



HAL
open science

Étude de modèles de représentations, de requêtes et de raisonnement sur le fonctionnement des composants actifs pour l'interaction homme-machine

Nicolas Sabouret

► **To cite this version:**

Nicolas Sabouret. Étude de modèles de représentations, de requêtes et de raisonnement sur le fonctionnement des composants actifs pour l'interaction homme-machine. Interface homme-machine [cs.HC]. Université Paris Sud - Paris XI, 2002. Français. NNT : . tel-00002979

HAL Id: tel-00002979

<https://theses.hal.science/tel-00002979>

Submitted on 11 Jun 2003

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



N° D'ORDRE : 7104

UNIVERSITÉ PARIS XI
UFR SCIENTIFIQUE D'ORSAY

THÈSE

Présentée

Pour obtenir

**Le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS XI ORSAY**

PAR

Nicolas SABOURET

Sujet : Étude de modèles de représentation, de requêtes et de raisonnement sur le fonctionnement des composants actifs pour l'interaction homme-machine

Soutenue le 19 décembre 2002 devant la commission d'examen :

M.	Jean-Pierre BRIOT	
M.	Luis FARIÑAS del CERRO	
M.	Andreas HERZIG	(Rapporteur)
Mme	Marie-Christine ROUSSET	
M.	Jean-Paul SANSONNET	(Directeur de thèse)
M.	Stuart SHAPIRO	(Rapporteur)

Résumé

Nous abordons dans cette thèse un problème émergeant dans la problématique de l'interaction homme-machine : celui de la construction de réponses à des requêtes formelles modélisant des questions de bon sens posées par les utilisateurs ordinaires concernant le fonctionnement d'un composant logiciel actif.

Les recherches actuelles dans la communauté du raisonnement sur les actions proposent de travailler sur une modélisation du système en logique pour répondre à des requêtes bien formées et ne prennent pas en compte l'existence de « *notions de bon sens* » utilisées par les utilisateurs ordinaires dans la formulation des questions. Nous voulons au contraire pouvoir travailler directement sur le code du composant, et répondre à des requêtes faisant intervenir des notions de bon sens.

Nous présentons tout d'abord un langage de programmation et un modèle d'exécution spécifique pour les composants actifs, qui permet d'avoir accès en cours d'exécution à une description des actions effectuées par le composant et des données manipulées. Nous présentons ensuite un langage de requêtes qui permet de modéliser simplement une large classe de questions sur le fonctionnement issues de l'interaction en langue naturelle. Nous n'abordons pas les problèmes spécifiques au traitement automatique du langage naturel (TALN), soulevés par la production de ces requêtes. Nous proposons des algorithmes de réponse qui permettent de traiter ces requêtes. Nous montrons qu'il est possible de prendre en compte des notions de bon sens dans les requêtes et nous proposons un cadre formel pour le traitement de ces *requêtes de bon sens sur le fonctionnement*.

Nous étudions ensuite le problème de la construction de réponses portant sur l'exécution passée du composant. Nous montrons qu'il est possible et nécessaire d'utiliser une approche *dynamique*, pour extraire des régularités dans les actions du composant et mettre en évidence des comportements, individuels dans le cas d'une interaction avec l'utilisateur ou collectifs dans le cas d'une interaction dans un système multi-agents, nécessaires à la production d'une explication en réponse aux questions de l'utilisateur.

Nous montrons que nos travaux peuvent s'intégrer dans le cadre du *web sémantique* pour définir des composants actifs sous la forme de pages web dynamiques munies de capacité de raisonnement et d'interaction avec les utilisateurs ordinaires, par exemple pour des tâches d'assistance. Nous présentons enfin l'implémentation de notre modèle qui a été réalisée dans le cadre du projet *InterViews* et nous donnons des exemples de programmation et d'exécution de ces composants actifs

Abstract

In this PhD thesis, we deal with a problem that is emerging in human-computer interaction. Our aim is to answer formal requests modelling *common-sense* questions that a human user can ask about an active software component's functioning.

Current research in reasoning about actions propose to work with a logic model of the system and to answer only to well formed requests. However, such requests cannot take into account common-sense notions that can be used by human users in their questions about actions. On the contrary, we want to be able to work directly with the component's code and to answer requests that involve common-sense notions about actions.

We first present a programming language and a runtime model for active components that allows us to have access, at any time of the execution, to the description of the component's actions and physical status. We then present a request model that can be used to represent a wide class of questions about actions coming from natural language interaction. We do not study the natural language processing issues underlying the production of such requests. We rather propose answering algorithms for use on *formal* requests, derived from natural language questions. We show that it is possible to take into account common-sense notions in these requests and we propose a formal frame for processing such "*common-sense requests*" about actions.

Then, we study the issue of answering requests about the component's past execution. We show that using a *dynamic* approach is both possible and required, so as to extract regularities from the component's runtime. We show how this exhibits component *behaviours* that are required for the production of an explanation, in answering the user's questions. We present an extension of this approach for extracting collective behaviours in multi-agent systems.

We show that our work can be used within the *semantic web* in order to define active components as dynamic page web, provided with reasoning capabilities and capable of interacting with ordinary users in need for help. Finally, we present the implementation of our model within the *InterViews* project. We give both programming and execution examples of active components in this implementation.

Remerciements

Ce document constitue pour moi l'aboutissement de trois années et demi de travail intense et difficile. Il n'aurait cependant pas été possible sans l'aide de tous les gens dont j'ai croisé la route au cours de ma thèse, qui m'ont guidé dans mes réflexions, qui m'ont soutenu au quotidien et qui m'ont encouragé dans les moments de doute.

Je voudrais tout d'abord remercier mes rapporteurs, MM. Herzig et Shapiro, qui ont eu le courage de se plonger dans ma thèse. Leurs remarques très pertinentes m'ont permis d'améliorer mon manuscrit et d'étendre, pour plusieurs années, ma réflexion. Je remercie également les autres membres du Jury, Mme Rousset et MM. Briot et Fariñas del Cerro, d'avoir accepté de faire partie de mon jury et d'avoir montré tant d'intérêt pour mes travaux.

Merci à Joseph Mariani et à Patrick Le Quéré pour m'avoir accueilli dans le laboratoire qu'ils dirigent. C'est grâce à leurs efforts quotidiens que les doctorants du LIMSI peuvent travailler dans d'aussi bonnes conditions. Merci à Gérard Sabah, qui m'a accueilli dans le groupe Langage & Cognition, pour son soutien et sa sympathie.

Merci à Jean-Paul Sansonnet qui, pendant ces trois années, a su me guider dans mes recherches avec discernement. Merci pour sa disponibilité au quotidien, ses capacités d'adaptation et d'analyse, ainsi que pour sa générosité à les faire partager. Son immense culture a été pour moi une grande source d'inspiration et de motivation tout au long de ma thèse.

Je voudrais également remercier les membres du laboratoire avec lesquels j'ai travaillé. Merci d'abord à tous ceux dont j'ai partagé le bureau, Khedija, Sébastien, Érika, Jean-Baptiste, Yann, Jean-Claude, Guillaume, Haïfa, Stéphanie, Sarkis, Ouriel, avec qui j'ai connu d'agréables moments... et aussi quelques fous-rires. Merci aux membres des groupes AMI, LIR et L&C pour les échanges enrichissants que nous avons eus tant sur le plan scientifique que sur le plan humain. Leurs conseils à tous m'ont souvent été d'un grand secours au long de ces trois années de recherche. Je voudrais remercier tout particulièrement Laura, Caroline et Christian pour leur amitié, qui va bien au-delà du cadre du laboratoire; Anne pour sa disponibilité, l'attention qu'elle m'a accordée et les conseils qu'elle m'a prodigués au quotidien; Jean-Baptiste pour ses réflexions pertinentes et pour son immense culture, qu'il a bien voulu partager avec moi.

Merci à ceux qui nous permettent chaque jour de travailler dans de bonnes conditions. En particulier, merci à Bernard, Michel, Élisabeth, Martine et Joëlle pour la confiance qu'ils m'ont accordée et pour la qualité des relations que nous avons sur le plan personnel. Merci à Nédé qui prépare patiemment, jour après jour, les chocolats chauds et les jus d'orange qui me permettent de bien démarrer la journée.

Plus généralement, je voudrais remercier l'ensemble des collègues du LIMSI, du département CHM, du département Mécanique Énergétique et de l'équipe direction, qui contribuent chacun à créer et à entretenir une ambiance de travail unique au laboratoire, fondée sur l'échange. Je voudrais remercier plus particulièrement les membres du conseil de laboratoire, Gilles, Annick, Pascal, Rachid, Laurie, François,

Michel, Phillipe, Marie-Christine, Catherine, pour l'expérience qu'ils m'ont apportée. Je voudrais remercier plus particulièrement Patrick Le Quéré pour la confiance qu'il m'a accordée. Merci aussi aux doctorants qui se sont impliqués, avec moi, dans la vie du LIMSI : Amandine, Othman, David, Linda et Guillaume.

Enfin, je voudrais remercier ma famille et mes amis qui, même s'ils n'ont pas toujours compris ce que je « cherchais », m'ont soutenu sans ménagement pour m'aider à « trouver ». Merci à ceux qui ont pu venir à ma thèse mais aussi à ceux qui n'ont pas pu, mais qui l'auraient certainement souhaité. J'ai tout spécialement une pensée pour ma femme et pour les adorables enfants que nous avons eus ensemble. L'amour que vous me témoignez tous les jours est mon plus grand réconfort et, j'en suis certain, je ne serais jamais arrivé au bout sans lui.

Table des matières

Introduction	1
1 Étude de la problématique	5
1.1 Un nouveau besoin	5
1.1.1 Notion d'utilisateur ordinaire	5
1.1.2 Utilisateur ordinaire et utilisateur averti	5
1.1.3 Interaction avec un utilisateur ordinaire	6
1.2 Notion de bon sens en CHM	6
1.2.1 Le bon sens en Intelligence Artificielle	7
1.2.2 Le bon sens en CHM	9
1.3 Le projet <i>InterViews</i>	10
1.3.1 Définition	10
1.3.2 Le langage VDL	11
1.3.3 Architecture	12
1.3.4 Objectifs de la thèse dans le projet <i>InterViews</i>	13
1.4 Modélisation et raisonnement sur les actions	14
1.4.1 Modèles de raisonnement sur les actions et le changement	14
1.4.2 Raisonnement de bon sens sur les actions	19
1.4.3 Planification, diagnostic et explications	24
1.4.4 Logiques réifiées	29
1.4.5 Raisonnement sur le fonctionnement dans les vues	31
1.5 Bilan	32
2 VDL Procédural	33
2.1 Analyse de la problématique	33
2.1.1 Le projet <i>InterViews</i>	33
2.1.2 Contraintes de VDL	34
2.1.3 Modèles de descriptions de fonctionnement	36
2.1.4 Définition de VDL procédural	43
2.1.5 Plan	44
2.2 Principe général	45
2.2.1 Partie structurelle	45
2.2.2 Notion d'observateur	47
2.3 L'observateur dynamique <i>exec</i>	50

2.3.1	Principe général	50
2.3.2	Références	52
2.3.3	Interprétation canonique	56
2.3.4	Exécution des actions	62
2.3.5	Exemples	73
2.3.6	Propriétés du langage	75
2.4	L'observateur d'interface	80
2.4.1	Principe général	80
2.4.2	Sémantique	81
2.4.3	Exemples	85
2.5	Bilan	88
3	Langage de requêtes	91
3.1	Étude du problème	91
3.1.1	Contexte de l'étude	91
3.1.2	Un corpus de questions	95
3.1.3	Classement des questions du corpus	97
3.1.4	Modèles de requêtes existants	101
3.2	Présentation du modèle de requêtes	103
3.2.1	Syntaxe	103
3.2.2	L'acte : α	104
3.2.3	Le type : τ	104
3.2.4	Le booléen ν	105
3.2.5	Le sujet : σ	105
3.2.6	L'objet : ω	106
3.2.7	La date : δ	106
3.2.8	Notions de requête multiple	107
3.3	Modélisation des questions du corpus	107
3.3.1	Requêtes d'acte $\alpha \in \{Ask, What\}$ au présent	108
3.3.2	Requêtes d'acte $\alpha = How$	113
3.3.3	Requêtes d'acte $\alpha = Why$ au présent	114
3.3.4	Commandes	116
3.3.5	Requêtes temporelles	118
3.3.6	Requêtes non traitées	120
3.3.7	Limites de notre modèle	121
3.4	Bilan	125
4	Traitement des requêtes	127
4.1	Étude de la problématique	127
4.1.1	Objectifs	127
4.1.2	Notion de requêtes de bon sens	128
4.1.3	Plan du chapitre	132
4.2	Manipulation de connaissances de bon sens	133
4.2.1	Catégorisation des CSR	133

4.2.2	Notion de <i>Ground</i>	135
4.2.3	Utilisation de <i>patterns</i>	137
4.2.4	Le <i>Ground</i> des Relations Statiques : \mathcal{G}_{rs}	141
4.2.5	Le <i>Ground</i> procédural : \mathcal{G}_p	145
4.2.6	Le <i>Ground</i> des relations procédurales : \mathcal{G}_{rp}	151
4.2.7	Bilan sur les CSR	159
4.3	Traitement des requêtes	160
4.3.1	Algorithmes de recherche d'action	161
4.3.2	Requêtes d'acte $\alpha = What$	167
4.3.3	Requêtes d'acte $\alpha = Ask$	174
4.3.4	Requêtes d'acte $\alpha = How$	182
4.3.5	Requêtes d'acte $\alpha = Why$	185
4.3.6	Requêtes d'acte $\alpha = WhatIf$	194
4.4	Bilan	198
5	Extraction de comportement à partir des flux d'interaction	203
5.1	Mémorisation de l'exécution	203
5.1.1	Analyse du problème	203
5.1.2	Chroniques en extension	207
5.1.3	Chroniques en intension	210
5.1.4	Extraction de comportement	214
5.1.5	Extraction de comportements collectifs dans un SMA	215
5.2	Utilisation dans le MRF	220
5.2.1	Requêtes temporelles : $\delta \neq \checkmark$	220
5.2.2	Requêtes d'acte $\alpha = When$	228
5.2.3	Requêtes d'acte $\alpha = Why$	230
5.3	Bilan	232
6	Application au web sémantique	235
6.1	Analyse de la problématique	235
6.1.1	Les composants actifs dans le web sémantique	235
6.1.2	Notion d'observateur en XML	237
6.2	Présentation du modèle	239
6.2.1	Partie structurelle	239
6.2.2	Partie fonctionnelle	240
6.2.3	Bilan	242
6.3	Conclusion	244
7	Implémentation de VDLp	245
7.1	Interprète VDL en Java	245
7.1.1	Présentation des classes Java	245
7.1.2	Interprétation des vues	248
7.1.3	Architecture du projet <i>InterViews</i>	249
7.2	Utilisation de code Java en VDLp	251

7.2.1	Syntaxe	251
7.2.2	Sémantique	252
7.2.3	Exemples	253
Conclusion		257

*Demander si un ordinateur peut penser revient à
demander si un sous-marin peut nager.*

(Anonyme)

Introduction

Avec le développement des outils informatiques pour le grand public, en particulier dans les services en ligne sur Internet, nous voyons apparaître de nouveaux axes de recherche dans le domaine de l'interaction homme-machine (IHM).

En effet, les utilisateurs qui interagissent avec des composants logiciels veulent pouvoir leur poser des questions en langage naturel à propos de leur activité, des opérations qu'ils effectuent et des données qu'ils manipulent. Mais les composants actuels ne peuvent pas répondre aux questions des utilisateurs sur leur fonctionnement, d'une part parce qu'ils ne sont pas encore capable d'interagir complètement en langage naturel, et d'autre part parce qu'ils ne sont pas munis de capacité d'inspection et de raisonnement sur leur propres actions.

De plus, les utilisateurs *ordinaires* ne disposent pas de connaissances spécifiques sur le fonctionnement interne du système avec lequel ils interagissent. Dans ce cadre, les questions qu'ils peuvent poser concernant les actions entreprises par le système font uniquement appel à des connaissances « de bon sens », partagées par tous les utilisateurs. Dans les modèles actuels, l'utilisateur doit disposer de connaissances spécifiques pour pouvoir construire des requêtes bien formées, qui seront traitables par le système. Ce prérequis ne peut pas convenir pour la prise en compte d'utilisateurs ordinaires en IHM.

Dans cette thèse, nous proposons une première approche pour la construction automatique de réponses à des requêtes formelles sur les actions d'un composant, modélisant des questions de bon sens posées par les utilisateurs ordinaires concernant le fonctionnement d'un composant logiciel actif. Deux problèmes majeurs se posent alors pour le traitement de ces requêtes :

1. Les systèmes actuels proposés par la communauté du raisonnement sur les actions travaillent uniquement sur des requêtes bien formées, ne prenant pas en compte l'existence de *notions de bon sens* partagées par tous les utilisateurs. En IHM, nous voulons au contraire pouvoir interpréter et répondre à des requêtes faisant intervenir des notions de bon sens.
2. Ces systèmes, fondés sur des mécanismes d'inférence en logique, travaillent uniquement à partir d'une modélisation formelle du composant par un expert. Dans ce cadre, le système ne peut raisonner que sur ce qui a été formalisé. Au contraire, pour répondre aux questions des utilisateurs ordinaires et interpréter les notions de bon sens, nous voulons pouvoir travailler directement sur le code du composant.

Le travail de recherche que nous présentons ici s'inscrit dans le projet *InterViews* du LIMSI-CNRS. Dans ce projet, nous essayons de modéliser et d'implémenter des 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. Dans cette thèse, nous ne prenons en considérations que des requêtes formelles, construites par le module de traitement automatique du langage naturel (TALN) et pouvant faire intervenir des notions de bon sens. Nous n'étudions pas les problèmes de TALN soulevé par la production de ces requêtes.

