Traitement de requêtes de bon sens sur les actions pour l'interaction homme-machine

Nicolas Sabouret Jean-Paul Sansonnet
nico@limsi.fr jps@limsi.fr

LIMSI-CNRS
BP133, 91403 Orsay Cedex

Résumé :
Dans cet article, nous nous intéressons à l'interaction entre un agent assistant d'interface et un utilisateur ordinaire en situation de demande d'aide. Nous proposons une modélisation formelle des questions posées par les utilisateurs concernant le fonctionnement de l'agent. Nous n'abordons pas les problèmes liés au traitement du langage naturel pour la construction de ces requêtes formelles mais nous montrons que ce type d'interaction fait intervenir des connaissances *de bon sens*, non définies dans le composant. Nous proposons des outils et des algorithmes pour modéliser et traiter ces connaissances au niveau formel.

Mots-clés : Raisonnement sur les actions, questions sur le fonctionnement, interaction homme-machine, composants actifs dialogiques

Abstract:
In this paper, we consider the interaction between an agent and an ordinary user in situation of failure while performing a task. We propose a formal model of the questions the human user can ask about the agent's action and changes. We do not study the natural language processing issues which appear in the production of the requests. We rather show that this interaction involves *common-sense* notions, not defined within the agent's code. We propose tools and algorithms to model and process such notions at the formal level.

Keywords: Reasoning about actions, questions about actions, human-computer interaction, conversational agents

1 Exposé de la problématique

Dans cet article, nous abordons une problématique qui est en train d'émerger dans les domaines de l'interaction homme-machine (IHM) et de l'interaction dialogique entre agents autonomes. Avec le développement des outils informatiques pour le grand public, en particulier dans les services en ligne sur Internet, nous voyons apparaître la nécessité pour un composant logiciel de pouvoir interagir avec un *utilisateur ordinaire* pour l'assister dans la réalisation d'une tâche donnée [1]. Lorsque l'utilisateur se retrouve en situation d'échec, les études [11, 14] ont montré qu'il voulait pouvoir *poser des questions* au système à propos de *ce qu'il fait*. Dans le projet *InterViews* [21], nous essayons de modéliser et d'implémenter de tels composants actifs, qui seraient capables de se représenter leur fonctionnement, de raisonner dessus et d'interagir en langue naturelle avec l'utilisateur pour répondre *en cours d'exécution* à un large éventail de questions portant sur leurs actions et leur exécution.

Les travaux effectués en diagnostic [25, 13], en planification [6] et plus généralement dans le domaine du raisonnement sur le fonctionnement [12, 9, 16] sont à la base de cette étude, puisqu'ils constituent les outils indispensables à la construction d'une explication pour l'utilisateur. La construction de ces réponses s'appuie aussi sur l'utilisation de modèles de raisonnement sur les actions [23, 3, 2] et la causalité [15, 10].

Cependant, tous ces formalismes travaillent uniquement sur des requêtes bien formées syntaxiquement et *sémantiquement*, c'est-à-dire qui utilisent les éléments *internes* au composant (ou à son modèle formel). Or ce qui caractérise les utilisateurs ordinaires en IHM, c'est qu'ils ne disposent pas de connaissances spécifiques sur le fonctionnement interne du système, les données qu'il manipule ou ses capacités d'interaction. Par conséquent, pour faire référence aux composants techniques sur lesquels portent leurs questions, ils ne peuvent pas utiliser des termes spécifiques, définis en interne ou dans le modèle. Au contraire, ils utilisent le sens commun, c'est-à-dire des *notions de bon sens* partagées par tous les utilisateurs [8]. Par exemple, un utilisateur ordinaire ne va pas demander « *à quoi sert le switch* x_1 ? » mais « *à quoi sert le gros levier de gauche* ».

L'interprétation de ces notions de bon sens, lorsqu'elles portent sur le fonctionnement du composant, ne peut pas être intégralement effectuée par un module de traitement du langage naturel parce que celui-ci n'a pas accès à la *sémantique opérationnelle* du modèle considéré. Dans ce cadre, les requêtes *formelles* construites par ce module font nécessairement intervenir des *connaissances de bon sens*, c'est-à-dire :

– Non définies dans le modèle formel du composant ;

– Issues directement de la question de l'utilisateur ;
– Compréhensibles par un humain dans le contexte du composant.

Par conséquent, pour traiter les requêtes formelles et plus généralement pour interagir avec un utilisateur en situation de demande d'aide, le système doit pouvoir utiliser et manipuler ces connaissances de bon sens dans les requêtes et les relier avec les éléments internes du programme : objets, connaissances, actions possibles, *etc*.

Dans cet article, nous proposons un modèle formel de l'interaction entre un agent et un utilisateur en situation de demande d'aide. Nous définissons d'abord un modèle de requêtes qui permet de représenter une large classe de questions qu'un utilisateur humain peut poser concernant le fonctionnement d'un composant actif simple. Nous proposons ensuite un cadre formel pour la représentation et la manipulation des connaissances de bon sens dans ces requêtes.

2 Étude du besoin

2.1 Schéma de principe

L'analyse du problème de l'interaction avec un utilisateur en situation de demande d'aide nous a conduit à proposer le schéma de principe présenté figure 1. Il s'appuie tout d'abord sur l'utilisation d'un langage de description de composants spécifique [17, 18], fondé sur la réécriture d'arbre. Ce langage :

– est suffisamment expressif pour représenter une large classe de composants actifs,
– permet d'accéder *en cours d'exécution* à une description formelle des actions des composants,
– est utilisable pour produire des explications sur le fonctionnement en réponse à des questions de l'utilisateur.

