Etude d'un modèle de requêtes sur le fonctionnement de composants actifs

Nicolas Sabouret
LIMSI-CNRS
Université d'Orsay
BP133 - 91403 Orsay Cedex

Jean-Paul Sansonnet
LIMSI-CNRS
Université d'Orsay
BP133 - 91403 Orsay Cedex

Résumé *Dans cet article, nous nous intéressons aux problèmes soulevés par le traitement de requêtes formelles portant sur le fonctionnement d'un composant actif. Après une brève étude de la problématique et de ses enjeux, nous présentons le langage de description des composants qui définit le cadre de ce travail. Nous présentons ensuite un modèle de requêtes sur le fonctionnement permettant de modéliser les différents types de questions qu'un utilisateur humain peut poser concernant les actions et le fonctionnement d'un composant actif. Enfin, nous étudions six problèmes importants qui apparaissent dans cette étude et nous proposons quelques pistes pour les résoudre.*
Mots-clefs : *Modèle de requêtes sur le fonctionnement, Raisonnement sur les actions*

1 Etude de la problématique

Avec le développement des outils informatiques pour le grand public, nous voyons apparaître la nécessité pour un composant logiciel de pouvoir expliquer à l'utilisateur ce qu'il fait. Dans le projet *InterViews* [20], nous essayons de modéliser et d'implémenter de tels composants, qui seraient capables de se représenter leur fonctionnement, de raisonner dessus et d'interagir en langue naturelle avec l'utilisateur pour répondre à un large éventail de questions portant sur leurs actions et leur exécution.

L'un de nos objectifs est de munir ces composants de modèles de raisonnement pour servir de support à un mécanisme explicatif. Les travaux effectués en diagnostic [22, 14], en planification [6] et plus généralement dans le domaine du raisonnement sur le fonctionnement [13, 16, 11] sont à la base de cette approche, puisqu'ils constituent les outils indispensables à la construction d'une explication pour l'utilisateur.

1.1 Cadre de l'étude

Nous nous intéressons ici au traitement des questions qu'un utilisateur peut poser concernant le fonctionnement d'un composant actif. Le Module de Langue Naturelle (MLN), réalisé au sein du projet par S. Gérard [9], 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.

L'idée que nous soutenons dans cette approche est que séparer la compréhension de la question de l'utilisateur de la compréhension du fonctionnement du composant permet de travailler plus efficacement sur chaque partie, en particulier de répondre à une plus large classe de questions complexes sur le fonctionnement. La construction de réponses dans un environnement intégré comme le projet *InterViews* résulte alors d'une interaction entre le MLN et le Module de Raisonnement sur le Fonctionnement (MRF), chacun résolvant des problèmes spécifiques.

1.2 Mémorisation du passé dans des chroniques

Afin de répondre à des questions portant sur le passé du composant, nous avons proposé un modèle basé sur l'utilisation de *chroniques* [18].

Pour chaque occurrence d'événement dans le système à l'exécution, nous construisons une représentation de la *situation* associée (*i.e.* l'état des variables concernées et quelques éléments d'informations sur l'action effectuée) et nous enregistrons cette situation. Cela constitue des chroniques qui sont ensuite *intensionalisées* pour en extraire les régularités. Nous avons proposé quelques outils d'interrogation de ces chroniques (voir [18]) pour servir de base à un modèle de raisonnement fondé sur l'analyse dynamique des composants. L'une des originalités de ce travail est d'utiliser au sein d'une même représentation une description statique des actions (comme dans le calcul des situations [13]) et une mémorisation de l'exécution passée sous forme de chroniques [4], comme McCarthy le propose dans Elephant 2000 [12].

Nous travaillons actuellement sur l'amélioration de ce modèle de chroniques afin de réduire la complexité de la recherche et de répondre plus efficacement aux questions temporelles.

1.3 Un modèle de requêtes sur le fonctionnement

Notre objectif dans la définition d'un modèle de requêtes sur le fonctionnement est double : il s'agit d'abord de faire le lien entre les outils de traitement de la langue naturelle (MLN) et les outils de raisonnement sur le fonctionnement du composant (MRF) à l'aide de requêtes formelles d'ordre procédural, c'est-à-dire portant sur la structure dynamique ou l'exécution de composants.

Mais nous souhaitons aussi proposer un modèle de raisonnement sur le fonctionnement des composants actifs basé sur la notion de causalité [3, 1] et qui serve de support au traitement de ces requêtes.