Le premier chapitre présente le cadre de l'étude, les travaux effectués dans le domaine du raisonnement en Intelligence Artificielle, sur lesquels nous pouvons nous appuyer, ainsi que les enjeux soulevés par notre problématique. Nous y présentons le projet *InterViews* et nous situons notre travail dans ce projet.

Le second chapitre présente le langage de description et le modèle d'exécution pour les composants actifs que nous utilisons dans le projet *InterViews* : VDL. Ce langage, qui intègre au sein d'une même représentation les aspects structurels d'un composant et le fonctionnement du composant, permet d'accéder *en cours d'exécution* à une description des actions et des données manipulées. Il est à la fois le langage de programmation des composants actifs et le modèle sur lequel s'appuient nos algorithmes de construction de réponse aux questions.

Le troisième chapitre présente un langage de requêtes qui permet de modéliser une classe de questions que les utilisateurs peuvent poser concernant les actions, le fonctionnement, l'activité ou le comportement d'un composant actif. Nous décrivons la méthode employée pour élaborer ce modèle ainsi que les contraintes imposées par notre problématique. Nous discutons ensuite l'expressivité de notre modèle.

Le quatrième chapitre présente les algorithmes de traitement des requêtes. Nous distinguons le cas des requêtes portant simplement sur des variables de celui, plus complexe, des requêtes portant sur le fonctionnement. Nous montrons qu'il est possible de prendre en compte des requêtes qui font intervenir des notions de bon sens, issues directement de l'interaction en langue naturelle avec l'utilisateur. Nous présentons un cadre formel qui permet de munir les composants actifs de « connaissances de bon sens », ainsi que des algorithmes de traitement de ces requêtes.

Notre étude nous a conduit à réfléchir à différentes approches possibles pour la construction *d'explications* en réponse à des questions des utilisateurs. L'approche classique utilisée dans le domaine du raisonnement sur les actions et les changements est une approche *statique* : le système a accès à une description en logique de ses actions et aux valeurs des prédicats à différents instants de l'exécution. Il utilise des mécanismes d'inférence pour construire une explication ou une réponse à la requête formelle. Le cinquième chapitre de cette thèse présente au contraire une approche *dynamique* pour le traitement des requêtes portant sur le passé du composant. Nous montrons que, en mémorisant non seulement les valeurs passées des variables mais aussi les actions effectuées et les situations dans lesquelles elles se produisent, il est possible d'extraire des régularités à partir des flux d'interaction et de dégager ainsi un comportement. Nous comparons les possibilités offertes par l'approche dynamique

et par à l'approche *statique* pour le construction d'explications et nous montrons dans quelles situations l'approche dynamique permet de découvrir de nouvelles règles pour construire des réponses aux requêtes de l'utilisateur.

L'évolution récente des technologies de l'information et de la communication a permis l'émergence d'Internet tel que nous le connaissons aujourd'hui. C'est le lieu, sans cesse en expansion, où se rencontrent les services informatiques et les utilisateurs ordinaires. L'information contenue dans le web actuel n'est pas décrite de manière structurée. Les agents, contrairement aux humains, n'ont donc pas accès à la *signification* de ce contenu. Le *web sémantique* se veut une extension du web actuel dans laquelle l'information est décrite de manière bien structurée pour palier ce problème et permettre des traitements automatiques portant sur la signification du contenu.

Les modèles actuels du web sémantique proposent des modèles pour l'information structurelle, mais il existe peu de recherche sur les services et l'utilisation de composants par des utilisateurs ordinaires. Le sixième chapitre présente une adaptation de nos travaux dans le cadre du web sémantique qui permet de résoudre plusieurs problèmes encore ouverts dans ce domaine de recherche. Nous montrons qu'il est possible de programmer des pages web actives en XML qui seraient capables de répondre à des requêtes portant non seulement sur leur structure, mais aussi sur leur fonctionnement.

Enfin, le septième chapitre présente une implémentation partielle de nos travaux afin d'illustrer sur des exemples concrets ce que notre travail, relativement théorique, pourrait permettre dans un cadre plus ingénieral. Nous avons implémenté entièrement la sémantique opérationnelle du langage VDL dans un interprète en Java, travaillant directement sur une représentation abstraite du composant. Nous donnons plusieurs exemples de composants logiciels simples implémentés dans notre modèle. Nous avons implémenté partiellement les algorithmes de construction de réponses aux requêtes formelles. Nous avons intégré cet interprète VDL et le module de traitement de requêtes avec des outils de traitement de la langue naturelle pour le projet *InterViews*, dans un modèle client-serveur HTTP. Nous illustrons ce travail sur quelques requêtes produites à partir de questions, en langage naturel, d'un utilisateur.

Chapitre 1

Étude de la problématique

1.1 Un nouveau besoin

1.1.1 Notion d'utilisateur ordinaire

L'un des principaux sujets d'étude de la communication homme-machine (CHM) est l'interaction entre un homme et une machine (IHM). L'objectif de ce domaine est de permettre à un humain d'interagir de manière naturelle avec le système pour la réalisation d'une tâche donnée. Les travaux récents dans ce domaine se sont orientés d'une part vers la conception d'interfaces graphiques (GUI) conviviales et ergonomiques [10, 128] et d'autre part vers l'utilisation du langage naturel pour réaliser des systèmes performants [4].

Avec le développement des outils informatiques pour le grand public, en particulier dans les services en ligne sur Internet, le domaine de l'IHM est confronté à un nouveau défi. Il doit permettre aux systèmes d'interagir avec des *utilisateurs ordinaires*. La prise en compte de ces utilisateurs nécessite d'apporter des fonctions d'assistance intelligente intégrées dès l'origine aux services [85].

1.1.2 Utilisateur ordinaire et utilisateur averti

La notion d'*utilisateur ordinaire* est encore nouvelle en CHM. Nous la définissons de la manière suivante :

Un utilisateur ordinaire est un utilisateur qui ne dispose d'aucune représentation du fonctionnement interne du système avec lequel il interagit, des structures de données qu'il manipule ou de ses capacités d'interaction.

Au contraire, un utilisateur averti possède une représentation (même partielle) du fonctionnement du système.

Pour illustrer ces deux notions, appuyons-nous sur un exemple simple : nous sommes presque tous des utilisateurs avertis face à une machine à café. Nous connaissons les règles de l'interaction avec la machine, les paramètres possibles en entrée

(mettre de l'argent, appuyer sur un bouton) et les résultats attendus en sortie (un café chaud), les dysfonctionnements possibles et les causes éventuelles (plus de « touillette »).

Remarquons qu'un informaticien de formation peut être un utilisateur ordinaire d'un logiciel qu'il ne connaît pas. Même un utilisateur averti devant un logiciel donné peut devenir un utilisateur ordinaire face à une nouvelle fonctionnalité (inconnue) du système. Enfin, un utilisateur ordinaire peut devenir un utilisateur averti avec le temps et l'expérience.

La CHM peut s'intéresser aussi bien aux utilisateurs ordinaires qu'aux utilisateurs avertis. Par exemple, l'amélioration d'une interface de PAO concerne des utilisateurs avertis, connaissant l'outil ou la classe d'outil étudié. Au contraire, l'étude de l'interaction en langue naturelle avec un agent conversationnel [4] se tourne plus vers les utilisateurs ordinaires.

1.1.3 Interaction avec un utilisateur ordinaire

Les systèmes actuels doivent donc être capables d'interagir avec des utilisateurs ordinaires. Or ce qui caractérise ces utilisateurs, c'est qu'ils ne connaissent pas les règles de l'interaction avec le système, en particulier en terme d'interface graphique. Dans ce contexte, en situation de demande d'aide, il ne peut que se replier sur les moyens d'interaction « naturelle » dont ils disposent que sont le geste et la parole pour *poser des questions à la machine* à propos de ses possibilités d'interaction, de son fonctionnement, de ses connaissances.

C'est la raison pour laquelle les systèmes d'IHM pour ces utilisateurs devraient être munis de capacités d'interaction en langue naturelle, en vue de répondre à des questions des utilisateurs en situation de demande d'aide.

De plus, pour désigner les éléments du système (au niveau de l'interface, du fonctionnement ou des données manipulées), l'utilisateur ordinaire *ne peut pas* utiliser les termes techniques tels qu'ils sont définis dans le système, c'est-à-dire tels que les utilisateurs avertis et les concepteurs ont l'habitude d'employer, puisqu'ils ne les connais pas. Au contraire, ils ne peuvent utiliser que le « sens commun », c'est-à-dire des notions « *de bon sens* » partagées par tous les utilisateurs.

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* ».

1.2 Notion de bon sens en CHM

Dans ce contexte, pour pouvoir interagir avec les utilisateurs ordinaires, les systèmes informatiques devraient être capable de comprendre et de manipuler ces connaissances « de bon sens ». L'un des nouveaux défis que doit relever la CHM est alors de munir les composants d'un « bon sens » que, initialement, ils n'ont pas.

1.2.1 Le bon sens en Intelligence Artificielle

La définition du « bon sens » en Intelligence Artificielle (IA) est fortement reliée aux modèles de raisonnement en logique. Elle est apparue dans les années 90 après plusieurs années de recherche dans le domaine de la représentation des connaissances autour de la formalisation en logique du raisonnement humain.

1.2.1.1 Modélisation du raisonnement

Le domaine de la *modélisation du raisonnement* a pour objectif de proposer une formalisation du raisonnement humain, c'est-à-dire de représenter sur une machine des mécanismes de raisonnement mis en œuvre par un être humain lorsqu'il essaye de mener à bien une tâche nécessitant de l'intelligence [59, 139].

Pour mener à bien cette tâche, les chercheurs en IA se sont appuyés sur les travaux initiés par les philosophes grecs sur *la logique*, qui permet de matérialiser un raisonnement dit « logique », *i.e.* construit et utilisant des règles acceptées par tous, comme ceux que l'on peut trouver dans la résolution d'un problème mathématique ou dans un essai de philosophie. Il s'agit de définir d'une part un cadre *syntactique* (comment représenter le raisonnement) et d'autre part une *sémantique* du langage (comment appliquer ce raisonnement).

L'exemple le plus célèbre est la logique des prédicats du premier ordre (LPPO) qui permet de représenter le raisonnement suivant : « si quelqu'un est un homme, alors il est mortel » par la formule :

$$\forall x.Homme(x) \rightarrow Mortel(x)$$

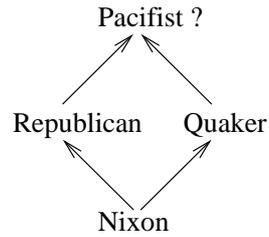
1.2.1.2 Le bon sens et la logique

Cependant, le dictionnaire francophone *Petit Robert* édition 1989 définit le « bon sens » comme :

[la] *capacité de bien juger, sans passion, en présence de problèmes qui ne peuvent être résolus par un raisonnement scientifique*

Dans ce contexte, les mécanismes de raisonnement logique utilisés de manière classique en IA sont condamnés à ne pas pouvoir effectuer de raisonnement de bon sens.

En effet, les logiciens ont rapidement mis en évidence l'un des problèmes majeurs de la logique : un modèle logique peut être inconsistant, c'est-à-dire qu'il peut permettre de déduire deux formules contradictoires. Nous pouvons illustrer ce problème sur l'exemple célèbre du « diamant de Nixon » : les républicains sont (généralement) en faveur de la guerre ; au contraire, les *quakers* sont pacifistes. Nixon étant à la fois républicain et *quaker*, la logique classique ne peut pas déterminer si Nixon est pour la paix :



Mais là où la logique entre en contradiction avec le bon sens humain, c'est que l'ajout de la proposition $Pacifiste(Nixon)$ dans les hypothèses du problème ne permet pas de rendre le monde consistant : la *sémantique* du langage choisi (la logique des prédicats) ne permet pas d'aboutir à un *modèle*, c'est à dire une interprétation¹ pour laquelle toutes les formules sont évaluées à vraies.

Pourtant, le *bon sens* nous dit que, puisqu'on sait que Nixon est pacifiste, le système ne devrait pas se poser de question et être capable de déduire immédiatement la réponse, sans aboutir à une contradiction. *La logique ne permet pas de formaliser ce raisonnement.*

1.2.1.3 Les logiques non-monotones

Ce problème de « raisonnement non conforme au bon sens » provient de la monotonie de la LPPO : si E et F sont deux ensembles de propositions, E inclus dans F , alors l'ensemble des théorèmes (*i.e.* des formules vraies) de F est inclus dans celui de E . Dit de manière moins formelle, plus on a de formules, plus on peut en déduire de nouvelles et donc plus on a des chances de prouver à la fois une chose et son contraire. Le fait d'ajouter une formule ne permet pas de rendre un modèle consistant. Au contraire, il augmente le « risque » d'obtenir un modèle inconsistant.

C'est pour palier ce problème que les chercheurs ont proposé au début des années 80 différentes logiques dites « non monotones » dont l'objectif est de permettre de *formaliser le raisonnement de bon sens* (*common-sense* en anglais [1]) comme celui qui est effectué par un humain en présence du diamant de Nixon ou, plus généralement, en face d'énoncés de la forme « *Les A sont B* » [136] :

- Les autruches sont des oiseaux ;
- Les oiseaux volent ;
- Les autruches ne volent pas.

La modélisation du raisonnement de bon sens en logique n'est pas un problème au centre de notre sujet. C'est la raison pour laquelle nous n'allons pas présenter ici les différents formalismes qui ont été proposés. Cependant, nous pouvons en citer deux qui se sont imposés comme standards dans le domaine.

¹Une interprétation au sens de Herbrand [139] peut-être définie comme étant l'assignation d'une valeur de vérité (vrai ou faux) pour à chaque atome du monde considéré.

La logique des défauts

La logique des défauts de Reiter [109] propose d'utiliser des *défauts*, c'est-à-dire des formules de la forme : $pre : jus \rightarrow csq$. L'interprétation d'un défaut est la suivante : la conséquence csq est déduite si et seulement si :

1. le prémice pre est vrai ;
2. la justification jus est cohérente avec les hypothèses \mathcal{H} du problème, c'est-à-dire que $\{jus\} \cup \mathcal{H}$ ne conduit pas à un modèle inconsistant.

Par exemple, pour modéliser que « les oiseaux volent, en règle générale », nous utiliserons le défaut :

$$\forall x.oiseau(x) : vole(x) \rightarrow vole(x)$$

C'est-à-dire que tout oiseau vole si ce n'est pas in cohérent de dire qu'il vole. Et pour exprimer que les autruches sont des oiseaux mais ne volent pas, nous écrirons simplement :

$$\forall x.autruche(x) \rightarrow (oiseau(x) \wedge (\neg vole(x)))$$

La circonscription

La *circonscription* est une technique introduite par McCarthy [88, 87] pour modéliser un raisonnement non monotone. L'idée principale est d'utiliser un prédicat ab (*abnormal*) pour exprimer qu'un individu a un comportement « anormal » par rapport à une propriété donnée. Par exemple, pour modéliser que « les oiseaux volent, en règle générale », nous utiliserons la formule :

$$\forall x.oiseau(x) \wedge \neg ab(p_1, x) \rightarrow vole(x)$$

C'est-à-dire que tout oiseau qui n'est pas une exception pour la propriété p_1 vole. Et pour exprimer que les autruches sont des oiseaux mais ne volent pas, nous écrirons :

$$\forall x.autruche(x) \rightarrow (oiseau(x) \wedge (\neg vole(x)) \wedge ab(p_1, x))$$

1.2.2 Le bon sens en CHM

La modélisation du raisonnement de bon sens consiste à définir une syntaxe et une sémantique qui permettent de mener à bien de manière automatique des raisonnements que la logique classique ne permet pas. A ce titre, elle fournit des outils qui peuvent être utilisés en CHM pour représenter les connaissances acquises et révisées lors d'un dialogue avec un utilisateur ordinaire, comme dans l'exemple « les oiseaux volent », et effectuer des inférences pour déduire les propriétés du monde conformes au bon sens.

Mais dans le cadre de l'interaction entre un utilisateur ordinaire et une machine, le problème pour le système n'est pas tant de raisonner suivant le bon sens que d'être capable de *modéliser des connaissances de bon sens*, c'est-à-dire communes à

tous les utilisateurs, qui ne sont jamais exprimées explicitement mais qui vont être utilisées par l'utilisateur ordinaire pour désigner les éléments du système avec lequel il interagit.

Définition

Dans le cadre d'une interaction entre un utilisateur ordinaire et une machine, nous pouvons définir le bon sens comme :

l'ensemble des connaissances sur lesquelles l'utilisateur ordinaire peut s'appuyer pour interagir avec un système.

Un système de CHM en interaction avec un utilisateur ordinaire doit pouvoir manipuler ces connaissances et les relier aux éléments internes du programme : données structurées, actions possibles, *etc.* Ce sont donc les utilisateurs ordinaires qui définissent ce qu'est le bon sens pour la CHM.

Les recherches actuelles en IA autour du raisonnement de bon sens ne permettent pas de manipuler des *connaissances de bon sens* au sens où l'entend la CHM. Le système ne peut en effet raisonner que sur les prédicats qui sont définis dans le monde considéré. Notre objectif est au contraire de définir :

- Un ensemble de connaissances qui ne sont pas spécifiques à un monde opératoire donné, mais qu'il faut interpréter dans le contexte de ce monde opératoire pour pouvoir interagir avec l'*utilisateur ordinaire* ;
- Les mécanismes d'interprétation que l'on peut mettre en œuvre pour manipuler ces connaissances de bon sens.

1.3 Le projet *InterViews*

Le travail de recherche que nous présentons ici s'inscrit dans le projet *InterViews* du LIMSI-CNRS, initié par Jean-Paul Sansonnet en 1999 [120]. Dans cette section, nous présentons le principe général de ce projet et la place que notre travail y trouve.

1.3.1 Définition

1.3.1.1 Objectifs

Le projet *InterViews* a pour objectif de définir et d'implémenter des *agents dialogiques*, c'est-à-dire des composants logiciels munis de capacité de raisonnement et capables d'interagir avec les utilisateurs ordinaires en langage naturel et avec d'autres composants en langage formel (comme KIF [47]).