Dans la suite de cet article, nous utiliserons une syntaxe LISP pour décrire les actions du langage, qui ne sera pas présenté plus en détail. Nous nous attacherons au contraire à la modélisation formelle de l'interaction, c'est-à-dire à :

– La présentation d'un langage de requêtes qui permet de relier le module de traitement du langage naturel (MLN) avec le module de raisonnement sur le fonctionnement (MRF) ;

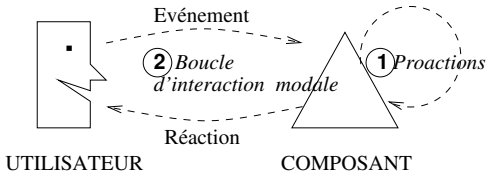

FIG. 2 – Comportement proactif et boucle d'interaction modale

– L'étude de la prise en compte des notions de bon sens dans ces requêtes et dans les mécanismes de construction de réponse au sein du MRF.

2.2 Modes d'interaction

Si nous considérons un composant muni de capacités d'interactions avec des utilisateurs humains ou d'autre composants, nous pouvons mettre en évidence deux modes d'exécution :

– Dans le mode *réactif*, les composants effectuent des opérations en réponse à des interactions de l'utilisateur (par exemple, cliquer sur le bouton « démarrer »).
– Dans le mode *proactif*, les composants fonctionnent en dehors de toute interaction. Par exemple, un processus de cuisson, une fois amorcé, ne nécessite pas d'interaction pour s'exécuter.

La figure 2 schématise ces deux modes de fonctionnement.

2.3 Perception de l'utilisateur

Lorsqu'un utilisateur humain interagit avec un composant logiciel sans connaissance préalable sur son fonctionnement, il tend à catégoriser celui-ci en utilisant des notions de bon sens [8]. Dans notre modèle, nous essayons de prendre en considération les différentes catégorisation possibles :

– Nous appelons *comportement* toutes les actions que le composant peut effectuer, aussi bien de manière réactive que de manière proactive. Par exemple, le comportement d'un ascenseur est de pouvoir se rendre à l'étage correspondant au bouton sur lequel l'utilisateur appuie, puis d'ouvrir automatiquement les portes lorsqu'il arrive à cet étage.
– Nous appelons *activité* les actions que le composant est en train d'effectuer à un instant

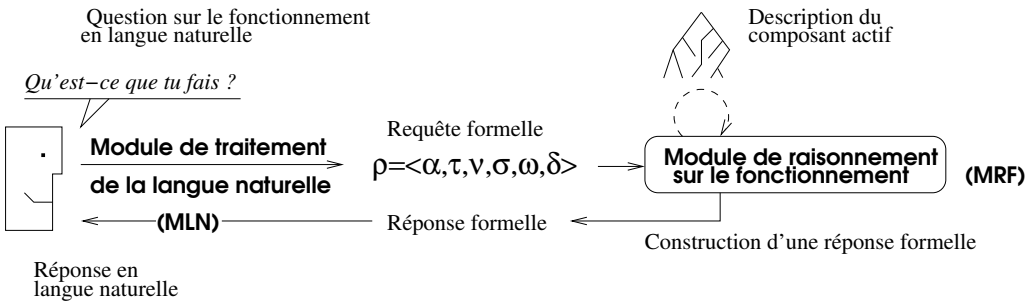

FIG. 1 – Traitement des questions sur le fonctionnement dans le projet *InterViews*.

donné de l'exécution, qu'il s'agisse de proactions ou de réactions à une interaction de l'utilisateur. Par exemple, le fait qu'un ascenseur se mette en route lorsque l'utilisateur appuie sur l'étage désiré fait partie de son activité à cet instant. De même, lorsqu'il est en mouvement, c'est cette action de déplacement dans l'espace qui fait partie de son activité.
– Nous appelons *état* d'un composant la description de la valeur courante de chacune de ses variables. Par exemple, l'état d'un ascenseur peut être d'être arrêté au 12ème étage et d'avoir les portes fermées.

2.4 Un corpus de questions sur le fonctionnement

D'autre part, en utilisant une technique du type « *magicien d'Oz* »[1], nous avons constitué un corpus de questions sur le fonctionnement d'un composant actif simple. En voici quelques exemples, issus de l'interaction avec deux composants : un compteur muni d'une vitesse (*Coco*) et un monde de cubes (*Jojo*)[2] :

– *Quelle est la valeur de ta vitesse ?*
– *Qu'est-ce que tu fais ?*
– *Pourquoi comptes-tu ?*
– *Qu'est-ce que tu peux prendre ?*
– *Quand valais-tu 10 ?*
– *Étais-tu arrêté il y a 10 minutes ?*
– *Comment faire pour mettre le triangle bleu au dessus du grand carré rouge ?*
– *etc.*

[1]L'utilisateur interagit avec ce qu'il croit être un système informatique alors qu'en réalité, ce système est simulé partiellement ou totalement par un humain (le « magicien »). L'objectif est d'observer le comportement des utilisateurs face au système avant d'entamer véritablement son implémentation .

