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**

1.1 Noyau 1

1.2 Tests unitaires 2

2 Interface utilisateur **3**

2.1 Ligne de commandes 3

2.2 Maintenance semi-automatique 3

2.2.1 Principe général 3

2.2.2 Implémentation 3

2.2.3 Cas limite 4

2.3 Réflexion sur une maintenance complètement automatique 4

3 Choix du langage `Prolog` **6**

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

2.1 Ligne de commandes

2.2 Maintenance semi-automatique

Nous avons été amenés à implémenter un mécanisme de maintenance semi-automatique dans notre système expert. Cela correspond au dispositif permettant à l'expert d'apprendre un nouveau cas particulier à notre système.

2.2.1 Principe général

Le processus se déroule comme suit :
- l'utilisateur coche les cases du constat pour la voiture A et la voiture B ;
- notre système détermine les torts avec les règles actuelles ;
- le système demande à l'utilisateur (l'expert) si ces torts sont valides. Si l'expert est satisfait, il répond « oui » et le système ne change pas son comportement. Dans le cas contraire, notre système enregistre la nouvelle règle.

2.2.2 Implémentation

Maintenant que nous avons vu le principe général, intéressons-nous au détail de l'implémentation.

Lorsque notre système ne renvoie pas les bons torts et que l'expert indique les torts de A, il est important que notre système enregistre la nouvelle règle comme étant **prioritaire** par rapport aux règles standards. Ainsi, elle se déclenchera avant celles-ci.

Nous utilisons le mécanisme de **prédicat dynamique** que nous fournit `Prolog` pour implémenter la maintenance semi-automatique.

```
:- dynamic reportEvaluateWrongsPriorDB/3.
:- retractall(reportEvaluateWrongsPriorDB(_, _, _)).
```

Le prédicat `reportEvaluateWrongsPriorDB/3` constitue la base de données de règles prioritaires que l'expert a rentrées lors de la maintenance semi-automatique. Au début, cette base est vide, et seules les règles standards sont utilisées pour déterminer les torts. Les paramètres de ce prédicat correspondent respectivement aux cases à cocher de A,

cases à cocher de B, et tort de A.

Il suffit ensuite de faire en sorte que ce prédicat dynamique soit essayé en premier, avant les prédicats statiques. Cela est implémenté comme suit :

```
reportEvaluate(ReportA,ReportB,WrongsA,Evaluator) :-
    reportPrune(ReportA,ReportB,NewReportA,NewReportB) ->
        (
                reportEvaluateWrongsPrior(NewReportA, NewReportB, WrongsA, Evaluator) ;
                reportEvaluateWrongs(NewReportA,NewReportB,WrongsA,Evaluator)
        ).
```

Il apparaît clairement dans ce code que `reportEvaluateWrongsPrior/4` est essayé avant `reportEvaluateWrongs/4`. `reportEvaluateWrongsPrior/4` n'est en fait qu'une surcouche du prédicat dynamique `reportEvaluateWrongsPriorDB/3` permettant de gérer la symétrie entre A et B.

2.2.3 Cas limite

Lors de la maintenance semi-automatique, il existe le cas limite que l'expert se contredise lui-même, c'est-à-dire qu'il rentre un même état de cases à cocher pour A et B, mais avec des torts différents. Cela rompt l'automatisme du système car notre système a donc stocké deux répartitions des torts différentes pour un cas qui semble être le même, et la réponse n'est plus unique. Il y a deux approches pour régler ce problème.

— On peut considérer que l'expert doit être cohérent avec lui-même, et lui afficher un message d'avertissement si l'on détecte plusieurs prédicats identiques mais avec des torts différents. On pourrait dans se cas lui proposer d'écraser l'ancienne règle par la nouvelle.

— Ou alors, on peut considérer qu'après tout, il n'est pas incohérent d'avoir plusieurs réponses, dans la mesure où, pour un même ensemble de cases cochées entre A et B, la situation peut être différente, lorsqu'un croquis permet de départager la situation. On pourrait alors demander à l'expert, lors de la maintenance automatique, d'associer une description textuelle à la nouvelle règle. Ainsi, lors du listing des différents torts possibles pour un ensemble de cases cochées, on verrait la description et on pourrait faire la différence.

Malheureusement, cette dernière solution ne résout pas le problème de la rupture de l'automatisme du système.

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 langage `Prolog`

`Prolog` est un langage créé en 1972 par Alain Colmerauer et Philippe Roussel. Ce langage tire son nom de « **PRO**grammation **LOG**ique ». Il supporte les mécanismes d' **unification**, de **récursivité** et de **retour sur trace** (*backtracking*).

Le langage `Prolog` permet au programmeur de se concentrer sur l'écriture des prédicats et des relations logiques qui les relient, laissant au compilateur la tâche de convertir cette écriture logique en instructions machine.

Le moteur de `Prolog` se sert d'une **base de faits** et d'une **base de règles** pour tenter de prouver un **but**. Lorsqu'un but n'a pas pu être prouvé en marche avant, `Prolog` revient sur ses pas (*backtracking*) pour tenter de prouver le but d'une autre manière.

`Prolog` étant un langage adapté à la programmation par contraintes, il a été particulièrement adapté à la programmation de notre système expert, qui repose justement sur des relations entre prédicats.

Concevoir un système expert a pour but à court terme d'assister l'expert, mais à long terme de le remplacer. Ainsi, le système expert doit être un programme appliquant des règles de départ statiques, avec la possibilité d'apprendre automatiquement (avec l'aide de l'expert) des cas particuliers, constituant les règles dynamiques. Le programme doit donc être capable d'**apprendre**. Et c'est cet apprentissage qu'il est particulièrement facile d'implémenter avec `Prolog`, grâce au mécanisme de **prédicat dynamique**.