Dans ce cadre, nous nous intéressons plus particulièrement à l'étude de *composants actifs* englobant un petit monde opératoire, comme les agents assistants d'interface [108, 81] ou les médiateurs [56, 7], qui interagissent avec un humain ou un agent pour la réalisation d'une tâche. Notre objectif est de définir des composants actifs capables de se représenter et de raisonner non seulement sur les données

qu'ils manipulent mais aussi sur leur fonctionnement, pour servir de support à un mécanisme explicatif, en réponse à des interrogations de l'utilisateur.

Considérons par exemple un opérateur humain en interaction avec un agent autonome dans le web sémantique. L'utilisateur peut s'interroger sur le bien fondé des actions entreprises par l'agent dans la réalisation de la tâche qui lui a été confiée. Il est alors nécessaire de pouvoir répondre à des questions sur ces actions.

Dans le projet *InterViews*, ces composants sont *intégrés* dans des pages web et sont appelés des *vues*. L'objectif premier du projet est de proposer un modèle dynamique de représentations sémantiques pour les vues. Le deuxième objectif est d'étudier la problématique de l'interaction en langue naturelle entre un utilisateur ordinaire en situation de demande d'aide et une vue munie de capacités de raisonnement.

1.3.1.2 Capacité d'introspection

Dans le monde de cube de Winograd [149], le système n'est pas muni de capacité de raisonnement sur ses propres actions. En effet, page 10, l'utilisateur pose la question « Peut-on poser une pyramide sur une pyramide ? » (*can a pyramid support a pyramid ?*) et le système répond « I DON'T KNOW » parce que, selon Winograd : « ... il n'y a pas de moyen simple de le dire sans examiner le programme interne du robot » (*there is no easy way to tell without being able to examine the robot's programs*).

Dans le cadre d'une interaction avec un utilisateur ordinaire en situation de demande d'aide, c'est justement à ce genre de questions – qui plus est formulées en utilisant des notions de bon sens – que nous devons être capables de répondre :

- Est-ce que je peux faire cela ?
- Comment faire pour faire ceci ?
- Pourquoi ceci est-il comme cela ?

Il est donc nécessaire de munir les composants de capacité d'introspection sur leurs propres données et sur leur fonctionnement.

1.3.2 Le langage VDL

Le modèle dynamique de représentations sémantiques des vues doit donc permettre de définir à tout instant et pour chaque vue :

- Les connaissances sur le monde opératoire qu'elle manipule. Cela comprend :
 - Une partie statique, *i.e.* l'ensemble des propriétés du monde vraies à un instant donné (par exemple : *la pomme est sur la table*), permettant de décrire des structures de données riches.

La vue doit en effet pouvoir raisonner sur sa structure statique pour répondre à des questions comme : « *Est-ce que la pomme est sur la table ?* ».

- Une partie dynamique, *i.e.* les changements possibles dans le monde. Par exemple : *il est possible de prendre la pomme.*

La vue doit en effet pouvoir raisonner sur ses fonctionnalités dynamiques pour répondre à des questions comme : « *Est-ce qu'il est possible de prendre la pomme ?* ».

- Son exécution, c'est-à-dire la manière dont les connaissances qu'elle manipule évoluent au cours du temps. Par exemple, *au début, la pomme est sur la table, puis l'utilisateur l'a prise et l'a mangée. Maintenant, il n'y a plus de pomme dans le monde considéré* ;

La vue doit en effet pouvoir raisonner sur son exécution pour répondre à des questions comme : « *Est-ce que j'ai déjà mangé la pomme ?* »

- Ses fonctionnalités dialogiques, c'est-à-dire ce qui permet de définir la classe des questions que l'utilisateur peut lui poser et les réponses que la vue peut donner.

C'est pourquoi il faut que :

1. Les composants soient programmés dans un langage qui intègre au sein d'une même représentation la description de la partie structurelle (les données manipulées dans le monde opératoire, modélisées dans une structure riche) et la partie procédurale et fonctionnelle (les actions et les fonctions de manipulation) ;
2. Ce langage soit aussi celui dans lequel se déroule son exécution, pour servir de support aux mécanismes de raisonnement sur les connaissances structurelles et fonctionnelles ;
3. Ce langage permette de définir pour chaque vue des informations de traitement automatique du langage naturel (TALN) spécifiques à la vue, utilisées pour l'interaction en langage naturel avec l'utilisateur.

C'est l'ensemble de ces contraintes qui nous a amenés, dans le projet *InterViews*, à décrire les vues dans un langage spécifique que nous appelons VDL pour *View Design Language* et qui intègre tous ces éléments. Nous appelons alors *vue* le composant décrit dans ce formalisme, qui comprend un corps (la structure statique et dynamique, en VDL), une exécution (l'état courant de la vue et la manière dont elle évolue) et une partie dialogique (décrivant ses capacités d'interaction en langue naturelle).

Une vue est donc caractérisée par un terme VDL contenant des éléments structurels (qui changent au cours de l'exécution) et des éléments procéduraux de même nature. Le langage VDL est au coeur du projet *InterViews* puisque c'est lui qui permet de définir les vues. C'est aussi le langage utilisé pour l'interaction *entre composants actifs*, pour leur permettre d'échanger des informations sur leur structure, leur fonctionnement ou leurs fonctionnalités dialogiques.

1.3.3 Architecture

Dans le projet *InterViews*, notre objectif est de munir les composants de capacités de raisonnement pour servir de support à un mécanisme explicatif. L'analyse du

problème nous a conduit à proposer le schéma de principe présenté figure 1.1 comme architecture pour le traitement des questions de l'utilisateur :

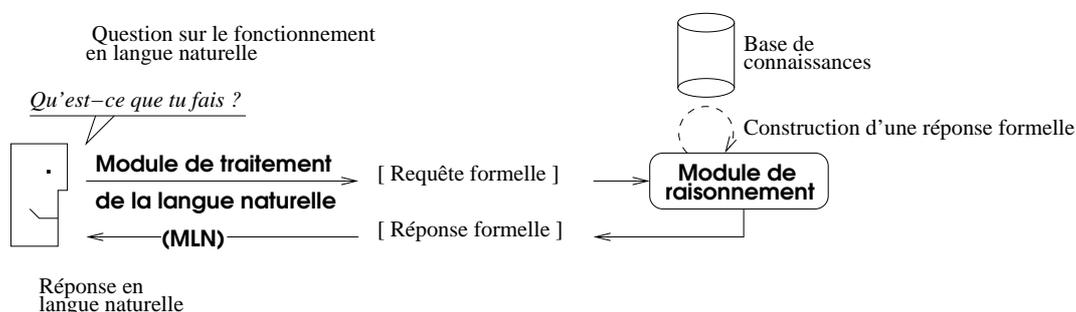


FIG. 1.1 – Traitement des questions dans le projet *InterViews*.

- Les questions de l'utilisateur sont traduites en requêtes formelles par un module de traitement du langage naturel (MLN) à l'aide d'outils de TALN qui s'appuient sur la partie structurale de la description en VDL du composant et sur les informations relatives au TALN propres au composant, comme cela se fait dans les systèmes conversationnels classiques comme [5] ;
- Ces requêtes sont ensuite traitées par un module de raisonnement au niveau formel pour produire une réponse, effectuer une action, modifier la base de connaissances, *etc.* La réponse construite par ce module est elle-même une requête formelle, qui peut-être utilisée pour produire une réponse en langue naturelle.

Nous parlerons de *question* pour une interaction en langue naturelle² et de *requête* pour une interaction formelle, issue d'une question humaine ou d'un autre composant actif en réseau (par exemple sur le web).

Le travail effectué dans cette thèse se situe uniquement au niveau du module de raisonnement formel. Nous n'aborderons pas les problèmes spécifiques au TALN, traités par le MLN. Au contraire, nous considérons que le MLN construit des requêtes bien formées.³

1.3.4 Objectifs de la thèse dans le projet *InterViews*

Dans notre étude, nous nous intéressons plus particulièrement aux aspects liés au *fonctionnement* des composants. L'objectif de la thèse est de permettre de modéliser et d'implémenter des 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.

²Même, comme nous le verrons au chapitre 3, lorsqu'il s'agit de commandes.

³Le modèle de requête utilisé et la définition d'une requête bien formée sont donnés au chapitre 3.

Dans ce cadre, nous devons définir :

- Les éléments procéduraux de VDL, c'est-à-dire les termes du langage qui permettent de décrire le comportement dynamique des vues, *i.e.* les actions possibles sur le monde opératoire manipulé ;
- La sémantique opérationnelle des vues, c'est-à-dire la fonction d'exécution des composants qui modifie le monde opératoire en fonction des actions décrites dans la vue ;
- Un mécanisme de mémorisation de l'exécution passée des composants pour pouvoir répondre à des questions temporelles ;
- Un modèle de requêtes sur le fonctionnement et les actions, utilisé par le MLN pour modéliser les questions des utilisateurs ordinaires ;
- Des algorithmes de construction de réponses à ces requêtes, fondés sur un modèle de raisonnement sur le fonctionnement et utilisant la mémorisation du passé du composant.

Notre travail se situe donc essentiellement au niveau de l'interaction entre un utilisateur et un composant. Toutefois, le modèle de requêtes sur les actions que nous proposons ici est aussi utilisé pour l'échange de connaissances d'ordre procédurales entre des agents. De plus, nous montrerons au chapitre 5 que la mémorisation de l'exécution passée du composant peut être utilisée non seulement pour répondre à des requêtes portant sur le passé, mais aussi pour extraire des comportements collectifs dans les systèmes multi-agents (SMA).

L'objectif central de cette thèse est donc de définir un Module de Raisonnement sur le Fonctionnement (MRF) pour produire des *explications* sur le fonctionnement du composant actif, en réponse à des questions de l'utilisateur.

1.4 Modélisation et raisonnement sur les actions

La problématique du raisonnement sur les actions (*reasoning about actions and changes* en anglais) est relativement ancienne dans la communauté IA. Nous pouvons considérer qu'elle est apparue en 1969 avec l'article fondateur de McCarthy et Hayes [91].

Pour concevoir le MRF, dont l'objectif est de raisonner sur la partie opératoire (les actions) et dynamique (l'exécution) de la vue, nous avons logiquement été amené à étudier les modèles de raisonnement sur les actions qui ont été proposés en réponse aux problématiques soulevées dans ce domaine.

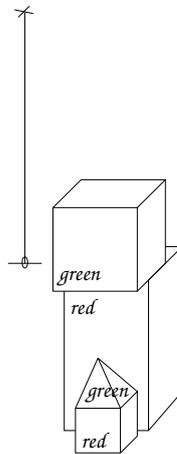
1.4.1 Modèles de raisonnement sur les actions et le changement

L'une des faiblesses de la LPPO est qu'elle ne permet pas directement de manipuler la notion d'action : les prédicats représentent des faits (ce qui est vrai dans le monde considéré) mais il n'est pas possible de modéliser des changements de valeur de vérité de ces faits en fonction de l'exécution d'actions (par exemple pour représenter

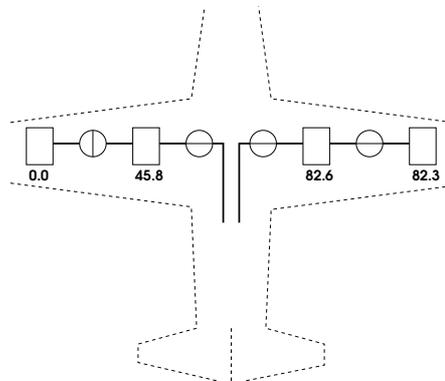
l'exécution d'un programme) pour raisonner dessus. Autrement dit, la modélisation du raisonnement en LPPO permet aux ordinateurs de raisonner sur un monde figé, mais elle ne leur permet pas de faire de raisonner sur ce qu'ils font (ou pourraient faire) pour mener à bien une tâche.

En particulier, deux problèmes majeurs nécessitent de pouvoir raisonner sur les actions et les changements du monde :

- La *planification*, c'est-à-dire la construction automatique par une machine de la séquence d'actions (le *plan*) qu'il faut effectuer à partir d'un état donné du monde (l'état initial) pour atteindre un autre état donné (l'état *but*) [48, 114]. Par exemple, dans le monde de cubes de Winograd [149], pour traiter le premier ordre : « *prend un gros pavé rouge* » (page 8, exemple 1), le système doit déplacer plusieurs objets. Dans ce cadre, il doit construire un plan, c'est-à-dire une séquence d'actions qui lui permet de libérer prendre le *gros pavé rouge* en retirant le cube vert qui est posé dessus.



- Le *diagnostic*, c'est-à-dire la mise en évidence d'un ensemble de dysfonctionnements responsables d'un état de fait anormal dans le système. Par exemple, dans un avion quadri-réacteur, il est important de pouvoir déterminer que c'est une fuite dans le moteur 2 qui a déclenché l'arrêt du moteur 1, parce qu'une réponse appropriée serait de couper aussi le moteur 2 et de fermer l'arrivée de fuel dans toute l'aile.



Les chercheurs en IA travaillant sur la modélisation du raisonnement ont donc proposé des formalismes pour représenter en logique la notion d'action et le lien entre les *faits* et les *actions*, avec pour objectif a terme de pouvoir raisonner sur les actions pour effectuer de la planification et du diagnostic. Ces travaux sont regroupé dans le domaine de la modélisation du raisonnement sur les actions et le changement (« *Reasoning about Actions and Changes* » en anglais). Deux approches concurrentes ont émergé rapidement : celle du calcul des situations [91, 112] et celle de la logique dynamique [104, 58], avec tout un spectre de formalismes à mi-chemin entre les deux.

1.4.1.1 Le calcul des situations

Le calcul des situations, dans sa version initiale [91], est une extension de la LPPO dans laquelle les valeurs de vérité des prédicats sont exprimées par rapport à des situations. L'élément de base du calcul des situations est la *situation*. Une situation est un état du monde, décrit de manière incomplète, et pouvant même être hypothétique.

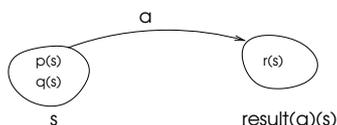
La sémantique du calcul des situation est fondée sur la notion de *fluent* (appelé parfois *quantité* en français). Un *fluent* est une fonction de l'ensemble des situations (*Sit*) dans :

- L'ensemble des booléens : nous parlons alors de *fluent propositionnel* ; ils sont utilisés pour modéliser les prédicats de la LPPO ;
- L'ensemble des situations : nous parlons alors de *fluent situationnel* ;

Les atomes (*ground terms* en anglais) du calcul des situations sont de la forme $f(s)$ où f est un *fluent* propositionnel et s une situation. Ils permettent de modéliser le fait que le prédicat représenté par le *fluent* propositionnel f est vrai dans la situation s . Par exemple, pour modéliser le fait que *Socrate* est vivant dans la situation s_0 , nous pouvons écrire :

$$vivant(Socrate)(s_0)$$

Pour décrire les transitions entre les situations, nous utilisons l'opérateur *result*. Si a est une action, l'expression $result(a)$ est un *fluent situationnel* et pour toute situation s , $result(a)(s)$ désigne la situation à laquelle on aboutit après exécution de l'action a à partir de la situation s .



Exemple (prise d'un objet)

Considérons par exemple le raisonnement suivant :

« si un objet est léger et sur un support proche de moi, alors je peux le prendre et le résultat est qu'il se retrouve dans ma main (et non plus sur le support) »

En calcul des situations, nous écrivons :

$$\forall x. \forall y. \forall s. \left[\begin{array}{l} leger(x)(s) \wedge sur(x, y)(s) \wedge proche(y)(s) \\ \rightarrow \left(\begin{array}{l} sur(x, main)(result(prendre(x), s)) \\ \wedge \neg sur(x, y)(result(prendre(x), s)) \end{array} \right) \end{array} \right]$$

Remarque

En calcul des situations comme dans tous les autres formalismes de raisonnement sur les actions et les changements, nous dirons que la formule :

$$(leger(x) \wedge sur(x, y)(s) \wedge proche(y))$$

décrit les *préconditions* de l'action *prendre(x)* et que :

$$(sur(x, main) \wedge \neg sur(x, y))$$

donne ses *postconditions* (ou ses conséquences).

Un formalisme fondateur

Le calcul des situation est le formalisme fondateur du raisonnement sur les actions. Il a permis de matérialiser de nombreux problèmes de raisonnement que nous présentons dans la section suivante.

Pinto et Reiter [101] ont montré que le calcul des situations était au moins aussi expressif que n'importe quelle logique temporelle [2, 73, 92]. En particulier, ils ont montré que les logiques modales temporelles [105] pouvaient être modélisée en calcul des situations, même lorsqu'elles font intervenir des actions concurrentes.

La version du calcul des situations qui est la plus utilisée aujourd'hui est la version de Reiter [112, 111] qui fait suite à l'analyse de Shanahan [131] montrant que, pour la construction d'explications, il est préférable de décrire les liens entre les situations par des relations d'ordre temporel ($s_0 < s_1$) que par l'opérateur *result*, qui ne permet pas de matérialiser la notion de causalité [99, 64].

Cette version permet de définir des occurrences d'action dans le futur et traiter les actions concurrentes [100]. Elle a été implémentée dans le système GOLOG [79] qui permet de modéliser le fonctionnement d'un robot et de le munir ainsi de capacité de raisonnement sur ses actions, essentiellement pour construire un plan d'actions pour la réalisation d'une tâche donnée (décrite par un état but).

1.4.1.2 La logique dynamique

La logique dynamique (PDL) est une logique modale [78], dans laquelle les opérateurs modaux sont les actions et des combinaisons d'actions. Comme le calcul

des situations, elle repose sur la notion de *situation* : une situation est l'assignation d'une valeur de vérité pour *tous*⁴ les prédicats du problème considéré.