[2]Il est possible d'interagir avec ces deux composants sur notre page web : `http://www.limsi.fr/Individu/nico/exemples/`

Nous ne prétendons pas que les questions que nous avons dégagées par cette méthode sont exhaustives. Mais elles constituent un bon échantillon représentatif des questions « naturelles » qu'un utilisateur peut poser concernant l'activité et le comportement d'un composant actif simple.

Nous avons ensuite classées ces différentes questions en fonction :

– De la fonction du message (performatif), en nous inspirant de la théorie des actes de langage [22] ;
– Du type de connaissances qu'elles font intervenir sur le plan procédural (état, activité, comportement) ;
– Des mécanismes de construction de réponse qu'il est nécessaire de mettre en œuvre pour traiter ces questions.

Cette classification des questions nous a conduit à élaborer le modèle de requêtes proposé dans la section suivante. Ce modèle permet :

– au MLN de traduire les questions sous la forme d'une requête formelle,
– au MRF de construire une réponse, en s'appuyant sur la structure des requêtes.

Nous ne pouvons pas présenter ici le modèle de requêtes dans sa totalité. Nous ne faisons qu'en rappeler le principe général, décrit en détail dans [19].

3 Un modèle de requêtes

3.1 Structure des requêtes

Dans notre modèle, une requête sur le fonctionnement est définie par un sextuplet :

$$\rho =< \alpha, \tau, \sigma, \omega, \nu, \delta >$$

Les éléments de la requête correspondent à six critères *caractéristiques* qui permettent de prendre en compte les différentes informations de la question de l'utilisateur :

- $\alpha \in \{Ask, Assert, What, Why, How, When, Error, Unknown\}$ est *l'acte de langage* de la requête (par analogie avec les travaux de Searle [22]).
- $\tau \in \{is, do, can, order\}$ est le *type procédural* de la requête. Il permet de déterminer le type d'élément procédural (action, processus, *etc*.) sur lequel porte la requête.
  A chaque couple (α, τ) correspond un type de question particulière et un traitement spécifique. Les valeurs pour l'acte et le type des requêtes proviennent de la classification des questions de notre corpus.
- $\sigma \in \mathcal{P}(\Upsilon)$ est une référence vers le *sujet* de la question, *i.e.* le sous-terme de la description du composant sur lequel porte la question. Le module de traitement de la référence garantit l'exactitude de σ.
- $\omega \in \mathcal{P}(\Upsilon)$ est *l'objet* de la question. C'est un ensemble de termes du langage de description de composants correspondant aux paramètres de la questions. Ils ne sont pas nécessairement des éléments (*i.e.* des sous-termes) de la description du composant.
- $\nu \in \{\top, \bot\}$ est un marqueur booléen.
- $\delta \in \Upsilon$ est la *date* sur laquelle porte la question. Dans les exemples de cet article, nous aurons généralement $\delta = \surd$, correspodant au présent.

Les réponses à ces requêtes sont elles-mêmes des requêtes dont l'acte de langage est $Assert$, $Error$ ou $Unknown$.

3.2 Construction des requêtes

Dans le projet *InterViews* [21], le Module de Langue Naturelle (MLN) est réalisé par S. Gérard [7]. Il apparie les concepts d'actions avec la question de l'utilisateur à l'aide d'une classification sémantique comme celle proposée par *WordNet* [5]. Il construit ainsi une requête dont les arguments sont les éléments de la description procédurale du composant. Le mécanisme mis en œuvre, reposant sur un ensemble de règles bien définies, est trop complexe pour être présenté ici.

Exemple. Considérons la question en langue naturelle : « *Est-ce que tu comptes ?* » pour le composant *Coco*. Le MLN produira alors la requête :

$$\left\langle \begin{array}{l} \alpha = Ask, \tau = do, \sigma = \{view\}, \\ \omega = \{count\}, \nu = \top, \delta = \surd \end{array} \right\rangle$$

Expressivité. Notre langage de requêtes permet de représenter la totalité des questions de notre corpus. Par exemple :

- *Quelle est la valeur de ta vitesse ?* $\langle What, is, \{speed\}, \emptyset, \top, \surd \rangle$
- *Qu'est-ce que tu peux prendre ?* $\langle What, can, \{view\}, \{take\}, \top, \surd \rangle$
- *Pourquoi comptes-tu ?* $\langle Why, do, \{view\}, \{count\}, \top, \surd \rangle$
- etc.

3.3 Résolution des requêtes

Le mécanisme de construction de la réponse dépend à la fois du type et de l'acte de la requête. Nous présentons ici simplement le principe général de construction d'une réponse, schématisé sur la figure 3. Nous illustrerons ce mécanisme sur un exemple dans la section 5.

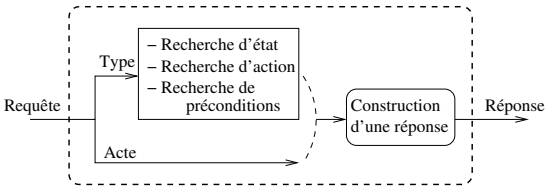

FIG. 3 – Traitement des requêtes dans le MRF

Le type τ de la requête permet de préciser la nature (processus, événement, état, *etc*.) des objets recherchés dans le traitement de la requête :

- $\tau = is$ pour les questions sur l'*état* du composant.
- $\tau = do$ pour les questions sur l'*activité* du composant. Le MRF recherche prioritairement des *proactions*. En cas d'échec, la recherche est étendue aux réactions.
- $\tau = order$ pour les questions sur l'interaction avec le composant, correspondant à la recherche des réactions.
- $\tau = can$ pour les questions sur le *comportement* possible du composant. Le MRF doit alors rechercher et analyser les préconditions de toutes les actions (proactions ou réactions).

