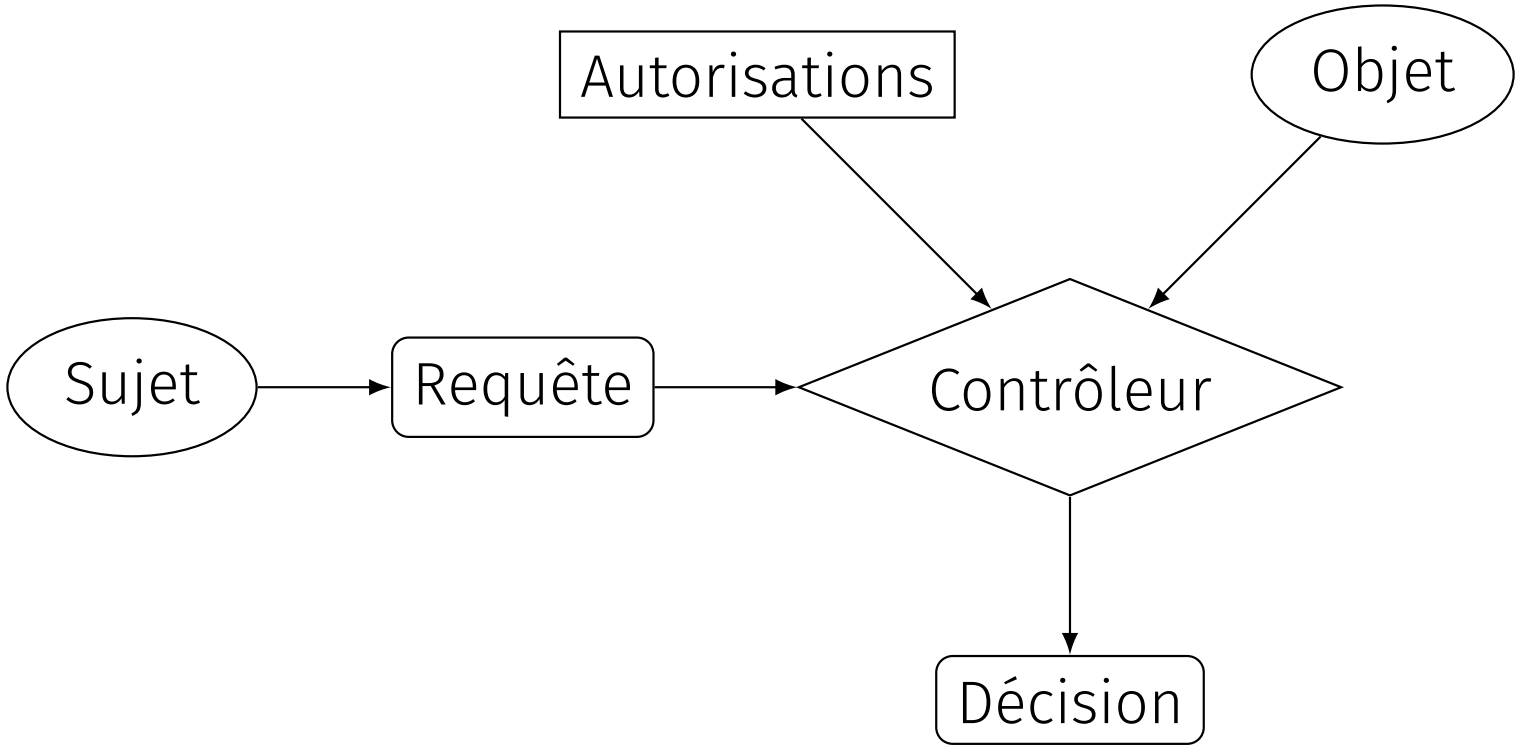
Programmation logique et contrôle d'accès