On appelle *monde* un ensemble de situations reliées entre elles par des actions. Un monde peut être visualisé par un graphe dans lequel :

- les nœuds correspondent à des situations
- les arcs de la forme $s_1 \xrightarrow{a} s_2$ signifient que l'on peut passer de la situation s_1 à la situation s_2 en effectuant l'action a .

Par exemple, la figure 1.2 représente un monde dans lequel nous avons 4 situations et 2 actions. Dans la situation s_1 , la proposition r est vraie. Effectuer l'action a dans

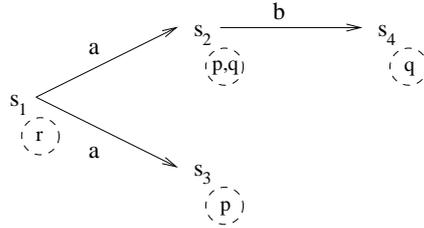


FIG. 1.2 – Un exemple simple en PDL

cette situation conduit à se retrouver soit dans la situation s_2 , soit dans la situation s_3 . Au contraire, dans la situation s_3 , aucune action n'est possible.

Les opérateurs modaux de la logique dynamique sont les actions et les combinaisons d'actions :

- $a; b$ exprime la composition séquentielle de deux actions, c'est-à-dire l'action obtenue en effectuant a puis en effectuant b .
- $a|b$ exprime le choix non déterministe entre deux actions, c'est-à-dire l'action obtenue en effectuant a ou en effectuant b .
- a^* exprime l'itération d'une action un nombre fini de fois, c'est-à-dire l'action obtenue en effectuant a puis a , puis encore a , etc. n fois de suite, avec $n \geq 0$.
- $\phi?$ exprime le test d'une propriété ϕ , c'est-à-dire l'action de tester la valeur de vérité de ϕ . Remarquons qu'il s'agit d'une action construite à partir d'une proposition logique et non pas, comme dans les autres cas, à partir d'autres actions.

Exemple

Si nous reprenons l'exemple précédent (prise d'un objet), nous pouvons représenter le changement dans le monde par la formule PDL suivante :

$$\langle (leger(x) \wedge sur(x, y) \wedge proche(y))?; prendre(x) \rangle (sur(x, main) \wedge \neg sur(x, y))$$

⁴Les situations sont complètement décrites, contrairement au calcul des situations.

La logique dynamique et le raisonnement sur les actions

Castilho [17] propose d'utiliser la logique PDL pour raisonner sur les actions. Il définit un langage multi-modal, \mathcal{LAP} , muni des opérateurs modaux $[a]$ pour toute action atomique a et d'un opérateur de nécessité L , et reprend les opérateurs de test ($p?$), de séquence ($a; b$) et de choix non déterministe ($a \cup b$).

Les actions en \mathcal{LAP} sont décrites sous la forme de lois d'action gardées : $L(G \rightarrow [a]p)$. La sémantique du formalisme permet de déterminer si une proposition p est conséquence logique de la « base de connaissances » initiale (*i.e.* la donnée de prédicats initiaux). Castilho montre comment cela peut être utilisé pour la planification et la construction d'explications [18].

1.4.2 Raisonnement de bon sens sur les actions

Dès l'apparition du calcul des situations en 1969, les chercheurs du domaine ont mis en évidence la difficulté de modéliser le raisonnement de bon sens lorsque la notion d'action intervenait. Comme pour la LPPO, la sémantique des premiers formalismes de raisonnement sur les actions conduit à des modèles contraires au bon sens. Trois principaux problèmes ont été identifiés.

1.4.2.1 Le problème de la qualification

Reprenons l'exemple « prise d'un objet ». Si l'objet est collé sur son support, le bon sens nous dit qu'on ne peut pas prendre l'objet. En logique, il faut l'explicitement en ajoutant la précondition $\neg \text{collé}(x)$ pour l'action $\text{prendre}(x)$. Mais dans ce cas, dans chaque état du monde et pour chaque objet prenable, il faut dire explicitement que l'objet n'est pas collé : $\neg \text{collé}(\text{objet}_1)(\text{situation}_1)$. Sinon, le système sera incapable de déduire que l'action prendre est possible pour l'objet considéré dans la situation courante.

Ce problème est appelé problème de la *qualification*. Il provient de l'hypothèse de *monde clos* faite en logique : tout ce qui est dit est vrai et tout ce qui n'est pas dit est faux⁵. Par conséquent, s'il fallait donner toutes les préconditions d'une action, y compris celles dictées par le bon sens, il faudrait aussi donner explicitement la valeur de vérité de toutes ces préconditions dans chaque situation, ce qui est impossible.

La solution proposée par les chercheurs du domaine est d'utiliser la circonscription [89] et de considérer (à l'aide du prédicat ab) que le fait d'être collé à son support est une situation *anormale* par rapport à l'action de prendre :

$$ab(\text{prendre}, \text{objet}_2)$$

De cette façon, il n'est pas nécessaire de donner les préconditions « de bon sens » d'une action (l'objet n'est pas collé, il n'est pas liquide, il n'est pas vissé, *etc*) : elles

⁵Cette hypothèse est fondamentale pour pouvoir effectuer des déductions en logique. Le raisonnement en monde ouvert est un problème beaucoup plus complexe [70].

sont regroupées sous cette notion de *normalité* pour l'action considérée. La seule précondition ajoutée à l'action *prendre* est :

$$\neg ab(\text{prendre}, x)$$

1.4.2.2 Le problème du décors

Considérons une fois encore sur l'exemple « prise d'un objet ». Conformément à ce que nous dicte le bon sens, nous avons uniquement décrit les *changements* effectués par l'action *prendre*, *i.e.* les nouvelles valeurs de vérité des propositions concernés par l'action : $(sur(x, main) \wedge \neg sur(x, y))$. Nous n'avons pas explicité les « non-changements », *i.e.* le fait que les valeurs de vérité des autres propositions ne changeaient pas. Cette *hypothèse d'inertie* est conforme au bon sens : a priori, une action ne change pas la valeur de vérité d'une propositions.

Mais en logique, il faut donner explicitement tous les non-changements pour que le système puisse les déduire dans la nouvelle situation⁶. Le *problème du décors*, c'est justement qu'il est impossible de donner comme postconditions d'une action toutes les propositions dont la valeur de vérité est inchangée (le « décors » de l'action). Au contraire, nous voulons que les valeurs de ces propositions soit déduits automatiquement par le système (en respectant la sémantique du langage).

Une première solution

McCarthy propose en 1986 [89] de *formaliser l'hypothèse d'inertie* dans le calcul des situations en utilisant le prédicat *ab* pour dire que tout fluent d'état *f* qui n'est pas anormal par rapport à une action *a* reste inchangé quand cette action est effectuée :

$$\neg ab(a, f)(s) \rightarrow [f(s) \leftrightarrow f(result(a, s))]$$

Le problème qui subsiste est de savoir comment modifier la sémantique du formalisme logique utilisé pour déduire automatiquement tous les « non changements » du monde à partir de cette hypothèse d'inertie. La solution de McCarthy est de minimiser le changement, c'est-à-dire considérer les modèles dans lesquels les valeurs des fluents ont changé le moins possible. McDermott montre dès l'année suivante [57], avec le *Yale Shooting Problem* (YSP), que cette solution au problème du décors échoue, c'est-à-dire que le résultat est contraire au bon sens, comme nous allons le montrer.

Le Yale Shooting Problem

Dans le YSP, trois actions sont possibles : *tirer* (avec un revolver), *charger* (le revolver) et *attendre*, pour modifier deux prédicats : *chargé* (le revolver est chargé) et

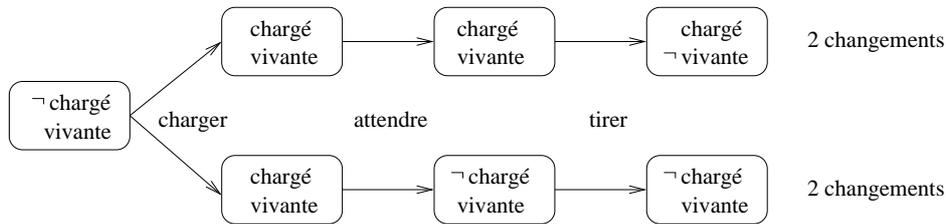
⁶Sans cela, et à cause de l'hypothèse de monde clos, les propriétés « non reconduites » deviendraient fausses.

vivante (la dinde que je vise est vivante). Le monde est décrit en calcul des situations de la manière suivante :

$$\forall s. \text{chargé}(\text{result}(\text{charger}, s)) \\ \forall s. \text{charge}(s) \rightarrow \neg \text{vivante}(\text{result}(\text{tirer}, s))$$

Soit s_0 la situation dans laquelle la dinde est vivante et le revolver non chargé. Soit $s_1 = \text{result}(\text{charger}, s_0)$, $s_2 = \text{result}(\text{attendre}, s_1)$ et $s_3 = \text{result}(\text{tirer}, s_2)$. Alors dans la situation s_3 , on s'attend à ce que la dinde soit morte.

Mais d'après la sémantique du calcul des situations, si l'on cherche à minimiser le changement entre s_1 et s_3 , deux modèles sont possibles :



2 modèles possibles

En effet, rien (sauf le bon sens) ne dit que le revolver ne peut pas se décharger tout seul. Le second modèle, contraire au bon sens, est cohérent avec la sémantique du formalisme et c'est bien un modèle minimal du point de vue du changement.

1.4.2.3 Les approches systématiques

Les travaux sur le problème du décors ont conduit à l'émergence d'une multitude de formalismes, chacun adapté pour traiter un cas particulier, mais aucun ne permettant de proposer une solution globale au problème du décors [19, 110].

C'est avec cet objectif de prendre en considération le problème dans son entier que sont apparues, au début des années 90, les *approches systématiques* [16]. Ces formalismes sont caractérisés par :

- Une sémantique simple, permettant de trouver l'ensemble des modèles à partir de la seule description des actions et des observations ;
- L'utilisation du principe d'inertie : les choses ne changent pas sauf quand c'est dit ;
- L'hypothèse de connaissance complète des actions (pas d'effet-surprise) ;
- L'hypothèse que toutes les actions possibles sont connues de l'agent.

Feature & Fluents

Feature & Fluents est la première des approches systématiques. Elle fut introduite par Sandewall dans [119]. Dans ce formalisme, les fluents ne sont pas limités à la logique bi-valuée et les actions sont munies d'une notion de durée.

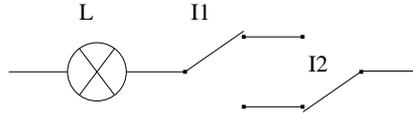
La solution au problème du décors de Sandewall repose sur la sémantique d'ego-monde, c'est-à-dire que la construction des modèles se fait suivant un jeu d'interaction entre l'agent et le monde extérieur, dans laquelle l'exécution des actions est simulée. Dans *Feature & Fluents*, si I est une interprétation au sens de Herbrand (*i.e.* une assignation de valeur de vérité pour chaque atome du monde) et a une action, alors :

- $Infl(a, I)$ est l'ensemble des atomes qui sont susceptible de changer de valeur de vérité si l'action a est exécutée.
- $Trajs(a, I)$ est l'ensemble des trajectoires. Une *trajectoire* est un ensemble ordonné de littéraux L_i tels que $|L_i| \in Infl(a, I)$, c'est-à-dire une liste des conséquences possibles de l'action a . $Trajs(a, I)$ est donc l'ensemble des « séries de conséquences possibles » d'une action a dans une interprétation I .

Pour construire les modèles, la sémantique repose alors sur un principe simple : l'agent « effectue » des actions et le monde « réagit » suivant la description des effets de $Infl$ et $Trajs$. Sandewall associe alors à chaque type de problème du décors une classe de trajectoire particulière. Il met ainsi en évidence les différents approches possibles pour traiter le problème du décors dans son ensemble, chacune associée à une définition précise de $Infl$ et $Trajs$.

1.4.2.4 Le problème des ramifications

Depuis *Feature & Fluents*, le problème du décors est considéré comme résolu. Il apparaît alors un nouveau problème de raisonnement de bon sens pour les actions ayant des effets indirects. Considérons par exemple un interrupteur en va et vient :



Deux actions (a_1 et a_2) sont possibles pour basculer les interrupteurs (respectivement I_1 et I_2). En utilisant la syntaxe du calcul des situations, nous pouvons décrire ce monde de la manière suivante :

$$\begin{aligned} \forall s. I_i(s) &\rightarrow \neg I_i(result(a_i, s)) \\ \forall s. \neg I_i(s) &\rightarrow I_i(result(a_i, s)) \end{aligned}$$

De plus, dans toute situation, la lampe L est allumée si et seulement si les deux interrupteurs sont dans la même position. En se fiant à notre bon sens, nous voudrions décrire ce fait comme une loi du monde, par la formule :

$$(I_1 \leftrightarrow I_2) \leftrightarrow L \tag{1.1}$$

Malheureusement, la sémantique de la logique conduit encore une fois à un résultat contraire au bon sens. Considérons par exemple la situation initiale $s_0 =$

$\{\neg I_1, I_2, \neg L\}$ et la situation $s_1 = result(a_1, s_0)$. Deux modèles minimaux (du point de vue du changement) sont alors possibles : $s_1 = \{I_1, I_2, L\}$ (c'est le modèle que nous dicte le bon sens) et $s_2 = \{I_1, \neg I_2, \neg L\}$ dans lequel c'est I_2 qui a basculé lorsque I_1 a été modifié, et non la lampe. En effet, la loi 1.1 est équivalente, du point de vue de la sémantique de la logique, à la loi :

$$I_1 \leftrightarrow (I_2 \leftrightarrow L)$$

et par conséquent rien n'interdit que ce soit I_2 qui s'aligne sur la lampe (au lieu du contraire) pour respecter la nouvelle valeur de vérité de I_1 . En fait, ce que dit le bon sens, c'est que le changement d'état de la lampe L est un *effet indirect* des actions a_i . Pour contourner ce problème, il faudrait donc le décrire comme tel, *i.e.* comme une postcondition des actions :

$$\forall s. [I_1(s) \leftrightarrow I_2(s)] \rightarrow \neg L(result(a_1, s))$$

...

C'est le problème des *ramifications*, énoncé par Finger dès 1987 [42] : il est impossible de décrire explicitement tous les effets indirects des actions. Au contraire, nous voudrions que ces effets soient inférés automatiquement par le système, conformément au bon sens.

La logique d'action \mathcal{A} et ses dérivés

Le problème des ramifications est encore ouvert aujourd'hui, mais en 1993, Gelfond et Lifschitz [45] proposent un nouveau formalisme fondé sur une approche systématique, la logique d'action \mathcal{A} , qui a conduit à des avancées significatives dans ce domaine. L'une des idées principales de ce formalisme a mi-chemin entre l'approche « *fluent* » du calcul des situation et l'approche « modale » de la logique dynamique est de différencier l'implication classique en logique et la *causalité* dans les actions.

La syntaxe de la logique \mathcal{A} repose sur la description d'une part d'observations et d'autre part d'effets d'actions :

- Les *propositions de valeurs* de la logique \mathcal{A} (les observations) sont de la forme : $L \text{ after } (\alpha_i)$ où L est un littéral et (α_i) une séquence d'actions. Par exemple, dans le YSP, nous écrivons :

$$\neg \text{vivant after tirer}$$

Dans la cas particulier où $(\alpha_i) = \emptyset$, nous écrivons simplement : *initially* L .

- Les *propositions d'effet* d'action de la logique \mathcal{A} (appelées aussi « lois d'action ») sont de la forme : $a \text{ causes } L \text{ if } C$ où a est une action, L un littéral et C une conjonction de littéraux qui constituent le prérequis de la loi d'action (*i.e.* la garde de l'action). Par exemple, pour modéliser l'interrupteur en va et vient, nous écrivons :

$$\begin{aligned} a_i \text{ causes } \neg I_i \text{ if } I_i \\ a_i \text{ causes } I_i \text{ if } \neg I_i \end{aligned}$$

Dans la cas particulier où $C = \top$, nous écrivons simplement $a \text{ causes } L$.

La sémantique de la logique \mathcal{A} repose sur la notion de *A-modèle* de l'espace des propositions obtenu en déroulant les actions possibles, comme dans *Feature & Fluents*, à partir de la situation initiale, décrite par l'ensemble des prédicats de la forme : *initially L*.

Effets indirects

La logique \mathcal{A} fournit une façon élégante de modéliser les actions, mais elle reste assez peu expressive. Par exemple, elle ne permet pas de modéliser des actions non déterministes, des actions concurrentes, des actions à effet indirect, *etc.* Plusieurs extensions de la logique \mathcal{A} ont été proposées pour palier ces manques :

- La logique \mathcal{A}_c [8] pour les actions concurrentes ;
- La logique \mathcal{AR} [50] pour décrire des actions à effets indirects (i.e. des ramifications), des actions non-déterministes.
- La logique \mathcal{C} [51], fondé sur le travail de McCain et Turner [86] pour les explications causales ;
- La logique \mathcal{E} [68] pour représenter les « *narratives* » [3, 73, 119], c'est-à-dire pour l'étude des effets d'actions variables selon les situations (comme *tirer* dans le YSP, qui ne tue pas toujours la dinde), basé à la fois sur la logique \mathcal{A} et le calcul des événements [73].

Gelfond et Lifschitz synthétisent ces différents travaux dans [46].

1.4.3 Planification, diagnostic et explications

Les travaux effectués dans le domaine du raisonnement de bon sens sur les actions et les changements sont à l'origine de nombreux formalismes qui permettent de modéliser les actions d'un système pour pouvoir raisonner dessus. Nous pouvons mettre en évidence l'existence de deux problèmes fondamentaux de raisonnement sur les actions et les changements : la planification et le diagnostic.

1.4.3.1 La planification

La *planification* est une branche du raisonnement sur les actions et les changements dont l'objectif est la construction automatique d'un plan, c'est-à-dire de la ou des séquence(s) d'actions qu'il faut effectuer pour que le système passe de l'état initial à l'un des états finaux, décrits par un ensemble de contraintes [48].

