INSA Lyon
20, avenue Albert Einstein
69621 Villeurbanne Cedex

Livrable de projet

Modélisation Cognitive

« Systèmes à Base de Connaissances : Évaluation de torts »

du 29 novembre au 11 décembre 2013

Hexanôme ***H4404*** *:*
Guillaume Abadie
Louise Crépet
Aline Martin
Martin Wetterwald

Enseignants :
Sylvie Calabretto
Mehdi Kaytoue

Année scolaire 2013-2014

Sommaire

1. Application

1.1 Noyau

Nous avons eu l'opportunité d'implémenter notre système expert à l'aide de `SWI-Prolog`, le langage `Prolog` étant particulièrement bien adapté à ce type d'application. L'avantage de `Prolog` est l'utilisation des prédicats pour l'implémentation des règles que l'expert doit appliquer. Considérons par exemple :

D'abord, je regarde si un des conducteurs a fait une faute grave. De deux choses l'une : soit la case qui correspond aux cas graves — la case 17 « N'avait pas observé un signal de priorité ou un feu rouge » — a été cochée, soit dans les observations manuscrites il est signalé une infraction du type : non-respect d'un stop, d'un panneau d'interdiction de dépasser ou d'un sens interdit. Dans ce cas, le conducteur a tous les torts : 100%.

Nous pouvons déduire la règle :

$$A_{17} \Rightarrow A_{torts} = 100\% \cdot B_{torts} = 0\%$$

Alors, on en tire le prédicat :

```
reportEvaluateFatalMistake(A,_,100) :-
    reportIsChecked(A,c17).
```

Néanmoins, nous avons voulu que le noyau utilise les règles de manière déterministe. De fait, nous avons créé un prédicat dynamique listant l'ensemble des règles de l'expert qu'il parcourt ensuite à l'évaluation des torts :

```
:- dynamic reportEvaluateRules/1.
:- retractall(reportEvaluateRules(_)).

reportDefineRule(RulePredicate) :-
    reportEvaluateRules(RulePredicate) -> (
        true
    ); (
        assert(reportEvaluateRules(RulePredicate))
    ).
```

Ainsi, il nous faut simplement dire au noyau de considérer la règle `reportEvaluateFatalMistake/3` :

```
:- reportDefineRule(reportEvaluateFatalMistake).
```

Et si les 2 conducteurs commettent une infraction grave ? C'est rare, mais dans ce cas, on partage les torts : 50% chacun.

Nous aurions pu implémenter une règle de la forme suivante :

```
reportEvaluateFatalMistake50(A,B,50) :-
    reportIsChecked(A,c17),
    reportIsChecked(B,c17).
```

Mais cela duplique le code pour chaque règle, et peut être la cause d'une erreur. Toutefois, le noyau va tester :

```
reportEvaluateFatalMistake(A,B,TortsA).
```

Mais aussi :

```
reportEvaluateFatalMistake(B,A,TortsB).
```

Alors, si les deux prédicats sont vérifiés, le noyau en déduit automatiquement :

$$A_{torts} = 50\% \cdot B_{torts} = 50\%$$

1.2 Tests unitaires

À chaque implémentation d'une règle, un test unitaire est codé : il construit un rapport avec certaines cases cochées, et on donne la règle qui doit s'exécuter.

En cas de collision, la procédure qui exécute tous les tests s'arrête, et donne la règle qui a effectivement été utilisée : soit la nouvelle règle est redondante et est supprimée, soit l'une des règles doit être précisée. Généralement, on lui ajoute le prédicat `not(autreRegle)`.

Par exemple, la règle 132 qui donne les torts quand l'accident a eu lieu alors que les deux véhicules roulaient en sens inverse entrait en collision avec la règle 131 qui donne les torts si l'accident se produit parce que l'un des véhicules a franchi l'axe médian, alors que les deux roulaient en sens inverse. La règle 132, qui avait cette forme :

```
reportRule132(A,B,50) :-
    reportReversedWays(A,B).
```

Est devenue :

```
reportReversedWays(A,B),
    not(reportRule131(A,B,_)).
```

La majorité des tests unitaires implémentés sont les exemples de constats fournis avec le sujet. Nous les avons complétés avec d'autres tests appelant d'autres règles pour vérifier de manière exhaustive leur validité.

2. Interface utilisateur

Le langage prolog ne permet pas d'implémenter naturellement une interface utilisateur graphique. Il s'utilise principalement en ligne de commande. Notre interface est donc uniquement textuelle. Cela pose des limites quant à notre mode de représentation dzns la mesure où un constat et normalement un document papier de type formulaire avec des cases à cocher.

