Rapport
-
Projet LRC

Binôme :

Shuyuan LUO
Maxence MAIRE

Introduction:

Ce projet vise à écrire un démonstrateur basé sur l'algorithme des tableaux pour la logique de description ***ALC***. Nous avons divisé le mécanisme de preuve en trois sections principales.

La première partie fait l'analyse sémantique et syntaxique des **Abox** et **Tbox** originales. On construit ensuite des listes des assertions des concepts et instances sous forme normale négative.

La deuxième partie consiste en la préparation de la preuve, qui vise à accepter la proposition à prouver. Il faut convertir la proposition originale en assertions sous forme normale négative ne contenant que des concepts atomiques. On les ajoute dans la **Abox** construite dans la première partie.

La troisième partie est le cœur du démonstrateur qui implémente l'algorithme de résolution basé sur la méthode des tableaux. On y construit un arbre **Abe** pour résoudre le problème initial, en appliquant les règles de résolution sur cet arbre.

Données:

On dispose de deux fichiers de données : **LRC_nnf.pl** qui contient des prédicats usuels donnés dans le sujet, et **LRC_Tbox_Abox.pl** qui contient des assertions de Abox et de Tbox inspirées de l'exercice 3 du TD4. On le chargera tout de suite pour tester le programme du démonstrateur dans Prolog.

Fichier LRC_Tbox_Abox.pl :

Contient des prédicats donnés dans le sujet du projet ainsi qu'une définition d'instances et de concepts, et des relations entre eux.

Fichier LRC_nnf.pl :

- **`concat(L1,L2,L)`** : concatène deux listes L1 et L2 dans L.
- **`enleve(X,L,LL)`** : enlever X de L, et renvoie le résultat dans LL.
- **`genere(Nom):`** génère une nouvelle instance dont le nom est Nom=instanceX, X un nombre qui changera au fur et à mesure que le nombre d'utilisations augmente.
- **`compteur(Num) :`** réalise le compteur.
- **`nombre(X,L1) :`** transforme un nombre X en une chaîne de caractères **`L1`**.
- **`chiffre_car(Num,C) :`** Transforme un numéro **`Num`** en un caractère **`C`**.
- **`my_flatten(L, Z)`** : linéarise une liste L pouvant potentiellement contenir des sous-listes et la renvoie dans Z.

Partie 1:

Le code de cette partie est contenu dans le fichier **`LRC_part1.pl.`** Pour exécuter cette partie, il suffit d'appliquer le prédicat **`premiere_etape.`**

L'analyse sémantique dans cette partie consiste à identifier si l'expression conceptuelle est bien une composition de concepts atomique et de rôles en utilisant le prédicat **`concept`**. Pour faire l'analyse syntaxique, il suffit d' utiliser le prédicat **`autoref`** pour vérifier que chaque concept non-atomique donné dans l'énoncé n'est pas autoréférent.

Après l'analyse syntaxique et sémantique, on passe pour la création des listes de Tbox et Abox d'après **`creation_Tbox, creation_Abox`** .
Puis les **`traitement_Abox`** et **`traitement_Tbox`** nous permettent de développer les concepts basés sur les concepts atomiques.

Fichier LRC_part1.pl :

- **`premiere_etape(TboxR, AboxR1, AboxR2)`** : exécute successivement la création et le traitement d'une Tbox et d'une Abox à partir des arguments passés au prédicat.

- **concept(ExprConcept)** : vérifie la correction sémantique de l'expression conceptuelle passée en paramètre (applique les règles **not**, **and**, **or**, **some** et **all**).
    - **instance(I)** : vérifie les identificateurs d'instance.
    - **role(R)** : vérifie les identificateurs de rôle.

- **autoref(ExprConcept)** : vérifie si le concept **C** est autoref ou pas .
    - **pas_autoref(CNA,CG)** : vérifie si le concept non-atomique **CNA** est autoref ou pas selon son expression générale **CG.**

- **creation_Tbox(L)** : crée une liste de Tbox **L** d'après la **Tbox** .