Pour construire un plan, le système être muni d'une représentation formelle de :

- L'ensemble des actions, des préconditions et des effets d'actions du système.

Par exemple, si le formalisme choisi est le calcul des situation :

$$\forall x. \forall y. \forall s. \left[\begin{array}{l} leger(x)(s) \wedge sur(x, y)(s) \wedge proche(y)(s) \\ \rightarrow \left(\begin{array}{l} sur(x, main)(result(prendre(x), s)) \\ \wedge \neg sur(x, y)(result(prendre(x), s)) \end{array} \right) \end{array} \right]$$

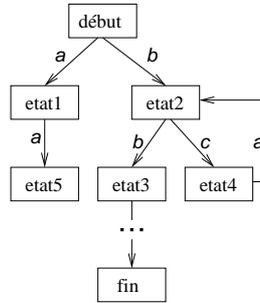
- L'état initial, c'est-à-dire la situation s_0 dans laquelle est le système et les valeurs de vérité des atomes dans cette situation (*i.e.* une interprétation initiale). Par exemple :

$$\begin{aligned} &sur(cube1, table)(s_0) \\ &leger(cube1)(s_0) \\ &proche(table)(s_0) \end{aligned}$$

- Les contraintes sur les états finaux, donnés par un ensemble de *fluents* (ou de propositions d'état en PDL ou en logique \mathcal{A} , par exemple) devant être vrais dans la situation résultat du plan. Par exemple :

$$sur(cube1, main)$$

Un plan peut être visualisé comme un parcours dans le graphe des états, partant de la racine (l'état initial) et allant vers les feuilles (les états buts) :



En calcul des situations, construire un plan revient à trouver un modèle de la formule :

$$état_initial(s_0) \wedge \forall s.lois_actions(s) \wedge \exists s_f.état_initial(s_f)$$

Malheureusement, le calcul des situations est trop général pour que cette méthode soit efficace : trouver un modèle d'une formule du calcul des situations est un problème NP-facile [114]. De plus, le plan obtenu peut contenir des étapes inutiles voir inappropriées. C'est la raison pour laquelle les systèmes de planification utilisent un *langage restreint* et des *algorithmes de résolution spécifique*.

STRIPS

Le premier système de construction automatique de plans, utilisant un langage de description spécifique, est le système STRIPS (STanford Research Institute Problem Solver) de Fikes et Nilsson [41, 40]. Il s'appuie sur un langage particulier dans lequel un état est décrit par une conjonction de littéraux :

$$sur(cube1, table), léger(cube1), proche(table)$$

Les états peuvent être décrits de manière incomplète et les états finaux peuvent contenir des variables quantifiées existentiellement :

$\text{sur}(\text{cube1}, x), \text{couleur}(x, \text{rouge}), \text{forme}(x, \text{cube})$

Les opérateurs (c'est-à-dire les actions de transition possibles entre les états) sont décrites par des schémas, pouvant contenir des variables, de la forme suivante :

$\text{op}(\text{Action: prendre}(x)$
 $\text{Precond: sur}(x, y), \text{léger}(x), \text{proche}(y)$
 $\text{Effect: } \neg \text{sur}(x, y), \text{sur}(x, \text{main}))$

Nous dirons qu'un opérateur est applicable à un état lorsqu'il est instanciable de telle manière que ses préconditions soient vérifiées.

Construction d'un plan

Pour construire un plan, la méthode la plus évidente, mais aussi la moins efficace, consiste à parcourir en largeur le graphe des états jusqu'à trouver un état but. Ce graphe n'est pas représenté en mémoire mais construit au fur et à mesure que les actions sont simulées. Cette technique est appelée technique de *progression*. Mais l'espace des états étant potentiellement immense, la construction d'un plan nécessite la création d'un très grand nombre d'états complètement inutiles car ne faisant pas partie d'un plan-solution. Cette méthode est donc totalement inenvisageable pour la conception d'un système efficace en temps de réponse.

Pour réduire l'exploration de l'espace des états, les chercheurs ont proposé d'utiliser plutôt une technique de *régression*, c'est-à-dire de partir de l'état but et de « remonter » le graphe des états, construit aussi au fur et à mesure, mais cette fois en partant des feuilles. Cette méthode, qui est celle employée par STRIPS, présente l'avantage de réduire l'ensemble des opérateurs appliqués, réduisant ainsi l'espace de recherche. Mais elle présente deux inconvénients majeurs :

1. Elle ne permet pas de retrouver des états initiaux complets, puisque seuls les préconditions de chaque état e_{i-1} sont construites par rétro-application d'une action a dont l'ensemble des effets est inclus dans e_i ;
2. Lorsque le but n'est pas unique, c'est-à-dire lorsque le système doit traiter une *conjonction* de buts, la technique de la régression est soit incomplète (le système ne trouve pas de plan alors qu'il en existe) soit trop coûteuse en complexité.

Espace des plans

La solution utilisée par les chercheurs du domaine est de parcourir l'espace des plans, et non l'espace des états. C'est-à-dire que le système part d'un plan partiel initial décrivant simplement le problème qui est étendu au fur et à mesure jusqu'à constituer un plan complet permettant d'aller de l'état initial vers l'état but. Le

chemin dans l'espace des plans qui conduit du plan partiel initial au plan complet final est alors inutile : seul le plan final atteint compte.

Un plan est décrit dans l'espace des plans par :

- Un ensemble d'opérations de transition entre les états ;
- Un ensemble de contraintes d'ordre sur ces opérations ;
- Un ensemble d'instanciations des variables d'état ;
- Un ensemble de *causes* de la forme (x, c, p) signifiant : « l'action x a été effectuée pour satisfaire la contrainte c du plan précédent p dans l'espace des plans ».

Le plan partiel initial contient uniquement les opérations « début » et « fin », qui sont des actions nulles telle que l'effet de *début* est l'état initial et la précondition de *fin* est la description du but, la contrainte *début* < *fin*. L'ensemble d'instanciation et l'ensemble de causes sont vides.

Deux classes d'opérateurs pour le parcours de l'espace des plans sont alors utilisés : les opérateurs de *raffinement* qui ajoutent des contraintes dans le plan partiel et les opérateurs de *modification* qui corrigent le plan pour satisfaire les contraintes, en ajoutant des opérations (et donc les causes associées) ou en instanciant des variables.

La construction d'un plan se fait alors en parcourant l'espace des plans par application systématique des opérateurs : choix d'un sous-but, choix d'un opérateur, vérification et satisfaction des contraintes. Seul le plan courant est conservé (comme pour la vérification d'une formule en PROLOG) et, en présence de plan qui échouent, le système « *backtrack* » sur les instanciation ou sur les opérations appliquées.

Raisonner sur les plans

De nombreuses recherches ont été effectuées dans ce domaine pour améliorer les algorithmes de construction de plan (meilleure détection des plans morts ou des contraintes à satisfaire, par exemple), prendre en compte des actions concurrentes, le problème de la ramification, etc.

McDermott propose en 1982 [92] une logique temporelle des instants (*time point logics* en anglais) pour raisonner sur les plans et sur les actions dans les plans. Il met ainsi en évidence que le temps dans les plans est généralement discret et propose des méthodes de raisonnement sur la causalité dans les plans.

En s'appuyant sur ces travaux et sur les recherches dans le domaine du raisonnement de bon sens sur les actions, Davis formalise en 1994 [27] les connaissances nécessaires pour qu'un agent puisse construire un plan, en utilisant le calcul des situations.

1.4.3.2 Le diagnostic et les explications

Le *diagnostic* est une branche du raisonnement sur les actions et les changements dont l'objectif est *d'expliquer* l'état d'un système. Plusieurs approches sont possibles :

- Dans l'approche « classique » [24, 9], le système est muni d'une formalisation de l'ensemble des actions, de la modélisation d'un état « anormal » et d'un en-

semble de valeurs de prédicats connues. Un diagnostic est alors un ensemble de valeurs inconnues initialement qui permettent d'expliquer pourquoi le système se retrouve dans l'état anormal, c'est-à-dire tels que cet état est atteignable à en effectuant les actions dès lors que ces variables inconnues prennent les valeurs données par le diagnostic.

Par rapport à la planification, nous pourrions dire que l'état anormal est un but et qu'un diagnostic est l'ensemble des instanciations nécessaires pour atteindre ce but, sans que l'état initial soit donné mais en précisant dans le plan partiel initial un ensemble d'instanciation initiales (les valeurs connues).

- Dans l'approche dynamique [141, 86], un diagnostic est un ensemble d'actions permettant d'atteindre l'état anormal. Cette approche est fondée sur la notion de causalité [51, 82] : il s'agit de déterminer quelle action est à l'origine de quel état et, dans le diagnostic explicatif de McIlraith [93], quelle séquence d'action est la cause de quelle autre action.

La formalisation de la notion de causalité est un problème encore ouvert aujourd'hui [99, 31]. Les nombreux travaux effectués depuis la logique \mathcal{A} , qui est la première à matérialiser cette notion dans un modèle de raisonnement sur les *actions*, ont montré que la construction d'un système qui serait capable de déterminer de manière automatique quelle est la cause (état ou action) d'un état ou d'une action donnée est un problème d'autant plus difficile que parfois, l'humain avec son bon sens [64] n'est pas capable d'y parvenir.

Pourtant, l'étude de la causalité est fondamentale pour la construction d'une explication [77] pour l'utilisateur, comme l'a montré Shanahan en 1993 [131]. En effet, un système interagissant avec un utilisateur ordinaire doit pouvoir répondre à des questions comme : « Pourquoi est-ce que tu as fait cela ? » ou « Pourquoi est-ce que ce truc ne marche plus ? ». Pour y parvenir, le système doit non seulement être capable de construire un diagnostic, mais aussi de donner une réponse *pertinente*, c'est-à-dire dans laquelle la *raison* du problème a été identifiée. Cette notion de *raison d'un problème* dépend alors fortement du modèle causal choisi.

Par exemple, si je renverse un vase, l'eau en séchant fait une auréole sur le tapis. Un observateur extérieur demandera : « Pourquoi y-a-t il une tâche sur le tapis ? ». La réponse : « *parce que l'eau a séché* » n'est pas une réponse pertinente, alors que : « *parce que j'ai renversé le vase* » en est une, conforme au « bon sens » [1]. De même, remonter jusqu'à la cause initiale d'une action ne permet pas nécessairement de construire une explication pertinente. Par exemple : « *parce que je suis passé à côté de la table* » ne constitue pas une bonne explication de l'auréole sur le tapis.

Les études actuelles dans le domaine du raisonnement causal [64] montrent que la recherche de l'action-cause est un problème difficile dans le cas général, et que l'utilisation d'heuristiques indépendantes du domaine ne permet pas toujours d'arriver à un résultat satisfaisant.

1.4.4 Logiques réifiées

C'est justement pour raisonner sur la notion de causalité que furent introduites au début des années 80 les *logiques réifiées*. L'idée commune à l'ensemble de ces formalismes est d'associer au sein d'une même représentation des composants atemporels (des propositions en logique) et temporels (intervalles ou instants). L'objectif est alors de raisonner sur la notion de causalité en fonction des précédences temporelles.

1.4.4.1 La logique de Allen

Le modèle proposé par Allen [3] fait suite à ses travaux sur la logique des intervalles [2]. Dans ce modèle, nous utilisons trois composants de base : les *propriétés*, les *événements* et les *processus*. Il permet alors de modéliser :

- Qu'une propriété p est vraie pendant un intervalle I :

$$holds(p, I) \Leftrightarrow \forall J tq J \subset I, holds(p, J)$$

- Qu'un événement e se produit pendant un intervalle I :

$$occur(e, I) \Leftrightarrow \forall J tq occur(e, J), \neg(J \subset I)$$

- Qu'un processus est en train de se dérouler pendant un intervalle I :

$$occurring(p, I) \Leftrightarrow \exists J \subset I tq occurring(p, J)$$

- Qu'un événement est la cause d'un autre :

$$ecause(e_1, I_1, e_2, I_2)$$

- Que l'agent A est la cause d'une action (événement ou processus) :

$$acause(a, act)$$

L'objectif est alors de modéliser dans ce formalisme des raisonnements causaux, comme « l'agent qui a fait tomber le vase est responsable de la tâche sur le tapis ».

1.4.4.2 La logique des instants

McDermott propose dans [92] une autre approche représenter et raisonner sur les notions temporelles : la logique des instants. Il s'agit d'une logique multi-sortée, en notation préfixe, dans laquelle les entités manipulées sont des *faits* (représenté par des propositions), des *événements* et des *états* (associés à des *instants*).

Dans ce formalisme, le monde est décrit par un ensemble de propositions P et un ensemble d'états S muni d'une fonction de date les reliant à un ensemble continu d'instants. Les états sont partiellement ordonnés par la relation \leq compatible avec la fonction *date*. L'ensemble des états est dense ($s < s' \rightarrow \exists s'', s < s'' < s'$) et le passé d'un état est totalement ordonné ($s \leq s'' \wedge s' \leq s'' \rightarrow s \leq s' \vee s' \leq s$).

Nous dirons qu'une proposition p appartient à un état s ($p \in s$) si et seulement si p est vraie dans l'état s . Dans la représentation graphique de la description d'un problème, les nœuds du graphe temporel sont étiquetés par des instants et les arcs, exprimant des relations de précedence, sont étiquetés par la durée séparant les deux instants (voir figure 1.3).

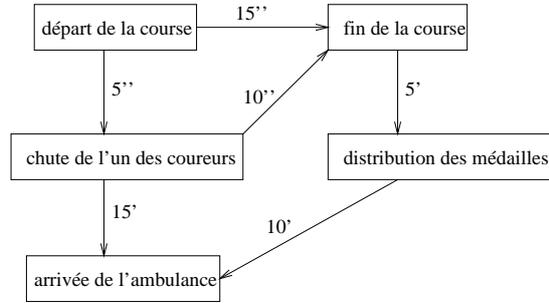


FIG. 1.3 – Un exemple de graphe temporel dans la théorie de McDermott

McDermott définit alors les *événements* par l'ensemble des intervalles pendant lesquels ils se produisent :

- (T S A) exprime que le fait A est vrai dans l'état S ;
- (OCC S₁ S₂ E) exprime que l'événement E se produit entre les états S₁ et S₂ ;
- ECAUSE permet d'exprimer qu'un événement E₁ est toujours suivi d'un autre événement E₂ avant une certaine durée D ;
- PCAUSE permet d'exprimer des causes persistentes, c'est-à-dire que E₂ reste vrai sauf si E₁ devient faux avant un certain délai D ;
- PREVENT et BRING-ABOUT permettent d'exprimer qu'une action exécutée par un agent empêche ou aboutit à un événement.

Le modèle de raisonnement causal de McDermott s'appuie sur la la notion de *chronique*, c'est-à-dire un ensemble d'états en bijection avec l'ensemble \mathbb{R} des réels. Un événement e se produit dans une chronique Γ si et seulement s'il existe un intervalle de cette chronique appartenant à cet l'intervalle défini par e :

$$[s_1, s_2] \subset \Gamma \text{ tq } [s_1, s_2] \in e$$

Ainsi, toute action exécutée crée une chronique différente de celle dans laquelle elle n'aurait pas été exécutée, mais les chroniques diffèrent uniquement pour les états postérieurs à l'exécution de l'action. Dans ce modèle de raisonnement le futur des états est indéterminé.

1.4.4.3 Comparaison des deux formalismes

Pour un problème donné, la représentation dans le formalisme de McDermott est généralement moins concise et plus complexe que la représentation par intervalles d'Allen. De plus, le formalisme de McDermott n'est pas aussi expressif (il n'est pas possible de représenter des disjonctions) et le futur d'un état est indéterminé [59].

Dans [134], Shoham montre que la décomposition des entités proposée par Allen (processus, événement, propriétés) et celle proposée par McDermott (événements, faits) ne sont pas appropriées dans certains cas et insuffisantes dans d'autres. Il suggère d'utiliser plusieurs représentations du temps pour le raisonnement causal déductif.

Dans [12], Bestougeff et Ligozat proposent aussi une dichotomie des entités en faits, événements et processus, qui sont mis en parallèle avec les états, accomplissements et activités du traitement de la langue naturelle. Ils démontrent que les différentes représentations du temps envisageables sont équivalentes.

1.4.5 Raisonnement sur le fonctionnement dans les vues

Dans cette thèse, nous essayons modéliser et d'implémenter des composants actifs qui seraient capables de se représenter leur fonctionnement, de raisonner dessus et de répondre en cours d'exécution des requêtes formelles sur leurs actions et leur exécution, modélisant des questions de bon sens.

Comme nous l'avons montré, les travaux effectués dans le domaine du raisonnement sur le fonctionnement permettent de traiter deux grandes classes de questions :

- Les demandes de plan : « Comment faire pour... », « Fais ceci... », *etc* ;
- Les demandes d'explications, basées sur la construction d'un diagnostic : « Pourquoi est-ce que ceci... ».

Cependant, plusieurs limites demeurent du point de vue de notre problème :

1. Ces formalismes travaillent sur une modélisation du fonctionnement en logique – ce qui pose le problème du décors et des ramifications – alors que nous voulons travailler directement sur le code des composants, pour répondre à des questions en cours d'exécution.

Nous devons donc adapter les méthodes de raisonnement classique pour travailler sur un langage de programmation spécifique.

2. L'interaction avec un utilisateur ordinaire n'a jamais été prise en compte dans ces modèles de raisonnement sur le fonctionnement : ils ne travaillent que sur des requêtes bien formées, produites par un expert.

Pour prendre en compte des *notions de bon sens* employées par l'utilisateur et relier ces notions aux éléments de la vue pour construire des réponses, nous devons proposer un modèle de requêtes sur les actions qui est à la fois :

- Suffisamment expressif pour permettre au MLN de représenter de manière formelle une large classe des questions qu'un utilisateur peut poser concernant le fonctionnement du composant actif ;
- Suffisamment simple pour servir de support pour les algorithmes de construction de réponse du MRF.