L'acte α permet ensuite de déterminer le travail à effectuer sur les éléments concernés :

- $\alpha = What$ pour rechercher simplement tous les éléments correspondant au type τ dans la description du composant,
- $\alpha = Ask$ pour rechercher dans la description du composant les éléments correspondant à l'objet ω de la requête,
- $\alpha = How$ pour effectuer une planification suivant des techniques classiques [6, 4].
- $\alpha = Why$ pour générer une explication qui s'appuie à la fois sur des techniques de diagnostique [25, 13] et sur des techniques de raisonnement causal [15, 10].

La forme de la réponse dépend de τ, α et du résultat $\mathcal{R}$ de la recherche. Par exemple, pour $\alpha = What$ avec $\omega = \emptyset$ et $\mathcal{R}_\nu \neq \emptyset$, la réponse est de la forme :[3]

$$< Assert, \tau, \sigma, \mathcal{R}_\nu, \nu, \delta >$$

Les différents cas possibles et les algorithmes de construction de réponse associés sont définis de manière plus détaillée dans [19].

4 Manipulation de *connaissances de bon sens*

4.1 Analyse du problème

Rappel de la problématique. Lorsque les utilisateurs ordinaires interagissent avec un composant pour lui poser des questions à propos de son état, de son activité, de ses capacités d'interaction ou de son comportement, ils ne peuvent pas faire appel aux connaissances spécifiques sur le composant, définies en interne dans le modèle, auxquelles ils n'ont pas accès. Au contraire, ils vont utiliser des notions de bon sens dans leurs questions.

Dans le meilleur cas, le traitement de la référence effectué par le MLN va permettre de relier ces notions à des éléments internes du composant. Par exemple : « le cube rouge » ou « compter » seront appariés à des éléments du modèle, en supposant qu'une variable de type figure dont les attributs correspondent à « cube » et à « rouge » ainsi qu'une action nommée « compter » ont été définis dans le composant. Les requêtes ainsi construites sont « bien formées » en ceci qu'elles utilisent uniquement les connaissances internes du modèle.

[3] C'est le seul cas où la forme de la réponse ne dépend pas du type τ. C'est pourquoi nous la donnons ici.

Au contraire, lorsque l'utilisateur fait appel à des connaissances de bon sens, c'est-à-dire qui ne sont pas définies explicitement dans le composant, les requêtes construites contiennent des éléments non définis en termes du modèle. Par exemple : « *augmenter* une variable » ou « compter *à l'envers* ».

Notion de *Grinder*. Ce problème de la prise en compte des connaissances de bon sens a été largement étudié en traitement automatique du langage naturel (TALN). L'approche dominante, proposée par Stalkner [24], consiste à proposer un *common ground*, c'est-à-dire un ensemble de connaissances communes à tous les composants, en plus des connaissances spécifiques, et un *module d'interprétation sémantique* qui interprète les notions de bon sens de ce *ground* dans le contexte du composant, en fonction de la question posée en langage naturel.

Les modèles formels de l'interaction proposés dans la littérature ne se sont pas intéressés à la prise en compte de ces connaissances de bon sens dans les requêtes *formelles*. Il s'agit pourtant d'un problème spécifique qui *ne peut pas* être traité en utilisant les techniques classiques proposées en TALN. En effet, l'interprétation de ces connaissances nécessite d'accéder à la sémantique opérationnelle du composant et de raisonner sur les actions.

Pour répondre à ce besoin, nous proposons de définir des *grinders*[4], c'est-à-dire des modules d'interprétation sémantique *en contexte* fondés sur la sémantique opérationnelle du langage et dont l'objectif est de :

- Modéliser les notions de bon sens qui apparaissent dans les questions des utilisateurs en situation de demande d'aide ;
- Interpréter ces notions en utilisant la *sémantique opérationnelle* du langage de description du composant.

Trois *Grinders*. L'étude de notre corpus de questions sur le fonctionnement fait apparaître l'existence de trois types de connaissances de bon sens, correspondant à trois *grinders* différents :

- Les relations entre des variables, comme « x est positif », qui seront modélisées et manipulées à l'aide du *grinder des relations statiques* (GRS) ;

[4] Par analogie avec le *common ground* de Stalkner.

– Les actions, comme « augmenter », qui seront modélisées et manipulées à l'aide du *grinder procédural* (GP) ;
– Les relations entre les actions, comme « $start$ est l'inverse de $stop$ » ou « ne rien faire », qui seront modélisées et manipulées à l'aide du *grinder des relations procédurales* (GRP).

Dans cette section, nous présentons une première proposition pour ces trois grinders et un mécanisme à base de *pattern-matching* qui permet de manipuler ces connaissances de bon sens.

Afin de décrire des notions de bon sens génériques, les éléments des grinders seront des patterns dont certaines parties ne sont pas instanciées. Elles seront notées $\#i$, $i \in \mathbb{N}$ et sont appariées lors de l'interprétation de la notion de bon sens par l'analyse *en cours d'exécution* du code du composant. Pour tout ensemble de termes E et pour tout pattern p, nous notons $\xi(p, E)$ l'ensemble des termes $e \in E$ s'appariant avec p, c'est-à-dire tel qu'il existe un ensemble l de substitutions de la forme $(i, \{t_k\}_i)$ tel que le terme p instancié par l soit subsumé par e (au sens de la comparaison de termes en logique de réécriture ou ou logique des prédicats).