- **creation_Abox(L1, L2)** : crée deux listes de Abox,avec **L1** la liste des couples d'assertions de concepts sous forme **(I,C)** , et **L2** la liste des couples d'assertions de rôles sous forme **(I1,I2,R)**

- **traitement_Abox(AboxI, AboxR, AboxI2, AboxR2)** : remplace les identificateurs des concepts complexes de l'expression d'une Abox **AboxI, AboxR** par une expression d'identificateurs de concepts atomiques mise sous forme normale négative **AboxI2, AboxR2**.
    - **prolonge(A,L)** : développe un concept en le mettant sous forme d'expression de concepts atomiques.

- **traitement_Tbox(Tbox1, Tbox2)** : remplace les identificateurs des concepts complexes de l'expression d'une Tbox **Tbox1** par une expression d'identificateurs de concepts atomiques mise sous forme normale négative **Tbox2**.

Partie 2:

Le code de cette partie est dans le fichier **LRC_part2.pl.** Pour exécuter cette partie, il suffit d'appliquer le prédicat **deuxieme_etape.**

Il s'agit de la préparation de deux types de proposition I:C (Montrer que l'instance I appartient au concept C) et $C1 \sqcap C2 \sqsubseteq \bot$ (Les concepts C1 et C2 ont une intersection vide).

Pour le premier cas, il est nécessaire d'effectuer le même traitement que partie 1 sur le concept $\neg C$, ie. ajouter $(I,\neg C)$ dans la liste de Abox. On réalise cette action en appliquant **acquisition_prop_type1.**

Pour le deuxième cas, il faut générer une instance inst, puis ajouter l'assertion (inst, C1 ⊓ C2) la liste de Abox. On réalise cette action en appliquant **`acquisition_prop_type2.`**

Le prédicat **`suite`** nous permet de vérifier si le numéro et la proposition entrés correspondent à programme exécutable, ie. le numéro **`R ∈ {0,1}`** et la proposition est correcte d'après l'analyse syntaxique et sémantique. Sinon, on vous demandera d'entrer encore une fois le numéro et l'expression de la proposition.

Fichier LRC_part2.pl :

- **`deuxieme_etape`** ; **`saisie_et_traitement_prop_a_demontrer`** ; **`suite`** : données dans le sujet ; fonctions structurant le traitement des requêtes entrées par l'utilisateur.

- **`acquisition_prop_type1(Abi,Abi1,Tbox)`** : permet à l'utilisateur du programme de rentrer une formule de type [instance, concept], dans le but de tester si instance appartient à concept (test réalisé dans la fonction **`deuxieme_etape`**).
  - **`prolonge_A_Tbox`** : prédicat permettant d'identifier les arguments entrés par l'utilisateur et les développe pour les rendre utilisables.
  - **`ajouter1(I,C,Abi,Abi1,Tbox)`** : traite les données de **`acquisition_prop_type1`** développées en les mettant sous forme normale négative, vérifiant l'appartenance de I à C.

- **`acquisition_prop_type2(Abi,Abi1,Tbox)`** : permet à l'utilisateur du programme de rentrer une formule de type [concept1, concept2], dans le but de tester si concept1 et concept2 ont des éléments en commun (test réalisé dans la fonction **`deuxieme_etape`**).
  - **`ajouter2(C1,C2,Abi,Abi1,Tbox)`** : traite les données de **`acquisition_prop_type2`** développées en les mettant sous forme normale négative, vérifiant l'intersection des concepts C1 et C2.

Partie 3:

Le code de cette partie est dans le fichier **`LRC_part3.pl.`** Pour exécuter cette partie, il suffit d'appliquer **`troisieme_etape.`**

Pour mettre en œuvre la méthode des tableaux, nous devons d'abord construire un arbre en utilisant **`Aboxr`** et **`Aboxi`**. On ne s'oriente que sur les derniers nœuds **`Abe`** de cet arbre pour la suite.