A partir de ce langage de requêtes et de la représentation des actions en VDL et en s'appuyant sur des algorithmes de manipulation de notions de bon sens, le MRF peut utiliser les techniques classiques de raisonnement sur les actions

(planification et diagnostic) pour construire des réponses aux questions des utilisateurs.

3. Ces formalismes travaillent sur une modélisation statique des actions. Au contraire, nous voulons pouvoir prendre en compte des processus émergents à partir de l'interaction avec l'utilisateur, qui ne sont pas matérialisés dans la vue mais qui peuvent être perçus par l'utilisateur et, par conséquent, utilisés dans les action, comme nous le montrerons au chapitre 5.

Nous devons donc proposer une méthode dynamique de résolution de ces questions.

1.5 Bilan

Dans cette thèse, nous abordons un problème nouveau : celui de la construction de réponses à des requêtes formelles sur les actions d'un composant, modélisant des questions de bon sens posées par les utilisateurs ordinaires. Les travaux actuels dans la communauté du raisonnement de bon sens [1] permettent de mener à bien des raisonnement logiques conformes au bon sens. Mais les systèmes proposés travaillent sur des requêtes bien formées, ne prenant pas en compte l'existence de *notions de bon sens* partagées par tous les utilisateurs, et à partir d'une modélisation du système par un expert. Au contraire, nous voulons pouvoir travailler directement sur le code du composant, et répondre à des requêtes faisant intervenir des notions de bon sens.

Dans le chapitre 2, nous définissons les éléments procéduraux du langage de description de vues (VDL) et le modèle d'exécution pour les composants actifs. Dans le troisième chapitre, nous présentons un langage de requêtes qui permet de modéliser une large classe de questions que les utilisateurs peuvent poser concernant les actions, le fonctionnement, l'activité ou le comportement d'un composant actif. Dans le chapitre 4, nous donnons les algorithmes de traitement des requêtes qui permettent de construire une réponse en fonction de la description de la vue à l'instant courant. Nous montrons qu'il est possible de prendre en compte des requêtes qui font intervenir des notions de bon sens et nous présentons un cadre formel qui permet de munir les composants actifs de « connaissances de bon sens » ainsi que les mécanismes de manipulation de ces connaissances dans les requêtes. Dans le cinquième chapitre, nous présentons un mécanisme de mémorisation de l'exécution d'un composant et nous montrons comment cette approche dynamique peut être utilisée d'une part pour répondre aux requêtes et d'autre part pour dégager un comportement, individuel dans le cas d'une interaction avec l'utilisateur ou collectif dans un SMA. Enfin, le chapitre 6 présente une application de VDL au web sémantique et le chapitre 7 décrit l'implémentation de nos travaux réalisée dans le cadre du projet *InterViews*.

Chapitre 2

VDL Procédural

Dans le cadre de notre étude et du projet *InterViews*, nous souhaitons modéliser et implémenter des composants actifs, appelés *vues*, capables de se représenter leur fonctionnement et de raisonner dessus pour répondre *en cours d'exécution* à des requêtes formelles portant sur leurs actions et leur exécution. Ces composants sont décrits dans le langage spécifique VDL (pour *View Design Language*) qui est à la fois le formalisme de représentation interne des connaissances des vues, le langage de programmation des vues, le modèle dans lequel leur exécution s'effectue et le support du raisonnement sur les connaissances et le fonctionnement des vues.

Dans ce chapitre, nous présentons la partie procédurale du langage VDL, c'est-à-dire les éléments qui permettent de décrire le fonctionnement d'une vue et le formalisme d'exécution des vues. Dans la première section, nous étudions la problématique de la modélisation des composants actifs, les contraintes imposées par le contexte de notre étude. Dans la deuxième section, nous présentons le modèle d'exécution choisi. Nous donnons ensuite, dans la troisième section, la sémantique opérationnelle de notre langage. Nous mettons en évidence plusieurs propriétés importantes pour notre étude et nous discutons certains choix. Enfin, nous illustrons les possibilités d'extension du modèle en présentant des éléments qui permettent de décrire une interface graphique simple.

2.1 Analyse de la problématique

2.1.1 Le projet *InterViews*

Le projet *InterViews* [120] propose de définir le concept de *vue* à partir de la problématique naissante des agents assistants d'interface. Une vue est un composant logiciel actif intégrant un petit monde opératoire et capable de représenter son propre fonctionnement, de raisonner dessus et d'interagir à un haut niveau sémantique pour recevoir des requêtes, fournir des explications et participer à des tâches d'assistance.

Pour qu'une vue soit capable de répondre *en cours d'exécution* à des questions sur son fonctionnement, il est nécessaire de pouvoir définir à tout instant ses connais-

sances statiques et ses actions. L'idée de base qui fonde la notion de vue est donc la *réflexivité* : il faut qu'une vue puisse représenter ce qui se passe en elle.

C'est l'ensemble de ces contraintes qui nous a amené à choisir de décrire les vues dans un langage spécifique, que nous appelons VDL pour *View Design Language* et qui intègre tous ces éléments. Nous appelons alors *vue* le composant décrit dans ce formalisme, qui comprend un corps (la structure statique et dynamique, en VDL), une exécution (l'état courant de la vue et la manière dont elle évolue) et une partie dialogique (décrivant ses capacités d'interaction en langue naturelle).

Une vue est donc caractérisée par un terme VDL contenant des éléments structurels (qui changent au cours de l'exécution) et des éléments procéduraux de même nature.

Objectifs

Dans ce chapitre, nous présentons la partie procédurale de VDL, c'est-à-dire :

- Les éléments procéduraux de VDL, c'est-à-dire les termes du langage qui permettent de décrire le comportement dynamique des composants ;
- La fonction d'exécution des vues, c'est-à-dire la manière dont ces termes sont utilisés pour transformer le petit monde opératoire intégré dans les vues.

Nous présentons ces éléments sous la forme d'un ensemble cohérent et autonome, c'est-à-dire comme un langage de description de vues à part entière, dans lequel la partie structurelle et les éléments de TALN sont réduits à leur plus simple expression. Nous appelons *VDL procédural* (ou *VDLp*) le langage de description de vues ainsi obtenu. La définition de ce langage comprend une caractérisation formelle des composants procéduraux de VDL, un schéma d'exécution pour les composants et une sémantique opérationnelle pour le langage VDL. Elle complète les versions 0.1 à 0.5 du langage VDL [121, 122, 123, 124] qui proposent un cadre général pour la structure statique et les aspects dialogiques des composants actifs que nous souhaitons modéliser dans le projet *InterViews*.

2.1.2 Contraintes de VDL

La définition du langage VDL (non réduit à sa partie procédurale) repose sur un ensemble de règles imposées par les objectifs fixés dans notre étude. Nous pouvons les regrouper de la manière suivante :

2.1.2.1 Contraintes structurelles

1. A chaque instant, le corps VDL de la vue décrit entièrement le petit monde opératoire manipulé ;
2. Cette description doit être *naturelle*, à la manière des graphes conceptuels [137], pour pouvoir être reliée aux outils de TALN et utilisée pour la construction de requêtes formelles ;

3. La partie structurelle de VDL, doit être riche pour permettre la description de données structurées. Notre objectif n'est pas simplement de décrire des valeurs de vérité de prédicats, mais, comme dans tout langage de programmation, de pouvoir modéliser des données de manière structurée pour pouvoir raisonner sur cette structure en réponse à des requêtes sémantisées.

Par conséquent, les éléments du langage VDL doivent modéliser à la fois des notions structurelles et des notions de TALN, utilisées par le MLN pour la production de requêtes lors de la demande d'explications.

2.1.2.2 Contraintes sur la partie dynamique

4. Le corps de la vue doit aussi décrire entièrement son fonctionnement, c'est-à-dire la manière dont le petit monde opératoire manipulé évolue et peut interagir avec le monde extérieur, comme dans tout langage de programmation ;
5. Les vues doivent intégrer la notion de monde ouvert [61] pour pouvoir interagir avec les utilisateurs et avec d'autres composants actifs ;
6. La description du fonctionnement doit être accessible à *tout instant de l'exécution*, pour que les vues puissent répondre à des questions sur leur fonctionnement en cours d'exécution.

Les méthodes de description d'architectures de composants, comme CORBAIDL ou les Java-Beans, ne permettent pas au système de répondre à des questions qui mettent en jeu des connaissances sur le comportement interne du composant : elles ne peuvent travailler qu'au niveau des liens entre les modules, définis a priori dans le système lors de sa conception. Pour répondre à des questions sur le fonctionnement d'un composant, il faut au contraire pouvoir raisonner sur son code, c'est-à-dire le munir de capacités d'introspection sur la description de ses propres actions.

7. La description en VDL des fonctionnalités de la vue doit pouvoir servir de support aux mécanismes explicatifs. Chaque élément procédural du langage (c'est-à-dire chaque élément intervenant dans la description du fonctionnement) doit être justifié :
 - Soit d'un point de vue opérationnel – il correspond à une fonctionnalité spécifique, qui n'est pas représentable autrement ;
 - Soit du point de vue de l'explication – il modélise une notion naturelle qui apparaîtra dans les explications données en réponses aux requêtes de l'utilisateur, qui ne peut pas être extraite autrement de la description du fonctionnement.

Nous dirons que chaque élément de VDLp est muni à la fois d'une sémantique *opérationnelle* et d'une sémantique *explicationnelle*, conforme au bon sens, et qu'au moins l'une des deux justifie son existence au sein du langage.

8. Le corps de la vue en VDL doit *décrire* le fonctionnement du composant, *i.e.* ce qui fait évoluer les données évoluent, mais pas les *algorithmes* de cette évolu-

tion, contrairement aux langages de programmation classiques dans lesquels le programmeur décrit les algorithmes à effectuer pour transformer des données.¹ Chaque action VDL doit donc être décrite comme un élément du monde opératoire : nous n’exprimons pas ce que font les actions, mais de quelles sous-actions elles sont composées, et ceci jusqu’aux actions de base, dont la sémantique opérationnelle est bien définie et qui doivent être trivialement explicables, ce qui n’est possible que si elles correspondent à des manipulations naturelles sur le monde opératoire de la vue.

C’est en ceci que VDL n’est pas un langage de programmation au sens classique du terme, et c’est ce qui rend impossible l’utilisation de formalismes comme LISP, C ou Caml pour décrire la partie procédurale de VDL : l’utilisation de tels langages conduit, pour mettre en oeuvre un mécanisme explicatif, à utiliser le nom des fonctions pour expliquer ce que fait l’action, comme dans [145]. Nous voulons au contraire pouvoir expliquer entièrement le corps de l’action.

2.1.2.3 Contrainte d’unification

9. La partie structurelle et la partie opérationnelle doivent être intégrés au sein d’une même représentation, de façon à pouvoir être utilisés conjointement dans les outils de raisonnement et d’interaction avec le MLN (développés au sein du projet *InterViews*). En effet, lors de la mise en oeuvre des mécanismes explicatifs, des éléments aussi bien statiques que dynamiques peuvent entrer en jeu simultanément. Il faut alors que le raisonnement porte sur ces deux parties.

2.1.3 Modèles de descriptions de fonctionnement

Alors que les langages de programmation « classiques » ne sont pas étudiés pour la constructions d’explications, au moins quatre modèles de description d’un fonctionnement ont été proposés dans la littérature dans ce but et implémentés sous la forme de langages de programmation. Nous les présentons ici et nous montrons qu’aucun ne satisfait toutes nos contraintes.

2.1.3.1 QPT

La physique qualitative a été introduite en 1979 par [60] pour pouvoir prédire et expliquer (par une preuve) le comportement d’un système physique. Nous pouvons mettre en évidence trois approches du raisonnement qualitatif [59, chap 7] :

- L’approche centrée contraintes de Kuipers [74, 75], permettant la prédiction du comportement d’un système physique.

¹Chaque langage de programmation étant mieux adapté à la description d’un certain type d’algorithme. Par exemple, C pour l’algorithmique de bas niveau, LISP ou Caml pour la programmation fonctionnelle, Java pour la programmation par objets, *etc.*

Dans ce modèle, l'espace des valeurs est un ensemble de couples (*intervalle, changement*). Par exemple, $vitesse = \langle [0, +\infty], \searrow \rangle$ exprime une vitesse positive décroissante. L'utilisateur énonce alors des lois de transition qualitatives pour décrire les changements et le système QSIM interpole l'évolution des paramètres du système.

- L'approche centrée composants de De Kleer et Brown [66].

Dans ce modèle, l'ensemble des valeurs se réduit à $\{\ominus, 0, \oplus\}$ muni des opérateurs usuels. Il faut ensuite écrire des équations différentielles qualitatives (par exemple $\delta P + \delta A - \delta Q = 0$, $\delta x / \delta t = \oplus$, etc) et des lois qualitatives valables dans certains états (ex : $P=Q$ dans e_1). Une preuve en logique du premier ordre permet alors de prédire ou d'expliquer le comportement du système.

- L'approche centrée processus de Forbus [44], dont l'objectif est de représenter de manière explicite et structurée les processus.

Dans ce modèle, appelé QPT pour *Qualitative Process Theory*, les composants sont décrits sous la forme de *vues individuelles* intégrant la description des valeurs et les processus agissant dessus. Les éléments d'une *vue* sont soit des objets, soit des vues. Chaque vue est munie de *conditions* qui doivent être vérifiées pour que la vue soit active et de relations de proportionnalité qualitative entre les individus (par exemple, a est proportionnel à b).

Un processus est représenté de la même manière qu'une vue mais il est muni en plus d'une partie spécifiant son influence sur les objets de la vue. Par exemple, on pourra dire que le processus p influence la variable a positivement. Les influences indirectes sont ensuite obtenues par propagation suivant les relations entre les individus. Les conditions de la vue jouent alors le rôle de garde pour le processus.

De ce point de vue, la description des composants dans le formalisme de Forbus se rapproche des objectifs fixés par le projet *InterViews*. C'est en fait pour cette raison, en référence à QPT, que les composants sont appelés *vues* dans notre projet.

Cependant, dans notre étude, nous ne souhaitons pas modéliser des changements qualitatifs mais bien représenter un fonctionnement quantitatif tel que pourrait le faire un langage de programmation classique. En particulier, les données dans QPT ne sont pas structurées (contrainte 3). De plus, une limite importante du formalisme de Forbus est qu'il ne prend en compte l'existence que d'une sorte d'actions (les processus) et ne permet donc pas de modéliser des systèmes interactifs (contrainte 5). Il s'agit simplement de prédire l'évolution du monde en fonction de contraintes données.

C'est pourquoi nous ne pouvons pas décrire les vues en QPT, mais nous en reprendrons les idées répondant à nos contraintes :

- Décomposition des vues en sous-vues ;
- Représentation des actions sous la forme de vues gardées et notion de *processus* agissant en dehors de toute interaction avec l'utilisateur ;
- Description simplement des effets directs, comme dans le langage de programmation classiques.

Ces trois propositions nous conduisent à proposer un modèle fondé sur la réécriture d'arbres (les vues) dans lesquels les nœuds décrivent :

- soit des données structurées,
- soit des préconditions pour les actions et les sous-actions,
- soit des effets directs d'actions (nous parlerons d'actions élémentaires).

2.1.3.2 Les algèbres évoluant

Un autre grand formalisme de modélisation de monde opératoire a été proposé par Gurevich en 1991 à travers les algèbres évoluant (ou EA pour *Evolving Algebras*), rebaptisées depuis *Abstract State Machines* [54]. Dans ce modèle, un **état** est une algèbre multi-sortée, contenant au moins les opérateurs booléens usuels (*true*, *false*, et les opérations classiques) et l'opérateur d'égalité. Les ensembles de sorte possèdent une signature et vérifient les propriétés usuelles des Σ -algèbres de types utilisées en programmation. L'algèbre est munie d'opérations *partielles*, c'est-à-dire dont le résultat peut être la valeur particulière *undef* de l'algèbre (par exemple lorsqu'elle n'est pas appliquée sur les bonnes sortes).

L'exécution d'une EA repose sur les notions de lieu (*location*) et de mise à jour (*update*) sur des états. Un lieu est un élément de l'algèbre, et une mise à jour correspond à une assignation possible de nouvelles valeurs de l'algèbre pour des lieux. Tirer une mise à jour, c'est remplacer la valeur des lieux de l'état indiqués dans la mise à jour par les nouvelles valeurs, et ceci simultanément pour tous les lieux concernés, construisant ainsi un nouvel état. Le modèle d'exécution repose donc sur l'idée de la réécriture de l'état (l'algèbre) à chaque pas d'exécution (*update*).

Pour programmer une EA, le concepteur utilise des *règles* qui peuvent prendre la forme suivante :

- Des instructions de mise à jour de la forme $f(t) = t'$;
- Une séquence de règles, construite à l'aide de l'opérateur de séquences noté \cup ;
- Une instruction conditionnelle : si k est un entier, g_0, \dots, g_k des termes booléens de l'algèbre et R_0, \dots, R_k des règles, alors l'expression suivante est une règle :

```

if  $g_0$  then  $R_0$ 
elseif  $g_1$  then  $R_1$ 
...
elseif  $g_k$  then  $R_k$ 
endif

```

Gurevich montre qu'il est possible de modéliser le langage de programmation C à l'aide de ces seules règles de base [55]. En ajoutant une fonction d'import que nous n'explicitons pas ici² il devient possible de modéliser n'importe quel algorithme séquentiel par une règle d'EA. Nous parlons alors de programme P .

L'exécution d'un programme est une suite d'états (S_n) telle que S_{n+1} est obtenu en tirant simultanément toutes les mises à jours possibles de P sur S_n , d'où le nom

²L'instruction *import* permet d'utiliser des variables quantifiées existentiellement dans les règles, ces variables étant « importées » temporairement dans le programme.

d'algèbre évolutive. En cas de concurrence entre deux *update* pour la modification d'un même *lieu*, le choix de la nouvelle valeur est non déterministe, localement au *lieu* considéré (Gurevich parle de *non-déterminisme local*).