L'instance d'un pattern p suivant une liste de substitution l est le terme obtenu en remplaçant dans p tous les $\#i$ par l'ensemble de termes $\{t_k\}_i$ associé à i dans l, ou en le supprimant s'il n'est pas instancié dans l. Pour tout terme t et pour tout pattern p, nous notons $pm(t, p, l) = (b, l')$ le résultat de l'appariement de p avec t en utilisant l comme liste de substitution initiale. Le résultat est un booléen $b = \top$ lorsque l'appariement réussit et une liste l' donnant l'ensemble des substituations.

4.2 Structure du GRS

Nous notons $\mathcal{G}_{rs}$ le grinder des relations statiques. Ses éléments sont des couples de patterns (s, t). Nous dirons que s est le pattern de ground et t le pattern associé, ne contenant que des éléments bien définis en terme de la sémantique opérationnelle du langage. Par exemple, nous aurons dans $\mathcal{G}_{rs}$:

$$(\ (positive\ \#1)\ ,\ (>\ \#1\,0)\ )$$

Dans $\mathcal{G}_{rs}$, un pattern de ground ne peut être associé qu'à un seul pattern du langage, et réciproquement[5] :

$$\{(s,t),(s',t')\} \subset \mathcal{G}_{rs} \Rightarrow (s = s') \leftrightarrow (t = t')$$

Algorithme. Nous notons $\Xi_{rs} : \Upsilon \times \mathbf{B} \longrightarrow \Upsilon$ l'algorithme de conversion d'un terme de ground en terme du langage (et réciproquement) pour le GRS. Le deuxième argument est un booléen permettant de préciser le sens de la conversion.

Algorithm 1 *fonction* $\Xi_{rs}(t \in \Upsilon, b \in \mathbf{B})$

```
si b, G = {s tq ∃t' tq (s,t')∈G_rs}
sinon, G = {t' tq ∃s tq (s,t')∈G_rs}
res = ξ(t,G)
si res≠∅
  soit t' = min(res)
  soit (l,b')=pm(t,t',∅)
  retourner t' instancié suivant l
sinon
  soit {st_1,...,st_n}=cdr(t)
  soit c=car(t)
  retourner
      (c Ξ_rs(st_1,b) ... Ξ_rs(st_n,b) )
```

L'heuristique qui consiste à ne considérer que le plus petit terme possible dans l'algorithme ($min(res)$) correspond à l'hypothèse que les relations construites par le MLN à partir des questions de l'utilisateur ne sont pas ambiguës. $\Xi_{rs}(t, b)$ est alors calculable en $\mathcal{O}(|t|^2.|\mathcal{G}_{rs}|^{|t|})$ dans le pire des cas. Ce résultat provient de la bijection dans $\mathcal{G}_{rs}$.

Remarques. Dans l'algorithme, $car(t)$ et $cdr(t)$ désignent respectivement, comme en LISP, la tête et le reste des éléments du terme t.

Exemple. Si $t = (positive\, value)$ et $b = \top$, alors $\Xi_{rs}(t, \top) = (>\ value\, 0)$. La valeur de vérité de la relation instanciée est obtenue en utilisant l'interprétation canonique définie dans la sémantique opérationnelle du langage.

4.3 Structure du grinder procédural

Nous notons $\mathcal{G}_p$ le grinder procédural. Ses éléments sont des triplets de patterns (s, t, r) tels que :

[5] Il y a une bijection entre les éléments en partie gauche et en partie droite de $\mathcal{G}_{rs}$. Elle correspond au fait que, puisque $\mathcal{G}_{rs}$ doit permettre de transformer des relations de bon sens en termes du langage, il ne peut pas y avoir d'ambiguïté : une relation de bon sens ne peut correspondre qu'à un seul terme du langage. De plus, cette bijection permet de réduire considérablement la complexité de nos algorithmes de conversion.

– s est le pattern d'action de ground, représentant une action de haut niveau sémantique,
– t est le pattern associé dans le langage, représentant une action élémentaire bien définie,
– r est un pattern de relation booléenne qui doit être vérifiée pour que la correspondance entre s et t soit valide. Cette relation peut utiliser des concepts de ground issus de $\mathcal{G}_{rs}$.
Nous dirons aussi que r définit une contrainte sur le pattern t.

Par exemple, nous aurons dans $\mathcal{G}_p$:

$$\left(\begin{array}{l} (increase\,\#1\,\#2), \\ (setq\,\#1\,(+\,\#1\,\#2)), \\ (positive\,\#2) \end{array} \right)$$

Un pattern d'action de ground ne peut être associé qu'à un seul pattern du langage et un pattern du langage contraint par une relation donnée ne peut être associé qu'à un seul pattern d'action de ground[6] :

$$\begin{array}{ll} & \{(s,t,r),(s',t',r')\} \subset \mathcal{G}_{rs} \\ \Rightarrow & (s = s') \leftrightarrow ((t,r) = (t',r')) \end{array}$$

Algorithme. L'interprétation des actions de ground ne peut pas se faire à l'aide d'une simple conversion, comme c'était le cas pour le grinder des relations statiques. En effet, les éléments $\#i$ apparaissant dans les patterns peuvent ne pas être instanciés, parce que la question de l'utilisateur ne fait pas *nécessairement* appel à ces éléments. Par exemple :

– *Qu'est-ce que tu augmentes ?* ($\#1$ et $\#2$ ne sont pas instanciés) ;
– *Est-ce que ta vitesse augmente ?* ($\#2$ n'est pas instancié).