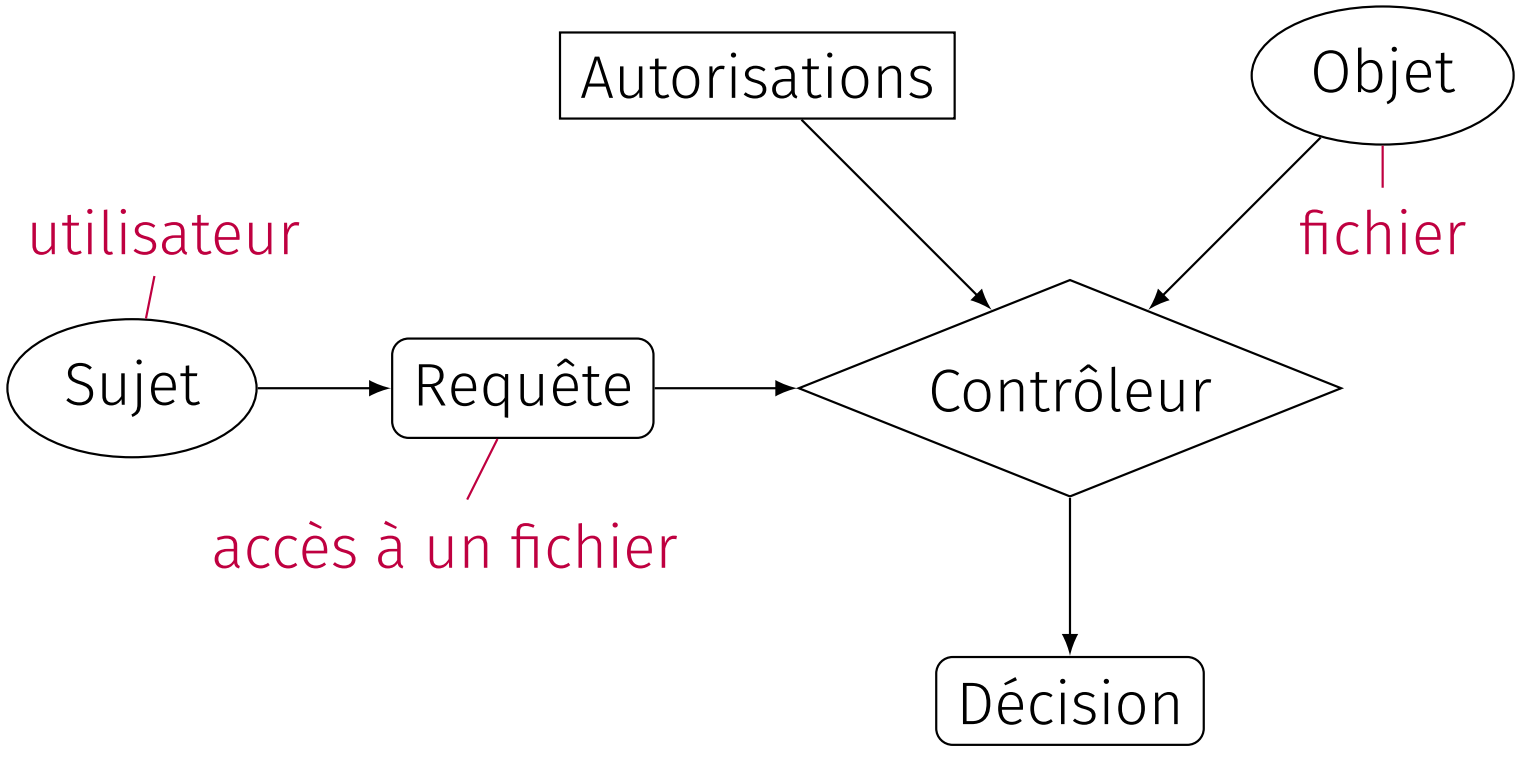
Samuel Gallay

21 novembre 2020

Contrôle d'accès : motivation



Contrôle d'accès : motivation



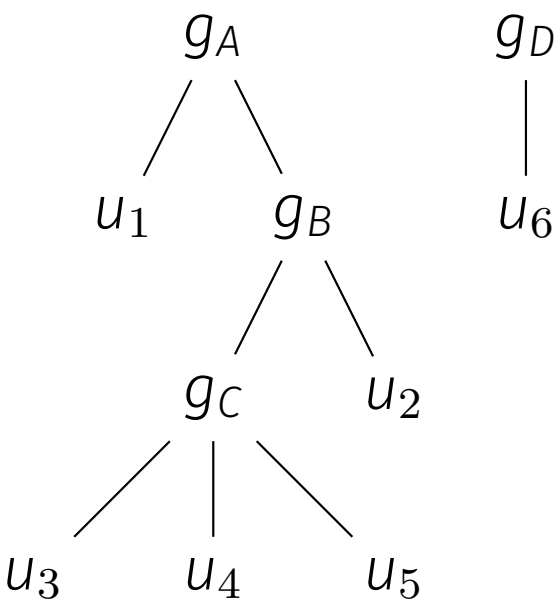
Matrice de contrôle d'accès : limitations