Implémentation

Il existe plusieurs implémentations des EA. Les premiers travaux dans ce sens, proposés par Kappel [69] consistent d'une part en un langage de spécification d'EA, baptisé DASL pour *Dynamic Algebras Specification Language*, et d'autre part en un interprète DASL. Del Castillo propose plus tard un environnement de spécification, de développement et d'exécution unifié permettant de décrire des EA distribuées [30].

En 1995, Schönegge [127] propose une extension de la logique dynamique [104] pour raisonner sur les Algèbres Evolutives. Il montre ainsi que ces deux formalismes permettent de représenter et de raisonner sur les mêmes entités.

Limites

Les EA constituent (avec les logiques de réécritures) l'un des modèles formels les plus puissants pour décrire l'exécution d'un composant. Cependant, dans ce modèle, la description des actions de modification peut être aussi riche que le souhaite le programmeur de l'EA (la fonction f peut être aussi complexe que l'on veut). Dans ce cadre, il n'est pas possible de raisonner sur les actions de modifications effectuées dans l'algèbre *en vue de fournir des explications* à un utilisateur ordinaire (contraintes 7 et 8), et nous retombons dans le problème des langages de programmation classiques.

C'est la raison pour laquelle nous ne pouvons pas décrire les composants actifs sous la forme d'EA. En revanche, nous en conserverons l'idée principale, c'est-à-dire la réécriture du monde à chaque cycle d'exécution. De cette façon, les vues ont accès à leur fonctionnement *en cours d'exécution* pour pouvoir raisonner dessus et répondre à des requêtes construites par le MLN.

2.1.3.3 Les logiques de réécriture

Les logiques de réécritures ont été introduites en 1990 par José Meseguer [95]. L'objectif de ce modèle est de relier la modélisation en logique et l'implémentation d'un calcul. L'état d'un système est représenté par un ensemble de formules et l'exécution concurrente des calculs dans le système est modélisé par des *preuves* (au sens de la logique). Les axiomes de la logique de réécriture sont des expressions de la forme :

$$t \longrightarrow t'$$

qui modélise une transaction concurrente : la sous-partie t de l'état du système peut être transformée en t' , de manière concurrente avec tous les autres changements

d'états. Du point de vue logique, $t \rightarrow t'$ signifie simplement que t' peut être déduite de t .

Les formules t modélisant les états peuvent être de nature quelconque, pour permettre de représenter des types de données. Les logiques de réécriture permettent donc à la fois de modéliser et d'exécuter des programmes dans un langage spécifique ou de prouver des formules dans une logique donnée.

L'exécution (c'est-à-dire la preuve) d'un programme en logique de réécriture s'appuient sur les quatre règles de déduction suivantes :

1. Réflexivité ($t \rightarrow t$);
2. Congruence ($t \rightarrow t' \vdash f(t) \rightarrow f(t')$);
3. Remplacement ($t_{(x)} \rightarrow t'_{(x)} \vdash t_{(x/y)} \rightarrow t'_{(x/y)}$);
4. Transitivité ($t \rightarrow t', t' \rightarrow t'' \vdash t \rightarrow t''$).

Maude

Il existe de nombreuses applications des logiques de réécritures, mais la plus importante est certainement le système *Maude* [96]. Maude est à la fois un langage et un système permettant de modéliser et d'exécuter une très large classe de programmes concurrents dans différents langages, en particulier en programmation concurrente orientée objet.

Limites

Du point de vue de notre étude, les logiques de réécritures présentent des avantages et des inconvénients similaires à ceux des algèbres évoluant. En particulier, elles sont trop riches pour pouvoir raisonner sur les actions de modifications et les règles de déduction ne sont pas munis d'une sémantique explicative. C'est la raison pour laquelle nous ne pouvons pas décrire les composants actifs en utilisant directement des logiques de réécritures.

Cependant nous voudrions retrouver en VDL plusieurs propriétés de ce modèle. Nous reprendrons l'idée d'un petit ensemble de règles de modifications, pour permettre de générer des explications sur le fonctionnement sur une base suffisamment réduite, ainsi que le principe générale de la réécriture du composant à l'aide d'actions concurrentes.

2.1.3.4 GOLOG

Le modèle le plus répandu en raisonnement sur les actions étant le calcul des situations, nous avons envisagé au début de notre étude d'utiliser le langage GOLOG [79] pour décrire les composants. Proposé en 1995, ce langage dont le modèle d'exécution a été implémenté en PROLOG s'appuie sur une idée théorique simple : la modélisation en calcul des situations des opérateurs classiques des langages de programmation (*if*, *while*, *etc*), à la manière de la logique dynamique [104] :

- $\phi?$ pour le test : $Do(\phi?, s, s') \equiv_{def} \phi[s] \wedge s = s'$, c'est-à-dire que l'action $\phi?$ permet de passer de la situation s à la situation s' si et seulement si ϕ est vérifiée dans la situation s et $s = s'$;
- $a; a$ pour la séquence : $Do(a_1; a_2, s, s') \equiv_{def} \exists s'' tq Do(a_1, s, s'') \wedge Do(a_2, s'', s')$;
- $a|a$ pour le choix non-déterministe : $Do(a_1|a_2, s, s') \equiv Do(a_1, s, s') \vee Do(a_2, s, s')$
- a^* pour l'itération 0 ou n fois (la formule exacte en calcul des situations n'apporte rien ici);

Le fonctionnement d'un composant actif est alors décrit dans ce modèle, sous la forme de règles logiques du calcul des situations « étendu » puis le composant peut être exécuté en appliquant ces règles en PROLOG.

Limites

Cependant, le modèle GOLOG permet uniquement de décrire des changements de valeur de vérité de prédicats de la logique du premier ordre. Comme QPT et de manière générale les formalismes issus des travaux sur le raisonnement en logique, il ne permet pas de modéliser des données structurées (contrainte 3) comme nous le souhaitons dans le projet *InterViews*.

De plus, malgré sa richesse d'expression, GOLOG reste un langage de modélisation : c'est le comportement du système qui est décrit dans le modèle, et non le système lui-même (contrainte 6). Par conséquent, le raisonnement ne peut s'effectuer que sur les éléments qui ont été spécifiés, comme pour les architectures à composants. Cela signifie d'une part qu'il faut qu'un expert ou un programmeur décrive le fonctionnement du système dans le formalisme considéré et d'autre part que les mécanismes de raisonnement n'ont accès qu'à cette partie du fonctionnement qui a été explicitement décrite.

Au contraire, nous voulons partir directement du code du système pour expliquer son comportement à l'utilisateur, sans travail préparatoire de la part du programmeur ou d'un expert, et en ayant accès à l'intégralité du fonctionnement.

Enfin, les modèles logiques ne satisfont pas la contrainte de naturalité (2) : les prédicats représentent toujours de notions arbitrairement choisies par l'expert pour modéliser le système, qui ne sont pas forcément adaptées à un traitement par des outils de langue naturelle en vue d'une interaction avec des utilisateurs ordinaires.

C'est la raison pour laquelle, dans le projet *InterViews*, nous proposons une approche spécifique, fondée à la fois sur l'implémentation directe du système dans un langage de programmation et sur l'utilisation d'outils de traitement de la langue naturelle et de raisonnement sur le fonctionnement qui pourraient s'appuyer directement sur ce langage.

Nous ne pouvons donc pas décrire nos composants en GOLOG. Toutefois, nous en reprendrons une idée importante : nous montrerons que notre langage permet de modéliser tous les opérateurs classiques utilisés dans les langages de programmation.

2.1.3.5 SNePS

Les réseaux sémantique sont une technique de représentation des connaissances [70] fondée sur des graphes dont les nœuds modélisent les objets et les arcs (orientés) des relations binaires. Ils sont particulièrement utilisés pour définir des ontologies en traitement du langage naturel (TALN) et en traduction automatique mais ils peuvent aussi permettre de représenter des connaissances structurées (les données d'un système), des raisonnements en logique, des procédures exécutables (actions) ou des réseaux apprenants (réseaux de neurones par exemple).

En s'appuyant sur cette technique, le système SNePS [133] a pour objectif de représenter et de raisonner sur les connaissances et les croyances d'agents conversationnels, commandés en langage naturel. Toutes les entités manipulées par l'agent (objets, classes, personnes, propriétés, abstractions, actions, temps, propositions et règles) sont modélisées sous la forme de nœuds dans un réseau sémantique. Le réseau, qui représente les croyances courantes et passées de l'agent, est modifié au fur et à mesure de l'exécution. La version courante de SNePS comprend le modèle de représentation des connaissances (SNePS), le système d'inférence (SNIP), le modèle de révision de croyances (SNeBR) et le modèle de description et d'exécution des actions (SNeRE), incluant un système de planification. Ce modèle est implémenté suivant une architecture en couche (GLAIR) comprenant :

- La couche « matérielle » des capteurs physiques et des composants agissant physiquement sur le monde ;
- La couche « sensori-motrice » dans laquelle sont codés les connaissances de base et les actions élémentaires du système ;
- La couche de représentation des connaissances (SNePS).

Le modèle de représentation des actions (SNeRE) utilise une vingtaine d'opérateurs (appelés *acts*) de contrôle, de manipulation et de modification des croyances permettant de combiner des actions élémentaires (définies dans la couche sensori-motrice) en actions complexes. Le choix des différents opérateurs de composition d'actions fait de SNePS un langage très expressif.

Limites

Les réseaux sémantique sont initialement orientés vers sur la représentation de connaissances structurées, même s'il est possible de modéliser et de travailler sur des actions. Dans ce contexte, la définition des actions par le « programmeur en SNePS » fait appel à des combinaisons spécifiques qui dépendent fortement des actions élémentaires entrant en jeu. La sémantique explicationnelle des opérateurs n'est donc pas définie a priori (contrainte 4) mais en fonction de l'utilisation qui en est faite pour modifier le réseau sémantique lors de l'exécution des actions élémentaires. C'est la raison pour laquelle nous ne pouvons pas utiliser SNePS pour représenter et manipuler les connaissances des agents. Cependant, nous devons reprendre les idées suivantes en VDL :

- Nous devons utiliser un ensemble restreint et bien défini d'opérateurs de com-

binaison et de manipulation des connaissances.

- De même que GLAIR sépare les actions élémentaires (définies dans la couche sensori-motrice) et les actions de manipulation des connaissances, nous devons pouvoir définir des actions spécifiques au composant (correspondant aux fonctionnalités d’interaction du composant physique modélisé) et leur interprétation doit se faire de manière externe et transparente par rapport à la sémantique opérationnelle du langage.
- De même que l’exécution d’un composant en SNePS conserve les croyances passées de l’agent, nous voudrions que l’exécution d’un composant en VDL (vue comme la modification des croyances du système et mémorisée comme telle) soit intégrée avec la description de ses fonctionnalités opératoires. Ainsi, l’exécution du composant modifie la description du système, comme dans les logiques de réécriture, mais la nouvelle description intègre la mémorisation des actions effectuées.

2.1.4 Définition de VDL procédural

L’analyse du problème et des modèles de représentation d’actions que nous avons étudié nous a conduit à proposer un langage de description des vues spécifique fondé sur la réécriture (comme dans les EA [54] ou *Maude* [94, 96]) d’une structure récursive (comme dans QPT [44]), c’est-à-dire représentable par un arbre. Le modèle temporel de l’exécution est donc un ensemble discret d’instantanés discrets. La vue au prochain instant v_{t+1} est calculée à partir de la vue à l’instant courant v_t , décrite en VDL, à chaque pas d’exécution.

Comme dans les vues QPT [44], le langage qui décrit le corps d’une vue (*i.e.* le petit monde opératoire qu’elle manipule) est aussi celui dans lequel sont décrits ses éléments procéduraux (c’est-à-dire le code responsable de son exécution). Cela nous permet d’intégrer dans une représentation sémantique *unifiée* les fonctionnalités opératoires (la partie exécution de la vue) et les connaissances de la vue sur le monde qu’elle manipule (la partie raisonnement de la vue).

Chaque nœud de l’arbre doit modéliser à la fois une notion structurelle ou procédurale (les objets des vues QPT) et des éléments de TALN. C’est la raison pour laquelle nous dirons que ces nœuds sont des *concepts* et nous dirons que chaque concept est muni d’une sémantique opérationnelle et d’une sémantique explicationnelle. Cependant, dans cette thèse, nous nous intéressons uniquement à l’étude du fonctionnement dans un composant décrit en VDL. Nous ne prenons pas en considération les problèmes liés au traitement du langage naturel ni les aspects structurels pouvant intervenir dans la construction de requêtes. C’est la raison pour laquelle nous nous limiterons à la définition d’un langage orienté vers la description du fonctionnement des vues, dans lequel la partie structurelle et les éléments de TALN sont réduits à leur plus simple expression. Nous représenterons les concepts par de simples chaînes de caractères, sans aucune notion de type : le corps de la vue est un simple arbre de chaînes de caractères. Nous parlerons cependant toujours de concept pour garder en mémoire l’idée que ces concepts seront, à terme, enrichis pour modé-

liser des notions structurelles riches et des éléments nécessaires au TALN. La partie structurelle a été étudiée par E. Valencia [126] et la partie TALN par S. Gérard [52] et G. Pitel [125, 102].

Enfin, comme dans les langages de programmation classiques et comme en physique qualitative [60], le système décrit uniquement les effets directs des actions. Le problème du décors, de la ramification ou de la qualification ne se posent alors pas dans la description du monde opératoire en VDL. En cela, VDL est un véritable *langage de programmation*, utilisable par exemple dans des applications de commerce électronique, de domotique ou dans des SMA. Nous montrerons qu'il vérifie les propriétés des langages de programmation classiques : il est Turing-complet et il permet d'exprimer simplement les opérations élémentaires de composition d'instruction utilisées des langages de programmation usuels.

2.1.5 Plan

Dans ce chapitre, nous présentons le langage VDL procédural (par abus de langage, nous parlerons de VDL dans la suite du document), qui consiste d'une part en la donnée des éléments des vues décrivant leur exécution et d'autre part en la description de cette exécution elle-même, c'est-à-dire la sémantique opérationnelle de ces éléments. Ces travaux font suite aux premières versions du langage VDL, défini par J.P. Sansonnet dans [121] et [122], ainsi qu'à un premier manuel de l'utilisateur VDLp [115].

Dans la deuxième section, nous présentons le principe général de VDL et le modèle d'exécution des composants actifs décrits dans ce langage. Nous l'illustrons sur un exemple simple ne faisant pas intervenir de dynamique.

Dans la troisième section, nous présentons de manière formelle la sémantique opérationnelle des vues, c'est-à-dire la mécanique de l'exécution des vues. Nous donnons l'ensemble des concepts propres à la description de l'exécution, nous décrivons les fonctions de calcul annexes utilisées pour la construction de la vue v_{t+1} à partir de la vue v_t et nous présentons le mécanisme de gestion de la concurrence mis en œuvre dans ce modèle. Nous illustrons cette sémantique opérationnelle sur un exemple simple. Nous montrons enfin que le langage VDLp et le modèle d'exécution proposés vérifient les propriétés usuelles des langages de programmation, *i.e.* qu'il est à la fois expressif et complet d'un point de vue opérationnel.

Dans la quatrième section, nous présentons une extension du langage qui permet de munir les vues d'interfaces graphiques (GUI). Nous donnons l'ensemble des concepts utilisés pour la description d'une GUI et nous présentons la sémantique de ces concepts. Nous illustrons cette extension du langage VDL sur un exemple graphique simple.

2.2 Principe général

2.2.1 Partie structurelle

2.2.1.1 Définition

Le corps d'une vue, c'est-à-dire la description en VDL du monde opératoire qu'elle manipule, est un arbre dont les nœuds sont des chaînes de caractères, appelées *concepts*, dont les branches ne sont pas étiquetées, et dont la racine est le concept « *view* ».

Dans le cadre de l'étude de la partie procédurale de VDL, nous appellerons abusivement *vue* cet arbre de concepts (qui est en fait le corps de la vue). Nous appelons *terme* tout sous-arbre de la vue et, pour un terme donné, nous appelons *attributs* ses sous-termes directs. Nous appellerons *atome* un terme qui n'a pas d'attributs. Les attributs d'un terme non atomique forment un multi-ensemble ordonné³.

Nous notons Υ l'ensemble (infini) des termes VDL, $\Lambda \subset \Upsilon$ l'ensemble des vues (c'est-à-dire des termes de concept *view*) et A^* celui des concepts. L'atome particulier de concept « mot vide » est noté \surd . Il ne peut pas être modifié par quelque action que ce soit et n'est jamais représenté dans une vue.

Conventions

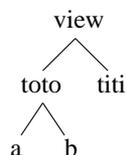
Nous notons \mathbf{B} l'ensemble des booléens, \mathbf{N} l'ensemble des entiers naturels et, pour tout ensemble E , nous notons $\mathcal{P}(E)$ l'ensemble des parties de E .

2.2.1.2 Représentation

Dans ce document, les vues seront représentées de manière textuelle sous la forme suivante : tout terme de concept c et d'attributs st_1, \dots, st_n sera noté $c[st_1, \dots, st_n]$. Par exemple, le texte suivant :

```
view[toto[a,b],titi]
```

représente une vue (le concept racine est *view*) ayant deux attributs : le premier de concept *toto* (ayant lui-même deux sous-termes atomiques : a et b) et le second, atomique, de concept *titi*. Elle correspond donc à la vue suivante :



³C'est-à-dire que l'ordre des attributs peut être utilisé dans les algorithmes de traitement lorsque cela est nécessaire. Cependant, nous essayerons de les rendre le plus indépendants possible de cet ordre. Ainsi, dans le cas général, le terme de concept a et d'attributs $\{b, c\}$ sera équivalent au terme de concept a et d'attributs $\{c, b\}$.