Pour résoudre les concepts complexes, le prédicat **resolution** applique récursivement les règles **SOME, AND, ALL** et **OR** sur les listes **Lie, Lpt, Li, Lu, Ls** obtenues par **tri_Abox,** jusqu'à ce que l'on trouve un clash dans **Abe**, ou que l'on a une situation où le noeud en cours ne peut plus évoluer (cas où **Lie, Lpt, Li, Lu** sont tous vides). Les assertions sont affichées par le prédicat **affiche_evolution_Abox** après chaque application des règles **SOME, AND, ALL** et **OR** .

Il faut noter que toutes les règles écrites ajoutent sur **Abe** au lieu de modifier directement les listes **Lie, Lpt, Li, Lu, Ls**; on doit donc réaliser chaque fois une répartition **separate_ABR** et une reconstruction **tri_Abox** pour travailler sur les nouvelles listes **Lie1, Lpt1, Li1, Lu1, Ls1.** (cette action correspond au prédicat **evolue** dans le sujet).

Fichier LRC_part3.pl :

- **troisieme_etape** : donnée dans le sujet ; fonctions structurant la démonstration de la proposition entrée (cf **deuxieme_etape**).

- **resolution(Lie, Lpt, Li, Lu, Ls, Abe**: applique **test_collision** pour signaler s'il y a une collision, et s'il n'y en a pas, essaie d'appliquer les règles de **SOME, AND, ALL** et **OR** définies au-dessus, si on n'arrive pas, retourner **'Fail to match any rule'.**
  - **test_collision(Abe)** : prédicat vérifiant s'il y a un clash dans **Abe**, ie. l'instance I de **Abe** appartient à la fois à C et à non(C).

- **evolue(Abe,Lie1, Lpt1, Li1, Lu1, Ls1)** : utilise **separate_ABR** et **tri_Abox** pour obtenir les nouvelles listes **Lie1, Lpt1, Li1, Lu1, Ls1** d'après **Abe**.
  - **separate_ABR(Abe,Abi,Abr**) : sépare **Abe** pour obtenir **Abr,Abi** (qui sert à executer **tri_Abox**).
  - **tri_Abox(Abi, Lie, Lpt, Li, Lu, Ls)** : classe les formules de Abox dans les 5 listes suivantes : Lie pour les formules de type (I, some(R, C)); Lpt pour (I, all(R, C)) ; Li pour (I, and(C1, C2)) ; Lu pour (I, or(C1, C2)).
  - **create_ABR(Abr, Abi, Abe**) : concatène les paramètres **Abr,Abi** pour créer **Abe**.

- **complete_some(Lie, Lpt, Li, Lu, Ls, Abr, Abe)**: prédicat appliquant la règle **SOME** sur un élément qu'il cherche dans la liste Lie.

- **`transformation_and(Lie, Lpt, Li, Lu, Ls, Abr, Abe)`** : applique la règle **`AND`** sur un élément qu'il cherche dans la liste Li.

- **`deduction_all(Lie, Lpt, Li, Lu, Ls, Abr, Abe)`** : applique la règle **`ALL`** sur un élément qu'il cherche dans la liste Lpt.

- **`transformation_or(Lie, Lpt, Li, Lu, Ls, Abr, Abe)`** : applique la règle **`OR`** sur un élément qu'il cherche dans la liste Lu.

- **`affiche_evolution_Abox(Abe1,Abe2)`**: affiche la transformation de **`Abe1`** **`à Abe2`**
  - **`print_concepte(CG)`**: affiche le concept **`CG`**
  - **`print_Abox(Abe)`**: affiche les assertions dans **`Abe`**

Conclusion:

Les trois parties du programme fonctionnent bien ensemble pour créer des structures de Abox et Tbox, puis appliquer l'algorithme des tableaux sur ces boîtes dans le but de faire une démonstration de la proposition initialement entrée par l'utilisateur.
Le programme est donc maintenant un démonstrateur fonctionnel.