2.1 Ligne de commandes

Il est demandé à l'utilisateur de renseigner 4 champs différents.
- "cocher" les cases de A
- "cocher" les cases de B
- demander confirmation du résultat
- demander le vrai résultat si celui calculé ne convient pas

Nous avons considéré que la partie vérification du nombre de case aura été gérée en amont par l'utilisateur de ce programme. Nous n'avons donc pas implémenté la fonction qui demande celle-ci et qui la vérifie.

L'interface pour "cocher" les cases est réduite à une simple chaine de caractère. L'utilisateur doit lister en un coup toutes les cases qu'il veut cocher pour un conducteur donné. Pour l'aider un recaptiulatif lui rappelle pour chaque case à quelle situation elle réfère.

Pour pouvoir prendre en compte les observations complémentaires qui influencent également le calcul des torts nous avons choisit de considérer les informations pertinentes comme de nouvelles "cases à cocher". Ainsi nous avons 22 cases au lieu des 17 de base. Celles-ci sont condisérées comme des cases normales et entrent dans la liste au même titre que les autres. L'interface n'exige pas d'ordre particulier pour remplir les cases.

A la page suivante, un exemple de l'interface.

```
c01 En stationnement / a l arret
c02 Quittait un stationnement / ouvrait une portiere
c03 Prenait un stationnement
c04 Sortait d un parking, d un lieu prive, d un chemin de terre
c05 S engageait dans un parking, dans un lieu prive, dans un chemin de terre
c06 S engageait sur une place a sens giratoire
c07 Roulait dans une place a sens giratoire
c08 Heurtant a l arriere, en roulant dans le meme sens et sur une meme file
c09 Roulait dans le meme sens et sur une file differente
c10 Changeait de file
c11 Doublait
c12 Virait a droite
c13 Virait a gauche
c14 Reculait
c15 Empietait sur une voie reservee a la circulation en sens inverse
c16 Venait de droite (dans un carrefour
c17 N avait pas observe un signal de priorite ou un feu rouge
c20 Etait en stationnement irregulier en agglomeration, pas le long d un trottoir
c21 Etait en stationnement irregulier hors agglomeration
c22 Avait franchi la ligne continue
c23 Fleche orange clignotante
c24 Avait franchi l axe median
c25 Roulait en sens inverse

conducteur A quelles cases ? (pensez a mettre les "" dans votre reponse):
|:

"c08 c11".
```

Le programme vérifie si les cases entrées correspondent bien à un des numéros listés, sinon il affiche un message d'erreur et s'arrête. L'utilisateur peut également n'entrer aucune case.

Une fois le résultat présenté, l'interface questionne le lecteur s'il est satifait du resultat en lui demandant d'entre y ou n. Le programme boucle tant qu'une de ces deux réponses n'est pas donnée.

Si l'utilisateur répond oui, l'exécution se termine. S'il répond non le programe fera sa maintenance semi-automatique.

2.2 Maintenance semi-automatique

Pour comprendre le système de maintenance semi-automatique, il faut expliquer le fonctionnement de prolog.

Prolog fonctionne par système de règles, posées dans des fichiers de code. Il est possible de déclarer de manière générique une règle en posant sa structure (nom, nombre d'arguments, etc.), puis d'en implémenter plusieurs à partir d'elle ensuite.

Nous avons construit une base de fait partir d'une règle qui permet de déclarer d'autres règles.

```
reportDefineRule(RulePredicate)
```

où les RulePredicate prennent la forme de

```
reportRuleZ(A,B,X)
```

où Z est le nom de la règle particulière
A les cases cochées par le conducteur A
B les cases cochées par le conducteur B
et X le tort calculé de A

Lorsqu'un utilisateur conteste une évaluation des torts, l'interface lui demande d'entrer les nouveaux torts que celui-ci attendait. Le programme créera ensuite une nouvelle règle nommée exceptionY (Y un numéro qui s'incrémente) qui sera testée avant toute les autres. Ainsi pour le cas particulier qui a ammené l'utilisateur à contester le calcul il y aura maintenant une règle propre.

2.3 Réflexion sur une maintenance complètement automatique

L'avantage de `Prolog` réside dans sa conception même. En effet, sa capacité de recherche de solutions vérifiant des prédicats pourrait être utilisée pour rechercher des cas particuliers de formulaire non déterministe. Alors nous pourrions imaginer que le système résoudrait tout d'un coup en posant les questions à l'expert pour chacune. Ainsi il serait ensuite impossible qu'un formulaire ne puisse pas être déterminé.

Grâce au développement de ce système, nous pourrions substituer l'expert et ainsi réaliser un investissement. Seule la durée du TP a pu nous empêcher de le prouver dans notre implémentation.

3. Choix du language Prolog