Dans ce cadre, il n'est ni possible de rechercher les actions correspondantes dans la description du composant, ni d'évaluer la relation de contrainte qui permet d'effectuer la correspondance entre une action de ground et une action bien définie en terme de sémantique opérationnelle du langage.

Le mécanisme d'interprétation sémantique des actions de ground se décompose alors en deux parties distinctes utilisées à deux moments différents de l'algorithme de construction de réponse dans le MRF :

1. L'action de ground est tout d'abord convertie en une action bien définie en ignorant la valeur de vérité de la relation r, à l'aide d'un algorithme Ξ_p similaire à Ξ_{rs} ;
2. L'action en question est alors recherchée dans le composant, en utilisant la *subsomption de termes* ;
3. Les actions trouvées sont reconverties en actions de ground, et vérifiant la relation de contrainte[7]. Toutes les actions qui ne correspondent pas à l'action de ground initialement cherchée, c'est-à-dire qui ne sont pas subsumées par cette action, sont supprimées

Le résultat est un ensemble d'actions qui va pouvoir être utilisé par le MRF pour construire une réponse, en fonction du type τ et de l'acte α de la requête. La figure 4 illustre ce mécanisme. Soulignons que le traitement des requêtes faisant intervenir des actions est beaucoup plus compliqué que celui des requêtes « statiques ». Le *grinder* ne peut plus être perçu comme une simple couche de traduction : il doit être intégré dans les algorithmes de construction de réponse, au niveau de la recherche des actions pour la construction de la réponse. Soulignons aussi que $\mathcal{G}_p$ utilise $\mathcal{G}_{rs}$ pour vérifier la relation r.

4.4 Structure du GRP

Nous notons $\mathcal{G}_{rp}$ le grinder des relations procédurales. Il permet d'exprimer des relations dont les éléments sont des actions élémentaires. Par exemple : « f est *l'inverse* de g » (relation binaire), « je compte *à l'envers* » (relation unaire), « je ne fais rien » (relation d'arité 0).

Sa structure et son utilisation sont différentes de $\mathcal{G}_{rs}$ et $\mathcal{G}_p$ car il donne directement l'ensemble des tuples d'actions qui vérifient chaque relation.[8]

Structure. Les éléments de $\mathcal{G}_{rp}$ sont des triplets (n, c_s, R) tels que :

– n est l'arité de la relation procédurale,
– c_s le concept de ground de la relation,
– R est l'ensemble caractéristique de la relation.

[6] Il y donc une bijection entre l'ensemble des patterns d'action de ground et celui des couples (pattern d'action, contrainte) : $s_j \longleftrightarrow (t_j, r_j)$.

[7] Ce qui est maintenant possible puisque le système travaille avec des actions qui ont été extraites du composant, donc qui sont entièrement instanciées.

[8] Contrairement à $\mathcal{G}_{rs}$ qui ne sert qu'à convertir la relation en une expression du langage pour pouvoir ensuite l'évaluer.

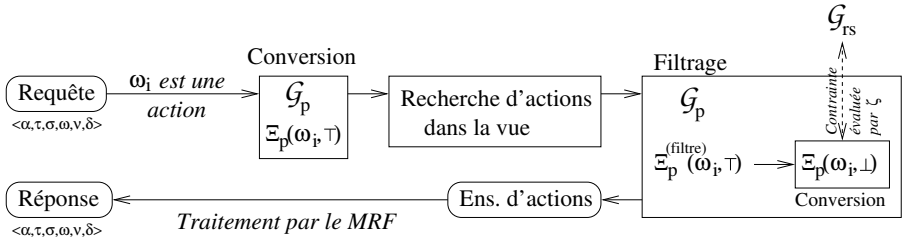

FIG. 4 – Traitement des requêtes procédurales

Les éléments de R sont des couples (T, r) tels que :

- T est un ensemble $\{t_1, ..., t_n\}$ de patterns d'actions (de ground ou du langage),
- r est un pattern de relation qui doit être valide pour que la relation procédurale soit vérifiée.

Dans R, il existe une bijection entre l'ensemble des T_i et l'ensemble des r_i. Dans $\mathcal{G}_{rp}$, un même couple (n, c_s) ne peut apparaître qu'une seule fois.

Exemple. Par exemple, nous aurons dans $\mathcal{G}_{rp}$:

$$\left(\begin{array}{c} 1 \\ backward \\ \left\{ \left(\begin{array}{c} \{(increase\ \#1\ \#2)\} \\ (negative\ \#2) \end{array} \right), ... \right\} \end{array} \right)$$

qui définit la relation unaire « à l'envers ». Dans cet exemple, nous définissons qu'un composant « incrémente à l'envers » une variable ($\#1$) si et seulement si l'incrément ($\#2$) est négatif.

Utilisation. Soit $\{t_1, ..., t_n\}$ un ensemble de termes représentant des actions élémentaires. Soit c_s une relation procédurale d'arité n (représentée ici pas son concept).