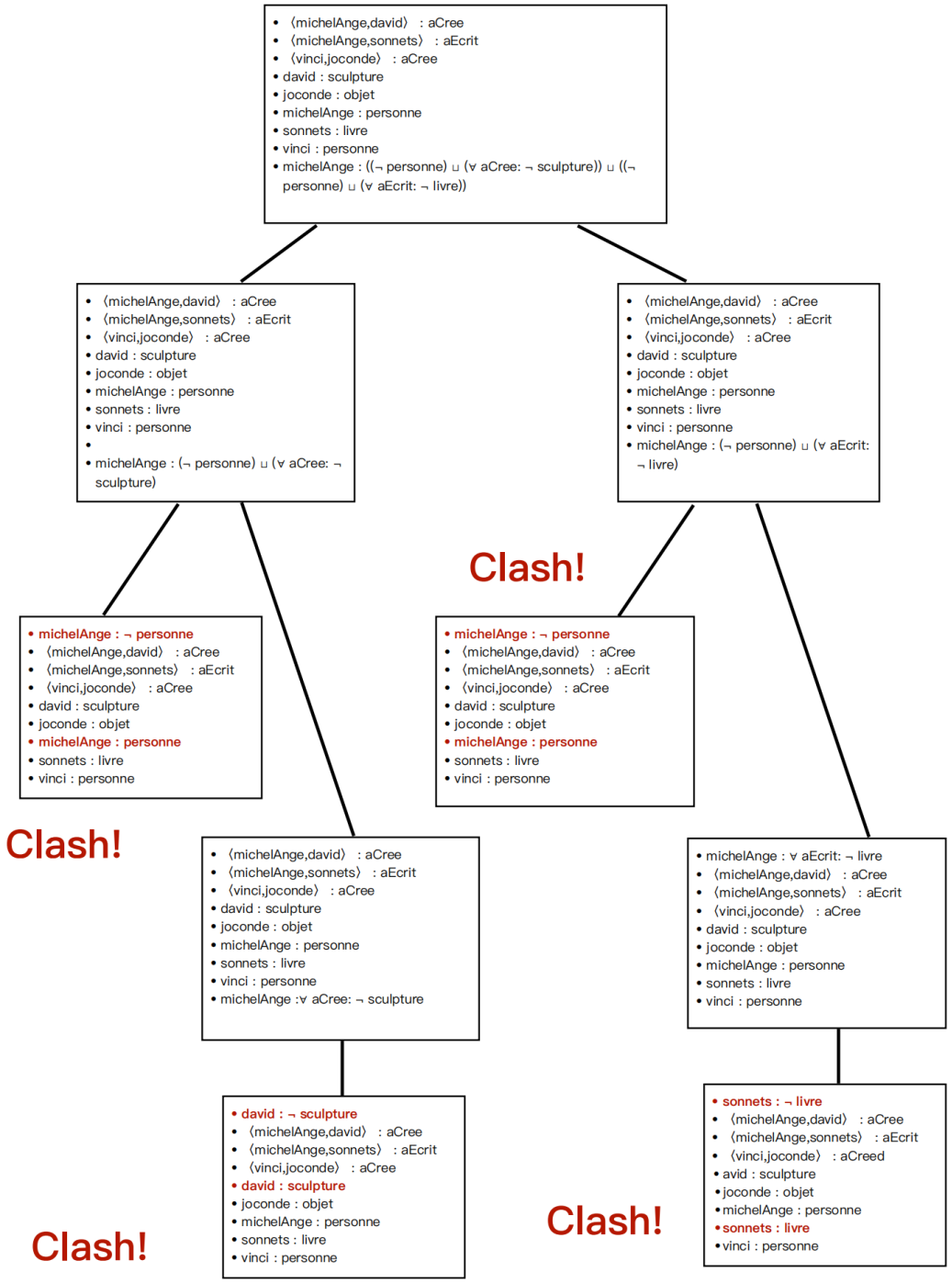
Nous pouvons donc, avec cet algorithme, prouver que n'importe quelle proposition (du type accepté par le démonstrateur) est valide en utilisant la méthode des tableaux pour la logique de description ***`ALC`*** de manière automatique.