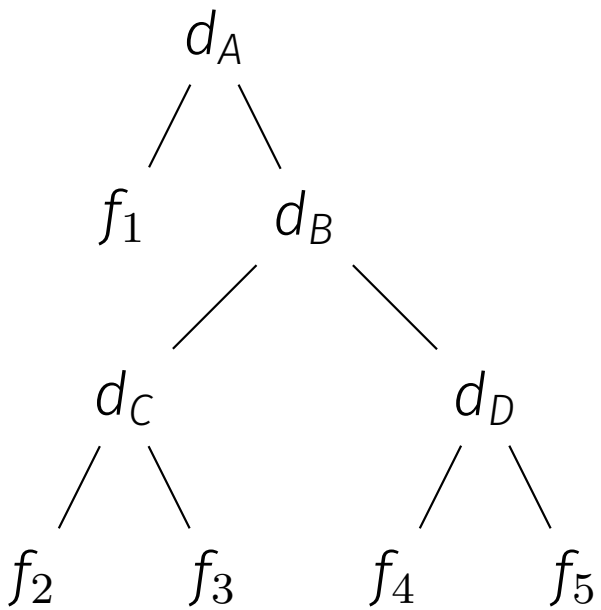
Fichier \ Utilisateur	Alice	Bob	Charlie	...
Fichier 1	1	1	0	...
Fichier 2	0	1	1	...
Fichier 3	1	0	0	...
...	...	...	...	

Inconvénients

- taille de la matrice
- suppression des autorisations

Arborescence de fichiers et groupes d'utilisateurs





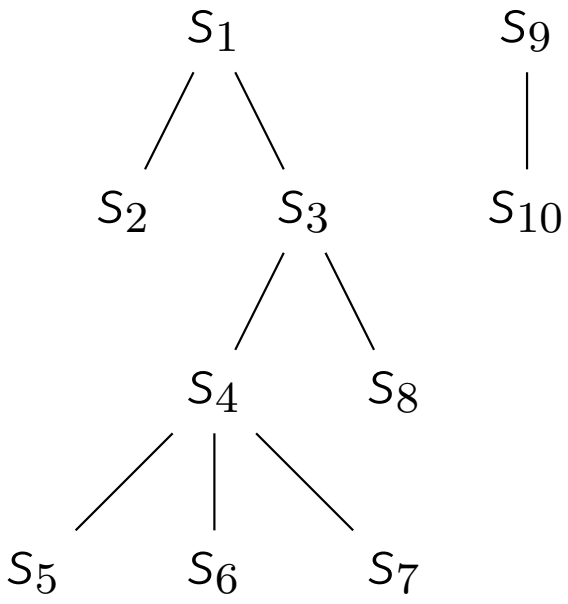
u_i : utilisateur
g_i : groupe

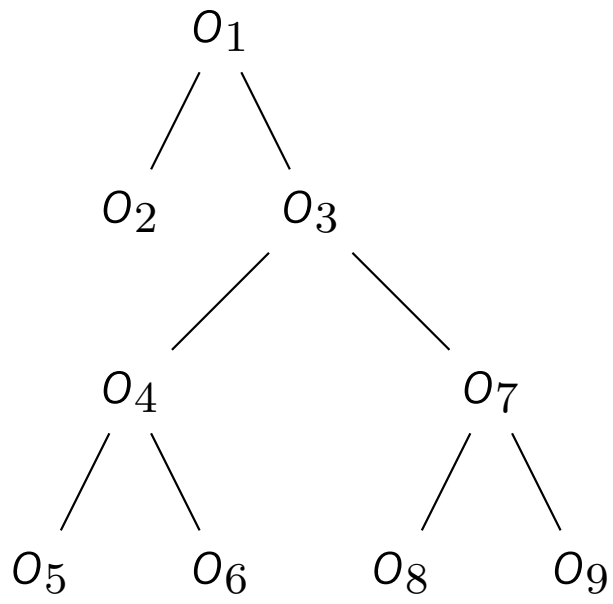
$appartient_{gr}(u_3, g_C)$
$inclus_{gr}(g_B, g_A)$