Pour cela, nous nous appuyons sur un modèle de représentation [17] qui permet de connaître à tout instant la structure statique et dynamique du com-

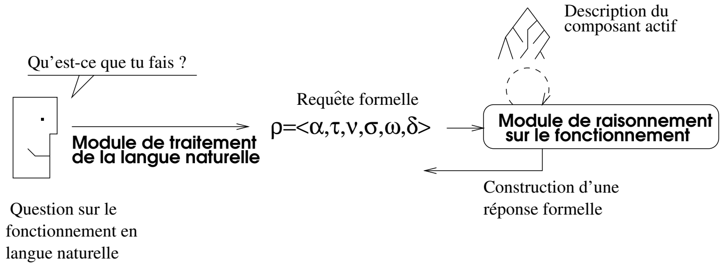

FIG. 1: Traitement des questions sur le fonctionnement dans le projet *InterViews*.

posant. La figure 1 illustre ce mécanisme d'interaction et de raisonnement.

Nous allons tout d'abord présenter le formalisme de description des composants que nous utilisons pour notre étude. Nous proposons ensuite un modèle de requêtes sur le fonctionnement et nous donnons le principe général de construction d'une réponse formelle. Ces requêtes sont classées en fonction des différents mécanismes de réponse qu'elles utilisent et des problèmes qu'elles soulèvent sur le plan procédural. Enfin, nous essayons de formaliser certains de ces problèmes et nous proposons quelques éléments de solution.

2 Présentation de VDL

2.1 Principe général

Dans le projet InterViews, les composants sont décrits dans un langage spécifique, appelé VDL pour ***V****iew* ***D****esign* ***L****anguage*. Le corps d'un composant, c'est-à-dire la description en VDL du monde opératoire qu'il manipule, est un arbre dont les noeuds sont des chaînes de caractères, appelées *concepts*. Nous appelons *vue*[1] cet arbre de concepts.

L'exécution des composants consiste en la réécriture de la vue à chaque *cycle*, comme dans les algèbres évoluantes de Gurevich [10] : nous définissons une fonction ϕ_{exec} qui recherche dans la vue v_t des concepts particuliers qu'elle interprète pour construire la vue v_{t+1}. Les composants sont donc proactifs : ils peuvent *s'exécuter* en dehors de toute interaction avec l'utilisateur.

Nous avons défini complètement la sémantique opérationnelle de ce langage dans [17] et nous l'avons implémenté en Java 1.1.7[2].

Syntaxe

Dans la suite, nous appelons *terme* tout sous-arbre de la vue et, pour un terme donné, nous appelons *attributs* ses sous-termes directs. Les vues seront représentées sous la forme suivante : tout terme de concept c et d'attributs

[1]par analogie avec les travaux de Forbus [8] sur la représentation d'un composant actif.
[2]Des démonstrations sont disponibles sur : http ://www.limsi.fr/Individu/nico/exemples

$st_1, ..., st_n$ sera noté $c[st_1, ..., st_n]$. Les concepts contenant des symboles réservés (crochets, virgules, espaces) seront écrits entre guillemets.

Enfin, nous notons A^* l'ensemble des concepts et Υ l'ensemble des termes VDL.

2.2 Concepts procéduraux

Nous ne pouvons pas présenter ici l'ensemble des concepts utilisés par la fonction ϕ_{exec} pour définir la sémantique opérationnelle des vues. Nous rappelons simplement les principes de base développés dans [17].

2.2.1 Actions de base

Nous avons montré qu'il existe seulement trois actions élémentaires de modification d'une vue formant une base : **add** pour ajouter des attributs à un terme de la vue, **put** pour remplacer tous les attributs d'un terme par d'autres et **del** pour enlever un sous-terme de la vue.

2.2.2 Concepts arithmétiques

Nous avons aussi défini un certain nombre de concepts correspondant aux opérations arithmétiques de base : **plus** et **times** pour l'addition et la multiplication, **minus** et **inverse** pour l'opposé et l'inverse ; **equals** et **"greater than"** pour les relations entre des termes ; **and**, **or**, **not**, **true** et **false** pour les opérateurs booléens.

2.2.3 Processus et événements

Pour modéliser l'interaction du composant avec l'utilisateur ou avec d'autres composants, nous avons proposé un mécanisme *d'événements externes*. Un événement externe est un terme VDL qui peut être envoyé au composant au cours de son exécution.

