



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

<b>1</b>	<b>Application</b>	<b>1</b>
1.1	Noyau . . . . .	1
1.2	Tests unitaires . . . . .	2
<b>2</b>	<b>Interface utilisateur</b>	<b>3</b>
2.1	Ligne de commandes . . . . .	3
2.2	Maintenance semi-automatique . . . . .	3
2.3	Réflexion sur une maintenance complètement automatique . . . . .	3
<b>3</b>	<b>Choix du langage Prolog</b>	<b>4</b>

# 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 entrainé 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

### 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 **récurtivité** 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**.