Actes de langage et jeux de dialogue
Manuel de référence
Simulateur de dialogue

Christophe Delord

delord@mygale.org
Laboratoire G.R.A.A.L.
(Groupe Raisonnement, Action et Actes de Langage)
Responsable de stage : Fabrice Evrard
ENSEEIHT-IRIT
2, Rue Charles Camichel - BP7122 - 31071 TOULOUSE Cedex 7

15 septembre 1998

Table des matières

1 Description des différents modules du programme

1.1 Programme principal : main.pl

Le fichier main.pl contient l'interface utilisateur et charge tous les modules nécessaires à la simulation de dialogue. Le prédicat main/0 permet d'éxecuter le menu principal. Ce menu possède les entrées suivantes :

Remise à zéro Suppression du résultat des compilations précédantes. Après cela, le simulateur se retrouve dans l'état initial (le système ne connait plus aucun agent ni aucun jeu).

Fichier Choix du fichier principal de la description des agents et des jeux.

Editer Exécution de vi avec le fichier principal.

Compiler Compilation des agents et des jeux définis dans le fichier principal et les fichiers inclus dans le fichier principal.

Simuler Simulation de la précédente description compilée.

Options Sous-menu permettant de modifier le comportement du simulateur.

Quitter Retour à la ligne de commande de Prolog ou du shell.

Le menu des options est le suivant :

Afficher les nouveaux états Affichage des états modifiés à chaque tour, ou affichage complet de l'ensemble des états mentaux des agents.

Afficher chaque étape Affichage des états mentaux à chaque coup de dialogue, ou seulement à la fin de la simulation.

Trace Activation du menu de trace à chaque coup de dialogue.

Fenêtre Debug Affichage d'une fenêtre supplémentaire pour la mise au point du programme.

Continuer Retourner au menu principal.

Il peut être utile de faire quelques remarques avant de continuer :

- Le nom de fichier saisi est un atome Prolog. Donc si le nom du fichier contient des caractères spéciaux (point, espace, ...), il faut le saisir entre guillemets. Par exemple : 'premier test.txt'.
- L'édition et la compilation ne sont possibles que si le fichier principal a été défini.
- La simulation n'est possible qu'après la compilation.

1.2 Déclarations préliminaires : descls.pl

Le module decls déclare les prédicats dynamiques qui vont être utilisés pendant la compilation et la simulation, ainsi que les opérateurs utilisés pour la description des agents et des jeux.

1.3 Compilation des agents et des jeux : compil.pl

Le module compil transforme la description des jeux et des agents en un ensemble de prédicats qui seront utilisés pour la simulation du dialogue. Les prédicats relatifs à un agent sont stockés dans un module dynamique de même nom que l'agent.

1.4 Simulation du dialogue : dialogue.pl

Le module dialogue utilise le résultat de la compilation pour simuler le dialogue. C'est ce module qui va appeler les routines de fenétrage pour l'affichage des états mentaux et des coups, les routines de trace, ...et bien sûr les règles de perception, mise à jour des états mentaux, de génération et d'interdiction des agents.

1.5 Outils de trace du dialogue : pasapas.pl

Le module pasapas permet à chaque coup de dialogue de visualiser l'état des agents et de modifier l'état du système. A l'invite, le programme attend plusieurs commandes :

espace Continuer

h Aide

c Modification du dernier coup

t Trace du programme au niveau de Prolog

a|x Abandon

1.6 Fenétrage textuel : windows.pl

Le module windows permet de gérer plusieurs fenètres (affichage de texte, scrolling) et de lier le scrolling entre plusieurs fenètres, par exemple, le scrolling de la fenètre d'un agent provoque celui de l'autre agent et de la fenètre des coups pour avoir toujours le contenu de ces trois fenètres bien aligné.

1.7 Utilitaires divers : cd-tools.pl

Ce dernier module contient des prédicats inclassables ailleurs. Ils sont décrits dans le rapport de stage.

2 Langage de description

Le langage utilisé pour la description des agents et des jeux est décrit dans le rapport. Nous allons en faire un résumé.

Tout d'abord, le langage est constitué uniquement de termes Prolog afin d'en faciliter l'analyse syntaxique. Il ne faudra donc pas oublier de terminer chaque déclaration par un point. On peut utiliser les commentaires de Prolog (% ... ou /* ... */).

2.1 Logique modale

Les expressions de la logique modale seront aussi codées par des termes Prolog. Les modalités sont représentées par des opérateurs binaires infixes[1] (b, w, kif, ...). La liste complète est dans le fichier decls.pl et peut être complétée.

2.2 Exemple de description

Pour décrire le langage, le plus simple est de reprendre l'exemple du policier présenté dans le rapport.

2.2.1 Déclaration des jeux

Le langage étant basé sur les termes de Prolog, on notera l'utilisation de la virgule pour séparer les éléments d'une liste. Les listes en Prolog sont écrites entre crochets. Ici les accolades sont converties en crochets par le compilateur. On peut utiliser les accolades ou les crochets (ou les deux).