Le concept **event** permet de donner l'ensemble des événements auxquels le composant doit réagir. Il permet donc de définir des *actions de réaction aux événements externes*.

Mais les composants peuvent aussi fonctionner de manière *proactive*. Le concept **guard** permet de définir les gardes booléennes des *processus*. La combinaison de ces deux concepts (*event* et *guard*) dans les termes décrivant des actions permet de décrire des actions de réaction aux événements externes munis de gardes. De nombreux exemples sont donnés dans [17].

2.3 Définitions

- Tout terme de concept $c \in \{add, put, del\}$ est une ***action élémentaire***.
- Tout terme ayant au moins un attribut de concept *guard*, *event*, *add*, *put* ou *del* est une ***action***.

– L'ensemble des ***préconditions*** d'une action a est :

$$\Pi_a = \{t \in \pi(a) \text{ tq } concept(t) \in \{guard, event\}\}$$

où $\pi(a)$ est l'ensemble des "oncles" de a, défini récursivement par :

$$\pi(a) = \{t \text{ tq } p\grave{e}re(a) = p\grave{e}re(t)\} \cup \pi(p\grave{e}re(a))$$

– Toute action a telle que $\forall t \in \Pi_a, concept(t) = guard$ (*i.e.* n'ayant aucun événement externe dans ses préconditions) est un ***processus***.
– Le ***comportement*** d'un composant est défini par l'ensemble des actions.
– L'***activité*** d'un composant est définie par l'ensemble de ses processus.

2.4 Exemple : *Coco*

Dans cet article, nous utiliserons souvent un exemple volontairement assez simple. Il s'agit d'un compteur appelé *Coco* dont la vitesse peut être modifiée à l'aide d'événements externes. Sa description en VDL est donnée ci-dessous (les concepts procéduraux sont en italique pour faciliter la lecture) :

```
view[value[0], running[true], speed[1],
      count[ guard[get[running]],
             put[path[value],plus[get[value],get[speed]] ],
      stop[ event[stop], put[path[running],false] ],
      start[ event[start], put[path[running],true] ],
      "speed up"[ event[faster],
                  put[path[speed],plus[get[speed],1]] ],
      "slow down"[ event[slower],
                   put[path[speed],plus[get[speed],-1]] ]
]
```

3 Un modèle de requêtes sur le fonctionnement

Dans cette section, nous présentons brièvement le modèle formel de requêtes que nous utilisons pour représenter et répondre à une large classe de questions sur le fonctionnement. Ce modèle est décrit plus en détail dans [19].

3.1 Présentation du modèle

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

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

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. Ces critères sont expliqués dans la section suivante.

La réponse à une requête sur le fonctionnement est aussi une requête. Nous donnons quelques exemples dans la section 3.3.

3.1.1 L'acte : α

Le critère α est *l'acte* de la requête. Chaque acte de requête correspond à une interrogation particulière, de même que les *actes de langage* [21] en traitement de la langue naturelle définissent l'interprétation d'un dialogue. L'acte de requête définit donc le cadre de la question. Pour les questions sur le fonctionnement, nous avons les actes suivants :

- *What* pour indiquer que l'objet de la question (donné par ω) est inconnu (par exemple : "*Que fais-tu ?*").
- *Ask* pour indiquer que l'objet de la question tient lieu de résultat supposé (par exemple : "*Est-ce que tu vaux 42 ?*" ou "*Est-ce que tu comptes ?*")
- *Why* pour les demandes de *diagnostic* de l'exécution passée du composant (par exemple : "*Pourquoi est-ce que tu t'es arrêté ?*")
- *How* pour les demandes de construction d'un *plan* (par exemple : "*Comment faire pour que tu t'arrêtes ?*") ou l'étude du corps d'une action (par exemple : "*Comment fais-tu pour compter ?*").
- *When* pour les requêtes temporelles, correspondant à une recherche d'état ou d'action dans l'exécution passée ou future du composant (par exemple : "*Quand valais-tu 42 ?*"). Ces requêtes ne seront pas étudiées ici.
- *Assert* pour affirmer un résultat. Dans le cadre des *questions* sur le fonctionnement, cet acte n'est utilisé que pour les réponses construites par le MRF et, de manière générale, les réponses du MRF ont toutes pour acte $\alpha = Assert$.
- *Error* pour signifier que la requête fournie est syntaxiquement incorrecte, *Unknown* pour signifier qu'aucun sous-terme de la vue ne correspond à l'un des arguments proposés dans le critère ω ($\forall t' \in \omega, t' \notin subterms(v_t)$).