f_i : fichier
d_i : dossier

$appartient_{dos}(f_4, d_D)$
$inclus_{dos}(d_C, d_B)$

Arborescence de fichiers et groupes d'utilisateurs





s_i : sujet

o_i : objet

$sous\_sujet(s_2, s_1)$

$sous\_objet(o_2, o_1)$

Les règles d'autorisation

- Règles simples :

$$autorise(s_i, o_j), \text{ ou } sous\_sujet(s_i, s_j)$$

- Propagation des autorisations :

$$\forall U, \forall G, \forall D,$$
$$sous\_sujet(U, G) \wedge autorise(G, D) \Rightarrow autorise(U, D)$$

$$\forall F, \forall D, \forall U,$$
$$sous\_objet(F, D) \wedge autorise(U, D) \Rightarrow autorise(U, F)$$

Objectifs

Écrire un système permettant :

1. de représenter ce type de règles
2. de répondre à des questions comme :
   - *Alice peut-elle accéder au fichier A ?*
   - *Qui est-ce qui peut accéder au fichier A ?*

Logique du premier ordre, ou calcul des prédicats

- Des prédicats, des variables, des fonctions, les connecteurs logiques $\vee$ et $\wedge$, et les quantificateurs $\exists$ et $\forall$.
- Les règles sont des *formules bien définies* dont toutes les variables sont liées (formules clauses).
- Mise sous forme prénexe.
- Skolémisation.
- Mise sous forme normale conjonctive.

Logique du premier ordre, ou calcul des prédicats

Les règles sont maintenant toutes de la forme :

$$\forall X_1 \ldots X_s, \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{k} L_{i,j} \right)$$

En ne notant plus les quantificateurs, et en séparant en n clauses, on se réduit à la forme :

$$\bigvee_{i=1}^{k} L_i$$

où les L_i sont des littéraux positifs ou négatifs.

Restriction aux clauses définies

Attention

Perte de généralité par rapport à la logique du premier ordre : existence d'algorithmes efficaces sur les clauses de Horn.

Exactement un littéral positif :

$$\left(\bigvee_{i=1}^{n} \neg P_i\right) \vee Q \equiv \bigwedge_{i=1}^{n} P_i \Rightarrow Q$$

Si $n = 0$: un fait, Q toujours vrai.

Notation Prolog :

```
q(f1(X), ...) :- p1(f2(Y), ...), ..., pn(fk(Z), ...).
```

Exemple de programme Prolog

```
apprend(eve, mathematiques).
apprend(benjamin, informatique).
apprend(benjamin, physique).
enseigne(alice, physique).
enseigne(pierre, mathematiques).
enseigne(pierre, informatique).

étudiant_de(E,P):-apprend(E,M), enseigne(P,M).

étudiant(E, pierre) ?
```

La SLD-Résolution

$$\frac{\neg(A_1 \wedge A_2 \wedge ... \wedge A_n) \qquad B_1 \leftarrow B_2 \wedge ... \wedge B_k}{\neg(B_2 \wedge ... \wedge B_k \wedge A_2 \wedge ... \wedge A_n)\theta_1}$$

Avec $\theta_1 = \text{MGU}(A_1, B_1)$.

Si $\neg(A_1 \wedge ... \wedge A_n) \Rightarrow \textit{Faux}$, alors $(A_1 \wedge ... \wedge A_n)\theta_1$ est *Vrai*.

Unification : substitution des variables de deux termes. Ex :
`étudiant(E, P) = étudiant(E, pierre)`

Il existe un unifieur plus général.