Le jeu de base ne possède pas de conditions d'entrée ou de sortie. Ce sont les conditions par défaut qui seront utilisées, de manière à rentrer dans le jeu et à ne plus en sortir.

La clause «import» permet d'inclure la description d'un jeu dans le jeu en cours de définition. L'argument de «import» doit être le nom complet du jeu importé, c'est-à-dire le terme avec tous ses arguments. Si l'on importe le jeu «question», il faut écrire «question(_)» pour traiter toutes les questions, ou «question(p)» pour ne prendre en compte que la question «p». Mais «import question» produirait une erreur.

```
game jeu_de_base =
{
    perception =
    {
        C => x b (C:t) | C\=($ _) % Mémorisation des coups autres que $
    },
```

[1] Les termes «x b p» et «b(x,p)» sont identiques, on peut donc utiliser les notations infixes ou préfixes indifférement. Cependant, l'opérateur «b» étant déclaré infixe, les affichages se feront dans cette notation. Pour changer ce comportement, il suffit de retirer les ces déclarations du fichier decls.pl.

```
    update =
    {
        x b (y ? Q:_) => x b y w y kif Q,
        x b (y ! P:_) => x b y b P,
        x b (y ! P:_) and x w x kif P => x b P,

        x b P => -(x b -P),

        x b (y ! P:_) => x b y b x a x kif P
    },

    generation =
    {
        true => $ x # 0.01 % fin du dialogue ...
    },

    interdiction =
    {
        x b (C:T) => -C | T<t # 100,
        x w _ or x b y w _ => - $ x % ... si plus d'intention
    }
}.

game question(Q) =
{
    import jeu_de_base,

    entrance = x w x kif Q,
    invariant = x w x kif Q,
    exit = x kif Q,

    generation =
    {
        x w x kif Q => x ? Q # 10
    },

    update =
    {
        x kif Q => -(x w x kif Q)
    }
}.

game reponse(Q) =
{
    import jeu_de_base,

    entrance = x b y w y kif Q and x b y a y kif Q,
    invariant = x b y w y kif Q,
    exit = x b y kif Q,

    generation =
    {
        x b Q => x ! Q # 10,
```

```
        -(x b Q) => x '!?' Q
    },

    update =
    {
        x b (P->Q) and -(x kif P) => x w x kif P,
        x b y kif Q => -(x b y w y kif Q),

        x b (x ! P:_) and x b y w y kif P => x b y b P

    }
}.

game conflit(Q) =
{
    import jeu_de_base,

    entrance = x b y b -Q and x b Q,
    invariant = x b y b -Q and x b Q,
    exit = x b y b Q or x b -Q,

    generation =
    {
        x b (P->Q) => x ! P # 100,
        x b (P->Q) => x ! (P->Q),
        x b Q => x ! Q
    },

    update =
    {
        x b y b Q => -(x b y b -Q),
        x b -Q => -(x b Q)
    }
}.

game mefiance(Q) =
{
    import jeu_de_base,

    entrance = x b y w Q and -(x b y a Q),
    invariant = x b y w Q and -(x b y a Q),
    exit = x b y a Q or -(x b y w Q),

    update =
    {
        x b (y ! P:_) and x b (P -> (y a Q)) => x b P,
        -(x b y a Q) => x w x kif y a Q
    },

    generation =
    {
        -(x b y a Q) => x ! -(y a Q)
    }
}.
```

2.2.2 Déclaration des agents

Comme pour les jeux, les agents peuvent être génériques. L'agent de base est par exemple paramétré par la profondeur maximale du moteur d'inférence.

```
agent agent_de_base(PROF) =
{

    start = jeu_de_base,
    games = { question(_), reponse(_) },

    depth = PROF,

    kb =
    {
        nonvar(Q) | x b (P->Q) and x b P then x b Q,
        nonvar(P) | x b (P->Q) and x b -Q then x b -P,

        x b (p(X)->(X a Q)),

        nonvar(P) | (x b P) or (x b -P) then x kif P
    }
}.

agent agent_evolue(PROF) =
{
    import agent_de_base(PROF),

    kb =
    {
        x b p(a1)
    }
}.
```

On notera la différence entre «import» et «accointance». «import» importe une description d'agent. «accointance» importe une description tranformée pour servir de représentation de l'autre.

```
agent a1 =
{
    import agent_de_base(1),

    accointance = agent_de_base(1),

    games = { conflit(_) },

    kb =
    {
        x w x kif q,
        x b p(x)
    }
}.
```

```
agent a2 =
{
    import agent_de_base(1),

    games = { mefiance(_) },

    kb=
    {
        x b q
    }
}.
```

2.2.3 Simulation du dialogue

Pour démarrer la simulation, il faut désigner deux agents. Ces deux agents vont être compilés puis utilisés pour simuler le dialogue. Si l'on utilise un agent générique, il faudra instancier ses arguments.

```
a1 and a2 are speaking.
```

2.2.4 Inclusion de fichiers

Il n'est pas nécessaire de décrire tous les jeux et tous les agents dans le même fichier. La commande «import fichier» permet d'inclure le fichier «fichier» dans le fichier courant à la manière de `#include` en C.