Ces deux derniers actes ne sont utilisés que dans les réponses et en cas d'erreur.

3.1.2 Le type : τ

τ est le *type procédural* de la requête. Il permet de déterminer le type d'élément procédural (action, processus, *etc.*) sur lequel porte la requête. Pour les questions sur le fonctionnement, nous avons les types suivants :

- *is* pour indiquer que la question porte sur un état (une *situation* suivant la terminologie de Reiter [16]). Par exemple, pour "*Pourquoi vaux-tu 47 ?*", $(\alpha, \tau) = (Why, is)$.
- *do* pour indiquer que la question porte sur l'activité de la vue (les *processus*). Par exemple, pour "*Comment faire pour que tu comptes ?*", $(\alpha, \tau) = (How, do)$.
- *order* pour indiquer que la question porte sur les réactions aux événements externes. Par exemple, pour "*Qu'est-ce que je peux te demander de faire ?*", $(\alpha, \tau) = (What, order)$.
- *can* pour indiquer que la question porte sur les actions possibles. Par exemple, pour "*Est-ce que tu peux compter ?*", $(\alpha, \tau) = (Ask, can)$.

A chaque couple (α, τ) correspond un type de question particulière et un traitement spécifique pour le MRF.

3.1.3 Le booléen ν

Le critère $\nu \in \{\top, \perp\}$ est un marqueur booléen. Sa sémantique dépend du cadre de la question, défini par (α, τ) . Par exemple, lorsque $\tau = do$, ν est généralement utilisé pour préciser si la question porte sur des processus actifs ($\nu = \top$) ou inactifs ($\nu = \perp$).

3.1.4 Le sujet : σ

$\sigma \in \mathcal{P}(\Upsilon)$ est une référence vers le *sujet* de la question, *i.e.* le sous-terme de la vue sur lequel porte la question. Par exemple, pour "*Est-ce que ta valeur est 47 ?*", $\sigma = \{value\}$. Le module de traitement de la référence garantit l'exactitude de σ.[3]

Comme dans [17], une référence est un ensemble de termes qui constitue un chemin partiel dans la vue. Nous noterons $calcRef(r, v)$ le résultat de la référence r dans la vue v.

Remarque : Les requêtes, qu'il s'agisse de question ou de réponses, sont toujours exprimées par rapport à la vue. Ainsi, pour la question "*Est-ce que tu comptes*" et la réponse "*Oui, je comptes*", nous aurons le même $\sigma = \{view\}$, correspondant au "*tu*" dans la question et au "*je*" dans la réponse.

3.1.5 L'objet : ω

$\omega \in \mathcal{P}(\Upsilon)$ est *l'objet* de la question. C'est un ensemble de termes VDL correspondant aux paramètres de la question. Ils ne sont pas nécessairement présents dans la vue, comme nous le verrons dans les exemples donnés section 3.3.

Par exemple, pour "*Tu comptes ou tu dors ?*", $\omega = \{count, sleep\}$.

3.1.6 La date : δ

$\delta \in \Upsilon$ est la *date* sur laquelle porte la question, exprimée en VDL.

Nous avons retenu plusieurs formes pour les dates qui permettent de décrire des instants passés, présents ou futurs, dans l'absolu ou relativement à d'autres dates ou à des occurrences d'actions. En particulier, $\delta = \surd$ pour le présent

Le mécanisme de réponse à des questions datées ($\delta \neq \surd$) ou temporelles ($\alpha = When$) s'appuie sur l'utilisation de chroniques, comme nous l'avons expliqué section 1.2 et n'est pas abordé dans cet article.

[3]Le traitement de la référence dans le cadre d'un dialogue homme-machine non contraint reste aujourd'hui un problème ouvert. Malgré cela, nous supposerons pour notre étude que σ a été correctement renseigné par le MLN.

3.2 Construction des réponses

L'objectif du MRF est de construire des réponses aux requêtes formelles du MLN. Toutefois, les mécanismes mis en œuvre ne peuvent pas être présentés en détail ici, compte tenu de leur aspect technique. Ils sont décrits de manière plus précise dans [19].