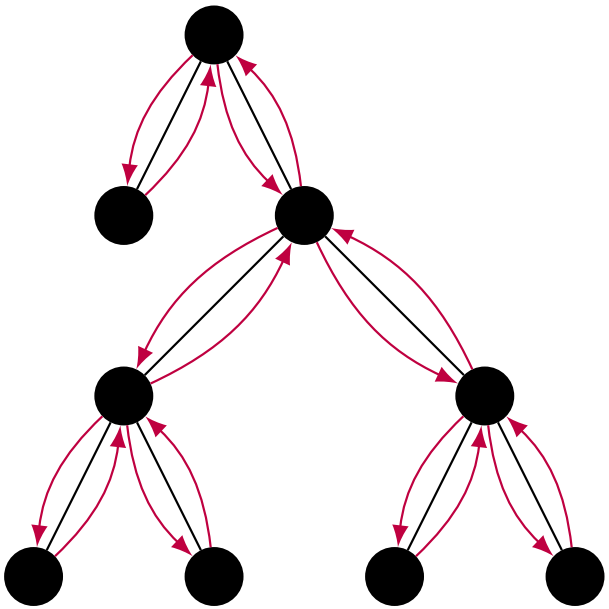
Algorithme d'unification

```
E = {étudiant_de(E,P) = étudiant_de(E, pierre)}.

Répéter tant que E change :
    Sélectionner une équation  s = t dans E;
    Si s = t est de la forme :
        f(s1, ..., sn) = f(t1, ..., tn) avec n >= 0
            Alors remplacer l'équation par s1=t1 ... sn=tn
        f(s1, ..., sm) = g(t1, ..., tn) avec f != g
            Alors ÉCHEC
        X = X
            Alors supprimer l'équation
        t = X où t n'est pas une variable
            Alors remplacer l'équation par X = t
        X = t où X != t et X apparait plus d'une fois dans E
            Si X est un sous-terme de t Alors ÉCHEC
            Sinon on remplace toutes les autres occurences de X par t
```

Termine et est correct.

Le retour sur trace



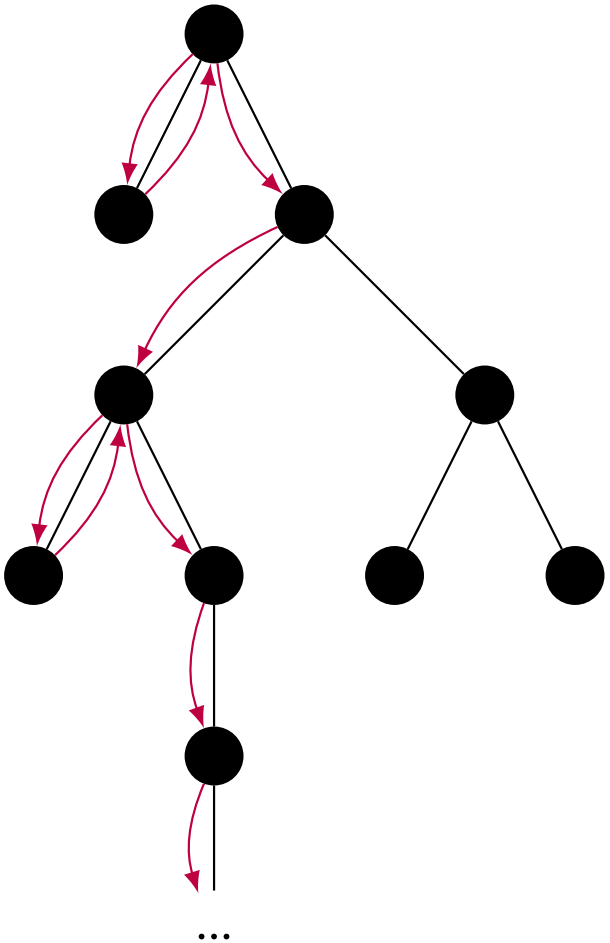
Il faut choisir dans quel ordre utiliser les clauses !

Solution utilisée en Prolog : parcours en profondeur.

Avantage

Faible cout en mémoire.

Le retour sur trace



Il faut choisir dans quel ordre utiliser les clauses !

Solution utilisée en Prolog : parcours en profondeur.

Avantage

Faible cout en mémoire.

Inconvénient

Risque manquer des solutions (boucles infinies).