Les termes $\{t_1, ..., t_n\}$ vérifient la relation c_s ssi il existe un ensemble R tel que $(n, c_s, R) \in \mathcal{G}_{rp}$ et un ensemble de patterns $(\{t'_1, ..., t'_n\}, r) \in R$ tels que :

- il existe $l \in \mathbf{N} \times \Upsilon$ tq $\forall i \in \mathbf{N}, pm(t_i, t'_i, \emptyset) = (l, \top)$,
- l'instance de r suivant l est valide, au sens de l'interprétation canonique.

Le *grinder* des relations procédural est utilisé directement au sein des algorithmes de construction de réponse du MRF. En effet, il est utilisé pour évaluer une relation procédurale dans les requêtes modélisant des questions comme :

- Est-ce que $start$ et l'inverse de $stop$?
- Pourquoi est-ce que tu comptes à l'envers ?
- *etc*.

C'est l'évaluation de la relation ainsi que les relations de contrainte qui sont associé à cette relation qui sont utilisés pour construire la réponse. Pour interpréter les requêtes, le MRF extrait les paramètres de la relation et les convertit en actions bien définies à l'aide du *grinder* procédural. Il obtient alors un ensemble d'ensemble d'actions élémentaires (chaque ensemble d'action élémentaires correspondant à l'interprétation d'une action trouvée dans la description du composant). La valeur de vérité de la relation est alors déterminée en calculant, pour chaque combinaison possible d'actions élémentaires, si elles vérifient la relation considérée, suivant l'algorithme donné ci-avant. La figure 5 illustre ce mécanisme.

5 Un exemple complet

Considérons la question « *Est-ce que tu incrémentes ta valeur ?* » pour le compteur *Coco*. Le MLN utilise la liste des patterns de ground, la liste des noms des actions et des variables de *Coco* et la classification sémantique *WordNet* [5] pour construire la requête suivante :

α=Ask τ=do σ={view} ω={(increase value)} ν=true δ=$\surd$

Le MRF vérifie que les objets $o \in \omega$ de la requête ne sont pas des relations procédurales puis il construit l'ensemble $\omega' = \{\Xi_p(t, \top), t \in \omega\}$ pour n'avoir plus que des actions bien définies. Dans notre exemple, $(increase\, value)$ sera apparié à $(increase\, \#1\, \#2)$ avec $l = \{(1, \{value\})\}$. Ainsi, nous obtenons :

$$\omega' = \{(setq\, value\, (+\, value))\}$$

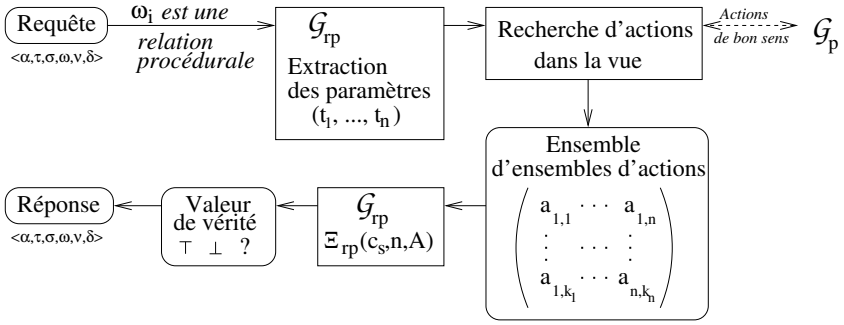

FIG. 5 – Traitement des relations procédurales

Le MRF recherche alors l'ensemble des actions correspondantes dans la description du composant, en filtrant toutes celles qui ne correspondent pas au type $\tau = do$ et au critère ν de la requête (c'est-à-dire en ne conservant que les processus actifs). En supposant que l'action « compter », correspondant à l'incrémentation de la valeur du compteur, soit active, nous obtenons alors l'ensemble d'actions :

$$\mathcal{A} = \{(setq\, value\, (+\, value\, speed))\}$$

Les éléments de $\mathcal{A}$ sont alors convertis en actions de ground en vérifiant la relation de contrainte associée. Dans notre exemple, en supposant que la vitesse soit positive, la relations de contrainte $(positive\, \#2)$ avec $\#2 = speed$ sera évaluée à vraie par $\mathcal{G}_{rs}$ et nous obtiendrons un ensemble d'actions de ground $\mathcal{A}' = \{(increase\, value\, speed)\}$ non vide.

Le MRF construit alors la réponse suivante (en raison de l'acte $\alpha = Ask$ de la requête initiale) :

```
α=Assert τ=do σ={view} ω={(increase
      value speed)} ν=true δ=√
```

Cette requête est envoyée au MLN qui a la charge de produire la réponse : « *Oui*[9] *: j'incrémente ma valeur par ma vitesse.* ».

6 Conclusion

Dans cet article, nous avons proposé d'une part un modèle formel pour l'interaction entre un agent et un humain en situation de demande d'aide et d'autre part un cadre formel pour représenter et manipuler les *notions de bon sens* sur le fonctionnement, non définies explicitement dans le composant, qui apparaissent dans cette interaction.

Le mécanisme de *grinder* que nous avons proposé constitue une première approche du problème qui présente trois avantages majeurs :

- Il est relativement simple ;
- Il est entièrement intégré dans les algorithmes de construction de réponse que nous développons au sein du projet *InterViews* et qui ont été proposés dans [19] ;
- Il permet de modéliser et de traiter toutes les notions de bon sens qui apparaissent dans le corpus de questions que nous avons constitué.