3.2.1 Principe général

Lors de la construction d'une réponse à une requête sur le fonctionnement, le MRF recherche les valeurs de variables ou les actions (selon la valeur de τ) dans les sous-termes de σ, en utilisant ω comme filtre.

Si $\alpha \in \{What, Ask\}$, la réponse est alors de même type (τ), d'acte $\alpha = Assert$ et son objet ω est le résultat de cette recherche. Nous donnons quelques exemples de ces mécanismes, dans le cas $\delta = \surd$, dans la section 4.

3.2.2 Génération de plans

Si $\alpha = How$, le MRF génère un plan et la réponse est l'expression, sous la forme d'un ensemble de requêtes, des différentes actions de ce plan.

Notre méthode de génération de plan est similaire à celle proposée dans STRIPS [6], fondée sur la technique de régression. Les états correspondent à des vues VDL et les actions sont décrites en utilisant le formalisme VDL comme un ensemble de *add*, *put* et *del*.

3.2.3 Génération d'explications

Si $\alpha = Why$, la réponse est l'expression des différentes valeurs et actions constituant l'explication de l'état ou de l'action de la requête (selon τ).

Les outils classiques de diagnostic construisent un ensemble de littéraux qui donne les causes d'un état. Au contraire, le *diagnostique explicatif* [14] construit une séquence d'action, correspondant à "ce qu'il s'est passé".

Le mécanisme d'explication des actions qui est actuellement en développement dans le MRF, fondé sur la recherche de "*conséquences fixes*" d'actions[4] et l'étude de la *causalité* dans le corps des actions, tend à unifier ces deux approches. Notre objectif est d'être capable de d'exprimer sous forme de requête les causes (qu'il s'agisse de situations ou d'actions) d'une action ou, indifféremment, d'un état.

3.3 Quelques exemples

Afin de clarifier notre propos, nous donnons ci après deux exemples représentatifs de requêtes produites par le MLN à partir de questions sur le fonctionnement de *Coco*.

[4]Nous appelons *conséquences fixes* les résultats d'une action ne dépendant pas de la situation. Elles sont aisément calculées à l'aide d'un algorithme qui recherche les actions élémentaires (*add*, *put* et *del*) et calcule leurs résultats.

– "*Que fais-tu ?*" : $\rho =< What, do, \top, \{view\}, \emptyset, \surd >$
  Le MRF recherche les processus actifs et construit la réponse : $\overline{\rho} =< Assert, do, \top, \{view\}, \{count\}, \surd >$, qui produira : "*Je compte.*".
– "*Pourquoi comptes-tu ?*" : $\rho =< Why, can, \top, \{view\}, \{count\}, \surd >$
  Le MRF recherche les préconditions invalides de l'action *count* et construit la réponse : $\overline{\rho} =< Assert, is, \top, \{running\}, \{true\}, \surd >$, qui pourra être utilisé pour produire : "*Parce que je suis en marche.*".

Chaque couple (α, τ) correspond à un type de requête particulier et peut générer plusieurs traitements spécifiques. Nous ne pouvons pas tous les présenter ici.

4 Six problèmes importants

Dans les exemples de la section précédente, nous nous sommes volontairement limité aux cas généraux. Nous présentons maintenant quelques problèmes de raisonnement qui apparaissent lors du traitement des requêtes par le MRF. Nous ne prétendons pas avoir résolu ces problèmes, mais nous essayerons de montrer comment notre modèle les prend en compte et comment il peut être utilisé pour rechercher des solutions.

4.1 Problèmes de référence dans ω

Nous avons vu section 3.1 que le MLN doit garantir l'existence et l'unicité du sujet σ de la requête.

Mais nous sommes aussi confrontés à un problème de référence lorsque $\tau \in \{do, can\}$, puisque les éléments de ω sont des références vers des actions possibles de la vue. Pour toute référence $r \in \omega$, si $|calcRef(r, v_t)| \neq 1$ (i.e. le résultat de la référence est nul ou, au contraire, multiple), alors nous devons résoudre un problème spécifique dont la solution dépend de la requête elle-même.

Par exemple, si la question "*Tu comptes ou tu dors ?*" est posée à *Coco*, elle est traduite par la requête :

$$\rho =< Ask, do, \top, \{view\}, \{sleep, count\}, \surd >$$

pour laquelle $calcRef(\{sleep\}, v_t) = \emptyset$.