Voir application de l'algorithme sur une proposition exemple, avec modélisation de l'arbre créé, sur la page suivante.

Exemples :

On teste sur proposition **michelAnge: sculpteur ⊓ auteur**, donc on ajoute **michelAnge:¬(sculpteur ⊓ auteur)** dans l'Abox, voici le résultat:

Arbre initial

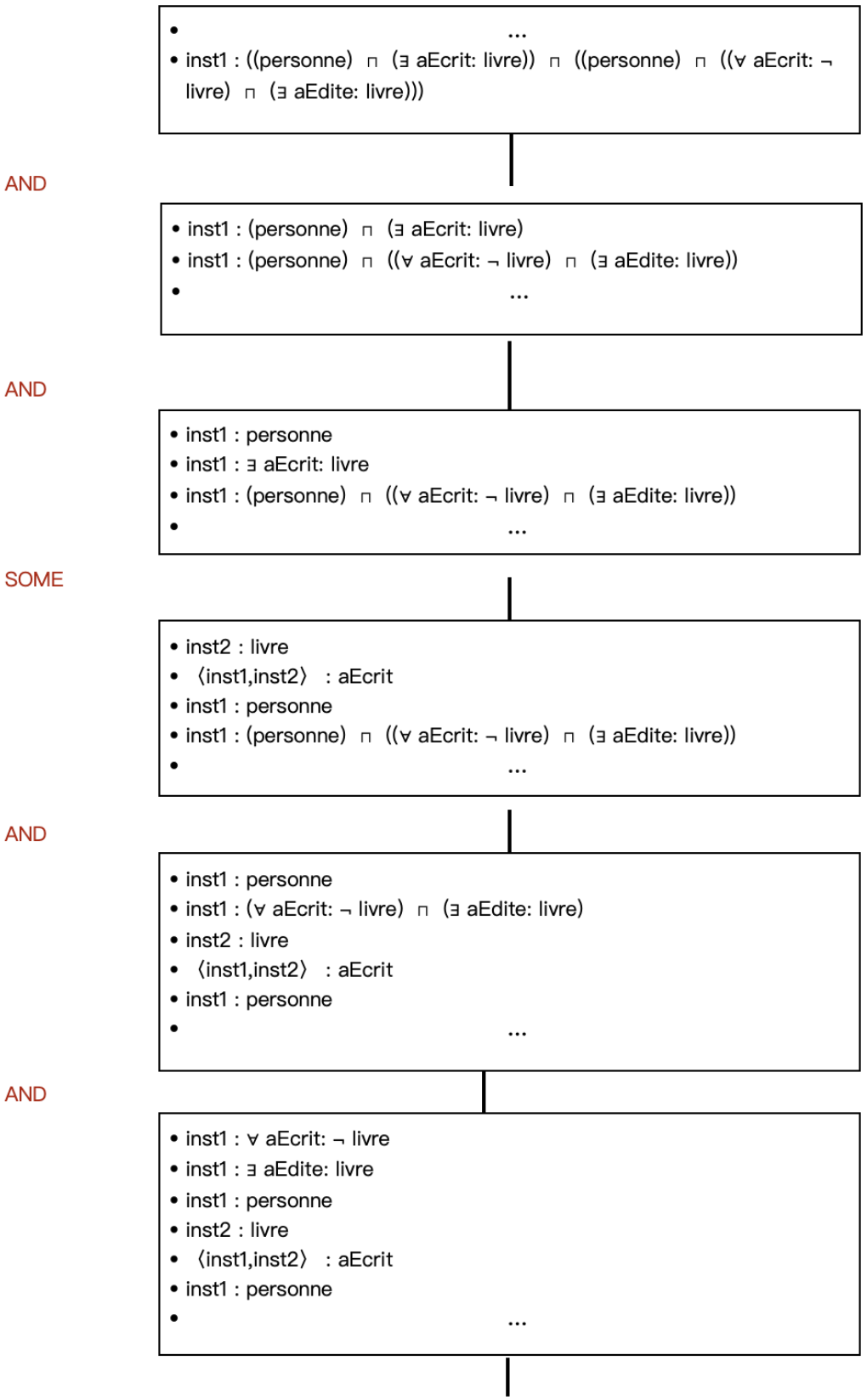


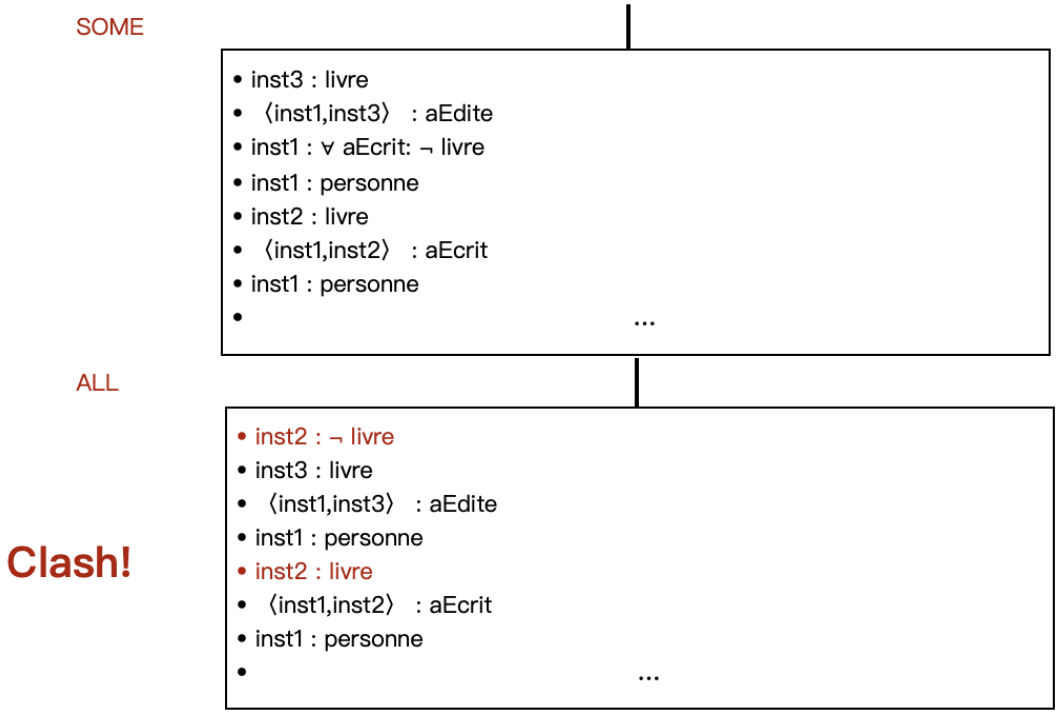
On a obtenu 4 clashs, **michelAnge:¬(sculpteur ⊓ auteur)** est donc insatisfiable.

La proposition initiale **michelAnge:sculpteur ⊓ auteur** est donc valide.

On teste sur **auteur ⊓ editeur** ⊑⊥, donc on ajoute **inst1:auteur ⊓ editeur** dans l'Abox, voici le résultat:

Arbre initial





Posons les trois points… qui signifie l'ensembles des assertions:

⟨michelAnge,david⟩ : aCree
⟨michelAnge,sonnets⟩ : aEcrit
⟨vinci,joconde⟩ : aCree
david : sculpture
joconde : objet
michelAnge : personne
sonnets : livre
vinci : personne

On a obtenu enfin un clashs, **`inst1:auteur` ⊓ `editeur`** est donc insatisfiable.

La proposition initiale **`auteur` ⊓ `editeur`** $\sqsubseteq \bot$ est donc valide.