
sonnets:livre
vinci:personne
inst1:personne &  E aCree.sculpture & personne &  E aEcrit.livre

<michelÀnge,david>:aCree
<michelÀnge,sonnets>:aEcrit
<vinci,joconde>:aCree


_____________________________________|            Etat d'arrivee           |_____________________________________

david:sculpture
joconde:objet
michelÀnge:personne
sonnets:livre
vinci:personne
inst1:personne &  E aEcrit.livre
inst1:personne &  E aCree.sculpture

<michelÀnge,david>:aCree
<michelÀnge,sonnets>:aEcrit
<vinci,joconde>:aCree
```

...

```
_____________________________________|            Etat de depart           |_____________________________________

inst2:livre
inst1:personne
david:sculpture
joconde:objet
michelÀnge:personne
sonnets:livre
vinci:personne
inst1:  E aCree.sculpture

<inst1,inst2>:aEcrit
<michelÀnge,david>:aCree
<michelÀnge,sonnets>:aEcrit
<vinci,joconde>:aCree


_____________________________________|            Etat d'arrivee           |_____________________________________

inst3:sculpture
inst2:livre
inst1:personne
david:sculpture
joconde:objet
michelÀnge:personne
sonnets:livre
vinci:personne

<inst1,inst3>:aCree
<inst1,inst2>:aEcrit
<michelÀnge,david>:aCree
<michelÀnge,sonnets>:aEcrit
<vinci,joconde>:aCree
```

L'état d'arrivée montre que plus aucune règle ne peut s'appliquer, la situation du noeud en cours ne peut plus évoluer et il n'est donc pas possible de trouver un clash. Par conséquent, le programme retourne **false** et il y a échec de la résolution. le programme n'a pas pu démontrer la proposition initiale.