Dans l'implémentation actuelle des algorithmes du MRF, nous avons retenu la solution partielle suivante :

– si $\forall r \in \omega, calcRef(r, v_t) \neq 1$, alors $\overline{\rho} =< Unknown, is, \top, \sigma, \omega, \delta >$
– sinon, soit $\omega' = \{r \text{ tq } calcRef(r, v_t) \neq 1\}$. Soit ω_ν l'ensemble des processus (ou actions si $\tau = can$) dans l'état ν et $\omega_{\neg\nu}$ l'ensemble des processus (ou actions) dans l'état inverse. Si $\omega_\nu \neq \emptyset$, $\overline{\rho} =< Assert, do, \nu, \sigma, \omega_\nu, \delta >$. Sinon, $\overline{\rho} =< Assert, do, \neg\nu, \sigma, \omega_{\neg\nu}, \delta >$.

4.2 Référence procédurale

Nous voudrions que l'utilisateur puisse référencer les actions en utilisant non seulement leur nom, mais aussi le "comportement perçu". C'est pourquoi les éléments de ω peuvent être des termes VDL complexes.

Par exemple, un utilisateur peut demander à un compteur "*Quelle est l'action qui incrémente la valeur ?*". Cette action correspond à la description VDL :

$$put[path[value], plus[get[value], \#1]]$$

où #1 est un terme quelconque. Par conséquent, la requête formulée devrait être :

$$\rho =< How, do, \top, \{view\}, \{put[path[value], plus[get[value], \#1]]\}, \surd >$$

Nous avons implémenté des algorithmes de *pattern-matching* dans le MRF qui permettent de rechercher dans une vue un terme VDL **quelconque**, simple concept ou arbre complet, éventuellement avec des paramètres $\#i$ (numérotés pour les algorithmes de *pattern-matching*).

4.3 Intensionnalisation des actions

4.3.1 *Ground* de concepts

Nous avons expliqué section 1.1 que le MLN n'utilisera pas les concepts procéduraux. Il travaille avec des concepts plus généraux représentant des notions de la langue naturelle comme "*incrémenter x*" (représentable en VDL par $increase[x, \#1]$). Dans le projet *InterViews*, nous appelons "concepts de *ground*" ces macro-concepts, proches de la langue naturelle.

C'est pourquoi nous devons développer des mécanismes d'intensionnalisation et d'extensionnalisation qui permettent de passer de la représentation en terme de *ground* à la représentation VDL :

$$\xi(put[path[\#1], plus[get[\#1], \#2]]) \equiv increase[\#1, \#2]$$

4.3.2 *Ground* de relations

D'autre part, pour pouvoir interpréter et expliquer certaines actions comme "*compter à l'envers*", il faut munir le *ground* de *relations* sur des actions VDL, correspondant à des contraintes procédurales. Par exemple, le MRF devrait pouvoir déterminer que *stop* est l'inverse de *start*. De même, si l'on considère la question "*Pourquoi est-ce que tu comptes à l'envers ?*" qui correspond à la requête :

$$\rho =< Why, do, \top, \{view\}, \{backward[count]\}, \surd >$$

dans laquelle *backward* est une relation unaire. Le MRF devrait savoir que $backward[increase[x, y]] = \top$ si et seulement si $y < 0$. Cette relation est représenté dans le *ground* par :

$$backward[\, increase[\#1, \#2], \, ''greater\, than''[0, \#2]\,]$$

Ainsi, $backward[count]$ correspond à $speed < 0$.

De même, le concept *nothing* que nous avons vu section 3.3 est aussi représenté par une relation de *ground*.

Lors du traitement des requêtes, il est nécessaire de détecter les relations de *ground* dans la question et de vérifier les contraintes associées (ex : $y < 0$) pour construire la réponse.

4.4 Le problème de la ramification

Le problème de la ramification, énoncé par Finger [7], est l'impossibilité de donner toutes les conséquences d'une action. En VDL comme dans tous les langages de programmation, les conséquences "indirectes" sont calculées automatiquement dans les actions effectives du composant, contrairement aux approches logiques où se pose le problème de leur description.

Le problème de la ramification est tout de même présent dans les questions sur le fonctionnement : un utilisateur humain aura souvent tendance à décrire une action par ses conséquences. Le MRF doit alors "remonter les ramifications", c'est-à-dire retrouver l'action responsable d'une modification dont parle l'utilisateur.