Cependant, cet outil reste très limité du point de vue de la représentation générique des actions de ground qui peuvent apparaître dans l'interaction homme-machine. C'est la raison pour laquelle nous travaillons actuellement sur deux extensions majeures :

- L'intégration du grinder avec des outils de TALN ;
- L'utilisation d'un grinder fondé sur l'analyse dynamique de l'exécution du composant [20].

Enfin, nous envisageons à terme d'étendre le modèle d'interaction vers un modèle plus dynamique, dans lequel l'utilisateur et l'agent pourraient dialoguer pour trouver un accord sur le sens d'une notion de bon sens (et donc son lien avec la sémantique opérationnelle du système). Cela suppose de définir un *grinder* évolutif, dans lequel des termes peuvent être retirés ou ajoutés au cours de l'exécution, ainsi qu'un langage de requêtes qui permette de modéliser ces sous-dialogues.

[9] L'acceptation « Oui » est obtenue par similitude entre la requête envoyée au MRF et la réponse.

Références

[1] J.F. ALLEN, D.K. BYRON, M. DZIKOVSKA, G. FERGUSSON, L. GALESCU, and A. STENT. Towards Conversational Human-Computer Interaction. *AI Magazine*, 2001.

[2] M. CASTILHO, A. HERZIG, and C. SCHWIND. Raisonnement sur les actions : les approches basées sur la causalité et la dépendance. In *Le temps, l'espace et l'évolutif, Ecole thématique du GDR I3*, volume 2, pages 193–207. 2000.

[3] M.A. CASTILHO, L.F. DEL CERRO, O. GASQUET, and A. HERZIG. Reasoning about actions in modal logic. In *Working Notes of the Symposium on Reasoning about Actions : Foundations and Applications - ESSLLI'98*, 1998.

[4] K.W. CURIE and A. TATE. O-Plan : the Open Planning Architecture. *Artificial Intelligence*, 52(1) :49–86, 1991.

[5] C. FELLBAUM, editor. *WordNet, An Electronic Lexical Database* . MIT-Press, 1998.

[6] R.E. FIKES, P.E. HART, and N.J. NILSSON. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 3 :251–288, 1972.

[7] S. GERARD and J.P. SANSONNET. A spatio-Temporal Model for the representation of Situations Described in Narrative Texts. In D.N. Christodoulakis, editor, *Proc. NLP2000*, volume 1835 of *LNAI*, pages 176–184. Springer-Verlag, 2000.

[8] J.J. GIBSON. *The Ecological Approach to Visual Perception*. Houghton Mifflin, Boston, 1979.

[9] H.J. LEVESQUE, R. REITER, Y. LESPERANCE, F. LIN, and R.B. SCHERL. Golog : a logic programming language for dynamic domains. *Journal of Logic Programming, special issues on actions*, 31 :59–84, 1997.

[10] V. LIFSCHITZ. Situation calculus and causal logic. In *Proc. KR'98*, pages 536–546, 1998.

[11] P. MAES. Agents that reduce workload and information overload. *Communications of the ACM*, 37(7), 1994.

[12] J. McCARTHY and P.J. HAYES. Some philosophical problems from the standpoint of Artificial Intelligence. *Machine Intelligence*, 4 :463–502, 1969.

[13] S. McILRAITH. Explanatory Diagnosis : Conjecturing Actions to Explain Observations. In *International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 167–177, 1998.

[14] D.A. NORMAN. *The invisible computer : why good products can fail, the personnal computer is so complex, and information appliances are the solution*. MIT Press, Cambridge, 1998.

[15] J. PEARL. Reasoning With Cause and Effect. In *Proc. IJCAI'99*, pages 1437–1449, 1999.

[16] R. REITER. *Knowledge in Action : Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[17] N. SABOURET. VDL Procédural : syntaxe et sémantique opérationnelle. Technical Report 2000-09, LIMSI-CNRS, 2000.

[18] N. SABOURET. Programmer des composants actifs dans le web sémantique. In *Journées de l'Action Spécifique STIC « Web Sémantique »*, Octobre 2002.

[19] N. SABOURET and J.P. SANSONNET. Un modèle de requêtes sur le fonctionnement de composants actifs. In *Proc. Modèles Formels de l'Interaction (MFI) 2001*, volume 3, pages 419–436, 2001.

[20] N. SABOURET and J.P. SANSONNET. Learning Collective Behavior from Local Interactions. In Barbara Dunin-Keplicz and Edward Nawarecki, editors, *From Theory to Practice in Multi-Agent Systems, Proc. CEEMAS 2001*, volume 2296 of *LNAI-LNCS*, pages 273–282. Springer-Verlag, 2002.

[21] J.P. SANSONNET. Description Scientifique du Projet InterViews — The InterViews Project. Technical Report 99-01, LIMSI-CNRS, 1999.

[22] J.R. SEARLE. *Speech Acts*. Cambridge University Press, 1969.

[23] M. SHANAHAN. Explanation in the Situation Calculus. In *Proc. IJCAI'93*, pages 160–165, 1993.

[24] R.C. STALNAKER. Assertion. In P. Cole, editor, *Syntax and Semantics : Pragmatics*, volume 9, pages 315–332. New York : Academic, 1978.

[25] M. THIELSCHER. A Theory of Dynamic Diagnosis. *Electronic Transactions on Artificial Intelligence (ETAI)*, 1997.