- La première difficulté est de déterminer quelle action est la véritable cause [15] de la modification d'un terme, *i.e.* jusqu'à quelle action remonter.
- La seconde est que les conséquences d'une action peuvent être le déclenchement d'autres actions. Par exemple, le traitement de la requête : $\rho =< Why, do, \top, \{view\}, \{count\}, \surd >$ ("*Pourquoi est-ce que tu comptes ?*") nécessite de rechercher les actions dont les conséquences sont de valider la garde de l'action *count*.[5]

Nous étudions actuellement un mécanisme d'analyse du corps de la vue fondé sur des approches logiques d'analyse statique [2, 1] qui permettrait de trouver les causes en remontant les ramifications.

4.5 Résolution de *can*

4.5.1 Présentation du problème

Lorsque $(\alpha, \tau) = (Ask, can)$, il faut résoudre un problème d'ambiguïté. En effet, *pouvoir* faire une action peut signifier :

- en avoir la possibilité physique : l'action est présente dans la vue sous la forme d'un *concept* lié par une relation *WordNet*,
- en avoir la possibilité morale : les préconditions de l'action sont vérifiées.

La puissance de notre approche vient du fait que le MRF est capable de lever cette ambiguïté. La recherche d'action dans la vue permet en effet de déterminer

[5]Le MLN dispose d'informations sur la nature des termes de la vue (s'il s'agit d'actions, quelles sont les termes qu'ils modifient, *etc*). Ces informations sont utilisées pour empêcher l'apparition de requêtes "mal typées", comme : $\rho =< why, is, \top, \{view\}, \{stop\}, \surd >$ (*stop* est en effet une action et ne peut donc pas être utilisée comme paramètre ω d'une requête de type $\tau = is$).

quelles sont les actions connues, puis le calcul des préconditions permet à la fois de déterminer quelles sont les actions possibles et de *définir le cadre de la question* (physique ou moral).

4.5.2 Une solution fondée sur la recherche de préconditions

Une solution envisageable est donc de construire les ensembles $\Pi_\top$ et $\Pi_\perp$ des préconditions des actions recherchées qui sont ($\Pi_\top$) ou qui ne sont pas ($\Pi_\perp$) vérifiées.

- Si $\Pi_\perp = \emptyset$, nous sommes dans le premier cas (possibilité physique). La réponse est alors $\overline{\rho} =< Assert, can, \top, \sigma, \omega', \delta >$.
- Si $\Pi_\perp \neq \emptyset$:
  - si ($\nu = \top$ et $\Pi_\top \neq \emptyset$) ou ($\nu = \perp$ et $\Pi_\top = \emptyset$), nous sommes dans le second cas (possibilité morale) et la réponse est positive pour certaines actions : $\overline{\rho} =< Assert, can, \nu, \sigma, \omega'', \delta >$, où ω'' est l'ensemble des actions dont les préconditions ont pour valeur ν.
  - sinon $\Pi_\top = \emptyset$, nous sommes dans le cas de possibilité morale avec réponse négative. Il faut alors fournir des explications, *i.e.* un ensemble de préconditions invalides (pour ν).

Par exemple, à la question $\rho =< Ask, can, \top, \{view\}, \{count\}, \surd >$ ("*Est-ce que tu peux compter ?*"), nous répondrons :

- $\rho =< Assert, can, \top, \{view\}, \{count\}, \surd >$ ("*Je peux compter.*") lorsque la garde de *count* est valide,
- $\rho = \{< Assert, can, \perp, \{view\}, \{count\}, \surd >, < Assert, is, \perp, \{running\}, \{true\}, \surd >\}$ ("*Je ne peux pas compter : je ne suis pas en marche.*") sinon. La réponse est donc une requête multiple, *i.e.* un ensemble de requêtes $\overline{\rho_i}$ telles que :

$$\begin{aligned} \rho_1 &= < Assert, can, \neg\nu, \sigma, \omega, \delta > \\ \overline{\rho_{i\geq 2}} &= < Assert, is, \perp, \overline{\sigma_i}, \overline{\omega_i}, \delta > \end{aligned}$$

  Chaque $\overline{\rho}_{i\geq 2}$ exprime l'invalidité d'une seule précondition ($\overline{\sigma_i}$) de $\Pi_{\neg\nu}$.

4.6 *What* avec $\omega \neq \emptyset$

Les requêtes dont l'acte est *what* ont généralement un objet ω vide (par exemple, $\rho =< What, do, \top, \{view\}, \emptyset, \surd >$ pour "*Qu'est-ce que tu fais ?*"). Il existe deux exceptions qui sont chacune munies d'une sémantique particulière :

- Lorsque $\tau = can$, les éléments de ω sont des actions (référencées généralement par leur concept) dont on recherche les arguments. Par exemple, dans un composant muni d'une action *take*, la question "*Qu'est-ce que tu peux prendre.*" sera traduite en :

  $\rho =< What, can, \top, \{view\}, \{take\}, \surd >$

  La réponse sera un ensemble de requêtes de la forme :

  $\overline{\rho_i} =< Assert, can, \top, \{view\}, \{take[x_i]\}, \surd >$

où x_i est un argument possible de l'action $take$ (associée à l'événement externe $take$).

- Lorsque $\tau = do$, les éléments de ω sont des actions (référencées généralement par leur concept) pour lesquelles il faut déterminer quels sont les sous-termes qu'elles modifient.

  Par exemple, la question "*Qu'est-ce que tu comptes.*" sera traduite en :

  $$\rho =< What, do, \top, \{view\}, \{count\}, \surd >$$

  et la réponse sera $\overline{\rho} =< Assert, do, \top, \{view\}, \{count[value]\}, \surd >$.

5 Conclusion et perspectives

Nous avons proposé dans cet article un modèle de requêtes permettant de formaliser une large classe de questions qu'un utilisateur humain peut poser sur les actions et le fonctionnement d'un composant actif. Ce modèle de requêtes est utilisé au sein du projet *InterViews* pour relier les outils de raisonnement sur le fonctionnement avec les outils de traitement de la langue naturelle.

Notre objectif est d'y intégrer rapidement les éléments de solution sur le fonctionnement que nous avons évoqués ici sans les traiter (chroniques et ordres de l'utilisateur en langue naturelle) et qui sont encore en cours d'étude.

Les mécanismes que nous utilisons au sein du MRF sont suffisamment génériques pour répondre à des questions portant sur n'importe quel composant actif défini dans notre formalisme VDL [17, 20]. Mais comme nous l'avons vu, la construction de réponses à ces requêtes formelles fait apparaître de nouveaux problèmes de raisonnement sur le fonctionnement. Nous pensons que l'étude de ces problèmes ouvre de nouvelles perspectives dans ce domaine et plus particulièrement dans le domaine de l'explication en langue naturelle de l'activité des composants.

Références

[1] 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.

[2] 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.

[3] R. DEMOLOMBE. Actions et causalité : essais de formalisation en logique. In *Le temps, l'espace et l'évolutif, Ecole thématique du GDR I3*, volume 2, pages 209–223. 2000.

[4] O. DESPOUYS, F. INGRAND, M. GHALLAB, and J.P. GOUYON. Extension d'IxTeT pour son application à la supervision d'un réseau électrique. In *Proc. RFIA'98*, pages 265–274, 1998.

[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] J.J. FINGER. *Exploiting Constraints in Design Synthesis.* PhD thesis, Stanford, 1987.

[8] K.D. FORBUS. Qualitative Process Theory. *Artificial Intelligence*, 24 :85–167, 1984.

[9] S. GERARD and J.P. SANSONNET. A spatio-Temporal Model for the representation of Situations Described in Narrative Texts. In *Proc. NLP2000*, Patras, June 2000.

[10] Y. GUREVICH. Evolving Algebra Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[11] 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.

[12] J. McCARTHY. Elephant 2000 : A programming language based on speech acts. Technical report, Stanford University, 2000. draft version.

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

[14] 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.

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

[16] R. REITER. *Knowledge in Action : Logical Foundations for Describing and Implementing Dynamical Systems.* Draft of book available at http ://www.cs.toronto.edu/cogrobo/book.ps.Z, 1999.

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

[18] N. SABOURET and J.P. SANSONNET. Un modèle de raisonnement sur les systèmes évolutifs fondé sur l'intensionnalisation de chroniques. In *Proc. RFIA 2000*, volume 2, pages 305–314, 2000.

[19] N. SABOURET and J.P. SANSONNET. Un modèle de requêtes sur le fonctionnement de composants actifs. A paraître dans MFI'01, 21-23 mai 2001.

[20] J.P. SANSONNET. Présentation de VDL 0.1. Technical Report 99-10, LIMSI-CNRS, 1999.

[21] J.R. SEARLE. *Speech Acts.* Cambridege University Press, 1969.

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