Les espaces et les mises en italique seront utilisés dans les descriptions textuelles pour faciliter la lecture des vues. Par exemple :

`view[toto[a,b], titi]`

Lorsqu'un concept contient des symboles réservés, nous le mettrons entre guillemets droits dans la description textuelle. Par exemple :

`"le crochet ouvrant [et fermant]"[x]`

est un terme de concept « *le crochet ouvrant [et fermant]* » et ayant un seul attribut atomique x .

Convention

Si $n, m \in \mathbb{N}^2$, $0 < n < m$, $\{t_1, \dots, t_m\} \in \Upsilon^m$ et $c \in A^*$, alors nous utiliserons la convention d'écriture suivante dans les algorithmes :

$$c[t_1, \dots, t_n, \{t_{n+1}, \dots, t_m\}] \equiv_{def} c[t_1, \dots, t_m]$$

Grammaire

La grammaire du langage VDL dans sa représentation textuelle est la suivante :

`vue ::= view[sous-termes]`

`sous-termes ::= terme | terme , sous-termes`

`terme ::= concept | concept[sous-termes]`

`concept ::= {a-z,A-Z,0-9}* | "chaîne de caractères entre guillemets"`

2.2.1.3 Fonctions de base

Dans la suite du document, nous utiliserons les fonctions suivantes :

- *concept* : $\Upsilon \rightarrow A^*$ qui donne le concept d'un terme ;
- *fils* : $\Upsilon \rightarrow \mathcal{P}(\Upsilon)$ qui donne l'ensemble (ordonné) des attributs d'un terme ;
- *pere* : $\Upsilon \rightarrow \Upsilon$ qui donne le terme parent d'un terme ou \surd si le terme n'est pas un attribut ;
- *node* : $A^* \rightarrow \Upsilon$ qui construit un terme atomique à partir d'un concept VDL.

2.2.1.4 Notion de terme flottant

Dans nos algorithmes, nous manipulons à la fois des termes rattachés à des vues et des termes qui ne le sont pas. Par exemple, considérons le terme $t = toto[a, b]$ dans deux contextes différents :

1. Si nous écrivons $v = view[c, toto[a, b]]$, t est rattaché à v ;
2. Si nous écrivons simplement $t = toto[a, b]$, t est bien un terme VDL, mais il n'est pas rattaché à une vue.

Nous dirons qu'un terme non rattaché à une vue est un terme *flottant*.

Remarquons que les vues sont elles-mêmes des termes flottants. Nous verrons dans la suite de ce chapitre comment les termes flottants sont utilisés dans les algorithmes et comment un terme flottant peut-être rattaché à une vue.

2.2.1.5 La fonction *vue*

Soit t un terme. Nous définissons la fonction $vue : \Upsilon \rightarrow \Lambda$ qui donne la vue à laquelle un terme est rattaché. Si le terme est flottant ou sous-terme d'un terme flottant qui n'est pas une vue, alors nous aurons $vue(t) = \surd$.

$vue(t)$ est définie récursivement de la manière suivante :

$$vue(t) = \begin{cases} t & \text{si } concept(t) = view \text{ et } pere(t) = \surd \\ \surd & \text{si } concept(t) \neq view \text{ et } pere(t) = \surd \\ vue(pere(t)) & \text{sinon} \end{cases}$$

2.2.2 Notion d'observateur

2.2.2.1 Définition

Le modèle d'exécution que nous proposons pour les composants actifs en VDL s'appuie sur la notion d'*observateur*.

Un observateur est un composant logiciel externe (par exemple un programme Java) qui recherche dans une vue des concepts spécifiques, appelés *modifieurs* et effectue des actions particulières en fonction des modifieurs trouvés.

Un observateur o est donc caractérisé par :

- L'ensemble $\mathcal{M}_o \subset A^*$ des modifieurs reconnus par cet observateur, décrivant la *syntaxe* de l'observateur ;
- Une fonction d'interprétation ϕ_o , implémentée dans un langage de programmation de l'observateur, qui prend en argument une vue $v \in \Lambda$, y recherche les éléments de \mathcal{M}_o et effectue des actions en fonction des modifieurs trouvés.

Elle définit ainsi la *sémantique* de ces modifieurs.

Les vues n'ont donc pas de sémantique propre. Ce sont les observateurs qui définissent la sémantique des concepts dans les vues, et non la vue elle-même⁴.

2.2.2.2 Deux classes d'observateurs

Du point de vue de notre étude, nous pouvons mettre en évidence deux types d'observateurs :

1. **Les observateurs statiques**, qui ne modifient pas la vue de manière proactive. Par exemple :

⁴De même que, dans les langages à balise comme SGML, le document ne définit pas la manière dont il doit être interprété.

- L'observateur d'interface qui gère l'interface graphique utilisateur du composant (modifiée par effet de bord) ;
- L'observateur du MLN qui construit des requêtes à partir des questions de l'utilisateur et de la description du composant à l'instant t ;
- L'observateur de requêtes qui répond aux requêtes construites par le MLN ;
- Un observateur de base de donnée qui modifie une base de données SQL en fonction des modifications effectuées dans la vue ;
- *etc.*

2. **L'observateur dynamique** qui considère la vue passée en paramètre comme la description du composant à l'instant t et qui construit sa description à l'instant suivant $t + 1$ en fonction des concepts spécifiques décrivant le fonctionnement du composant actif, à la manière des EA [54].

Notre modèle d'exécution basé sur les observateurs permet de décrire des documents à la fois structurés et dynamiques. Toutefois, il s'appuie sur l'hypothèse que les observateurs sont robustes : si un élément recherché n'est pas trouvé dans un document ou si le document n'est pas conforme à la syntaxe de l'observateur, alors aucune action n'est effectuée par cet observateur.

2.2.2.3 Modèle d'exécution

L'exécution d'un composant actif dans ce schéma consiste alors en la réécriture de la vue à chaque cycle d'exécution, comme illustré sur la figure 2.1. Les compo-

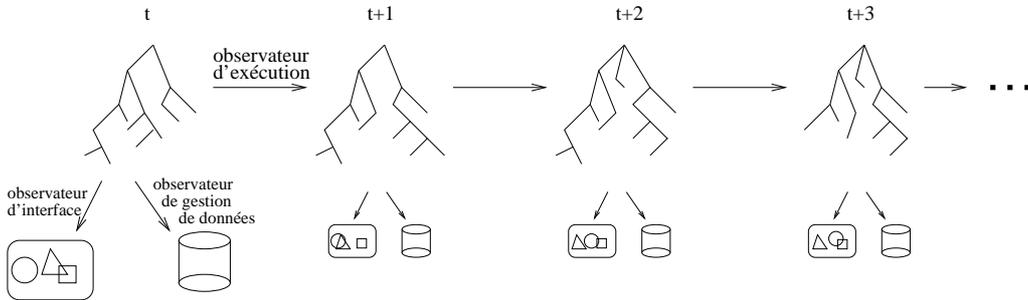


FIG. 2.1 – Les observateurs

sants sont donc proactifs : ils peuvent s'exécuter en dehors de toute interaction avec l'utilisateur. Les observateurs statiques sont alors appliqués sur la nouvelle vue.

Nous appelons \mathcal{S} l'ensemble des observateurs statiques qui doivent être appliqués à chaque modification de la vue. Par exemple, l'observateur d'interface appartient à \mathcal{S} mais pas l'observateur du MLN.

Un cycle d'exécution d'une vue v est alors défini comme étant l'application de la fonction $\Phi = (\circ_{i \in \mathcal{S}} \phi_i) \circ \phi_{exec}$ sur la vue v , c'est-à-dire :

1. La réécriture de la vue par l'observateur *exec* (fonction ϕ_{exec}), ce qui construit une nouvelle vue ;

2. L'application de tous les observateurs statiques (l'ordre importe peu puisqu'aucun ne modifie la vue).

L'exécution de n cycles correspond alors à : $v_n = \Phi^n(v_0)$.

2.2.2.4 Un exemple simple : l'observateur *print*

Imaginons un observateur **statique** *print* qui recherche uniquement le modifieur *print* dans la vue et retourne les concepts de ses attributs. Il est défini par :

- $\mathcal{M}_{print} = \{print\}$;
- La fonction $\phi_{print} : \Lambda \longrightarrow \mathcal{P}(A^*)$ telle que, pour tout terme t de la forme $t = c[t_1, \dots, t_n]$, où c est le concept de t et t_1, \dots, t_n ses attribut,

$$\phi_{print}(t) = \begin{cases} \{\text{concept}(t_i), i \in [1, n]\} & \text{si } c = \text{print} \\ \bigcup_{i=1}^n \phi_{print}(t_i) & \text{sinon} \end{cases}$$

Exemple

Soit v la vue suivante :

`view[print [a, b[c]], d, print [e]]`

Alors $\phi_{print}(v) = \{a, b, e\}$.

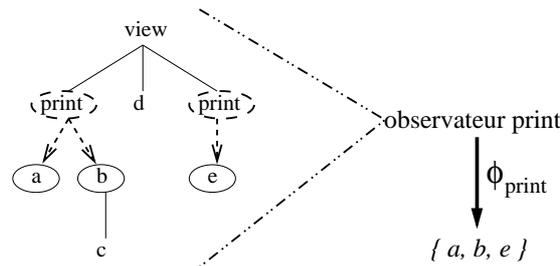


FIG. 2.2 – Un observateur statique simple.

Rappelons qu'un autre observateur *print2*, caractérisé par une autre fonction ϕ_{print2} , peut définir une sémantique complètement différente pour le modifieur *print*. Par exemple, ϕ_{print2} peut *imprimer* les termes à l'écran, par effet de bord, sous la forme d'arbres.

Soulignons aussi que *print* n'est pas un observateur dynamique : il ne modifie pas la vue et ne définit donc aucun *fonctionnement*.

2.2.2.5 Objectifs

Dans ce chapitre, nous présentons deux observateurs :

1. L'observateur d'exécution, appelé *exec*, qui transforme à chaque cycle d'exécution la vue v_t en une nouvelle vue v_{t+1} , en utilisant les *modifieurs procéduraux* présents dans le corps de la vue v_t ;

2. Un observateur statique, *interf*, permettant de munir les composants d'une interface graphique simple.

Dans le modèle d'exécution faisant intervenir ces deux observateurs, $\mathcal{S} = \{interf\}$ et nous avons :

$$\Phi = \phi_{interf} \circ \phi_{exec}$$

Les fonctions utilisées dans ces observateurs sont commentées, expliquées et illustrées sur des exemples simples.

2.3 L'observateur dynamique *exec*

2.3.1 Principe général

2.3.1.1 Modèle d'exécution

La description du fonctionnement en VDL s'appuie sur la notion d'actions élémentaires de modification du composant. Ces actions sont décrites par des termes VDL, dans la vue elle-même. Elles modifient d'autres termes de la vue. Plusieurs actions élémentaires peuvent être regroupées dans un même terme pour composer des actions plus complexes.

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

- Dans le mode *réactif* (ou *modal*), les composants effectuent des opérations en réponse à des interactions de l'utilisateur (par exemple, cliquer sur le bouton « démarrer »). Ce mode correspond à l'appel d'une fonction avec des paramètres, donnés directement ou issus de l'interaction.
- Dans le mode *proactif* (ou *non modal*), les composants fonctionnent en dehors de toute interaction. Par exemple, si l'on considère un joueur de rugby qui transforme un essai, le mouvement du ballon, une fois initié par le coup de pied, s'effectue sans interaction. Ce mode correspond aux boucles en algorithmique, dans lesquelles l'humain n'intervient pas.

Dans notre modèle, nous voulons pouvoir représenter ces deux modes de fonctionnement, schématisés sur la figure 2.3. Pour modéliser l'interaction d'une vue avec un

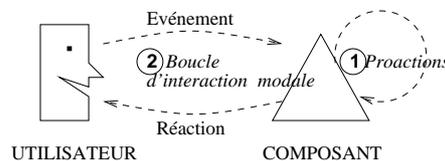


FIG. 2.3 – Les deux modes de base de fonctionnement des composants : 1) comportements proactifs et 2) boucle d'interaction modale

utilisateur humain ou avec d'autres composants, nous avons proposé un mécanisme

basé sur des *événements externes*. Un événement externe est un terme VDL qui peut être envoyé au composant au cours de son exécution pour être interprété par la vue.

Les actions VDL peuvent alors être munies de deux sortes de préconditions :

- Des gardes booléennes, pour les comportements proactifs ;
- Des événements externes, pour le mode réactif.

2.3.1.2 Modifieurs de l'observateur *exec*

Les modifieurs utilisés par l'observateur dynamique *exec* sont les suivants :

- **path** et **get** pour faire référence à des termes de la vue dans les actions ;
- **add**, **put** et **del** pour respectivement ajouter des attributs à un terme, remplacer tous les attributs d'un terme donné par d'autres et enlever un terme dans la vue. Ce sont les *actions élémentaires* ;
- **plus**, **minus**, **times**, **inverse**, **and**, **or**, **not**, **equals**, **"greater than"**, **true**, **false** pour effectuer les opérations arithmétique de base ;
- **random** pour générer des nombres aléatoirement.
- **guard** pour décrire des processus gardés ;
- **event** pour définir des réactions aux événements externes ;
- **"event get"** pour manipuler les éléments de cette interaction ;
- **size** et **"size ref"** pour compter le nombre d'attributs des termes et obtenir des informations *en cours d'exécution* sur la structure de la vue.
- **node** et **concept** pour construire de nouveaux termes dans la vue ;
- **"is a"** pour définir une relation structurelle simple ;
- **script** et **global** pour utiliser du code Java dans l'exécution des vues.

Dans cette section, nous présentons la sémantique de chacun de ces modifieurs dans l'observateur *exec*.

2.3.1.3 Opérateur de subsomption : \succeq

Dans les fonctions de ϕ_{exec} , nous utiliserons la relation de *subsomption* entre les termes, notée \preceq . Elle est définie récursivement par :

$\forall (x, y) \in \mathcal{T}^2$, $x \succeq y$ si et seulement si :

$$\begin{cases} \text{concept}(x) = \text{concept}(y) \\ \forall x' \in \text{fils}(x), \exists y' \in \text{fils}(y) \text{ tq } x' \succeq y' \end{cases}$$

C'est-à-dire que $x \succeq y$ si et seulement s'ils ont même concept et si l'ensemble des attributs de x est inclus, au sens de la subsomption, dans celui de y ($\forall x' \in \text{fils}(x), \exists y' \in \text{fils}(y) \text{ tq } x' \succeq y'$). *L'ordre des attributs n'est pas utilisé* dans l'opération de subsomption.

Par exemple :

$$\begin{aligned} a[b] &\succeq a[b, c, d] \\ a[b[c]] &\succeq a[d, b[c]] \\ a[b[c], d] &\succeq a[d, b[c, e]] \end{aligned}$$

Dans chacun de ces exemples, le premier terme est *plus général* que le second.

Si E est un ensemble de termes, nous notons $\min(E)$ le plus petit élément de E au sens de la subsomption.

Remarque

La définition de la subsomption que nous avons donnée ici a pour conséquence $a[b, b] \succeq a[b, c]$. Nous pourrions envisager dans des travaux futurs un autre opérateur *apparier* qui subsume un terme en n'utilisant pas deux fois le même sous-terme (ce problème de choix d'un bon opérateur de subsomption est très étudié en théorie des langages de programmation).

2.3.2 Références

Les modifieurs *path*, *get*, "*event get*" et "*size ref*" permettent de définir des références aux termes dans la vue, dans les actions. Leurs attributs sont vus comme un « chemin partiel » dans l'arbre de la vue pour accéder à un ou plusieurs termes. Nous dirons que la liste des attributs t_1, \dots, t_n d'un terme de concept *path*, *get*, "*event get*" ou "*size ref*" constitue une *référence* à un ou plusieurs termes t'_1, \dots, t'_n de la vue, appelés *résultat de la référence dans la vue*.

Une référence est une liste de termes (*i.e.* un multi-ensemble ordonné) définissant un chemin partiel dans la vue.⁵

Le résultat de la référence est défini par la fonction *calcRef*.

2.3.2.1 La fonction *calcRef*

Soit $\{t_1, \dots, t_n\} \in \mathcal{P}(\Upsilon)$ une référence et $T \in \Upsilon$ un terme. La fonction *calcRef* : $\mathcal{P}(\Upsilon) \times \Upsilon \rightarrow \mathcal{P}(\Upsilon)$ calcule le résultat de la référence $\{t_1, \dots, t_n\}$ dans T . Elle est définie récursivement de la manière suivante :

$$\begin{aligned} \text{calcRef}(\{t_1, \dots, t_n\}, T) = & \\ \left\{ \begin{array}{ll} \emptyset & \text{si } n = 0 \text{ ou } \text{concept}(T) \in I_{\text{calcRef}} \quad (\text{cas 1}) \\ \{T\} & \text{si } n = 1 \text{ et } t_1 \succeq T \quad (\text{cas 2}) \\ \bigcup_{t' \in \text{fils}(T)} \text{calcRef}(\{t_2, \dots, t_n\}, t') & \text{si } n > 1 \text{ et } t_1 \succeq T \quad (\text{cas 3}) \\ \bigcup_{t' \in \text{fils}(T)} \text{calcRef}(\{t_1, \dots, t_n\}, t') & \text{sinon} \quad (\text{cas 4}) \end{array} \right. \end{aligned}$$

où $I_{\text{calcRef}} = \{\text{add}, \text{put}, \text{del}, \text{guard}, \text{event}, \text{path}, \text{get}, \text{"event get"}, \text{"size ref"}\}$ est l'ensemble des modifieurs qui ne doivent pas être pris en compte dans le calcul de la référence.

⁵Il ne faut pas confondre les références en VDL qui permettent de désigner un ou plusieurs termes d'une vue et le *problème de la référence* en TALN [103], traité par le MLN, que nous aborderons au chapitre 3. Nous montrerons que le MLN doit résoudre le problème de la référence pour construire des références VDL utilisées dans les requêtes.

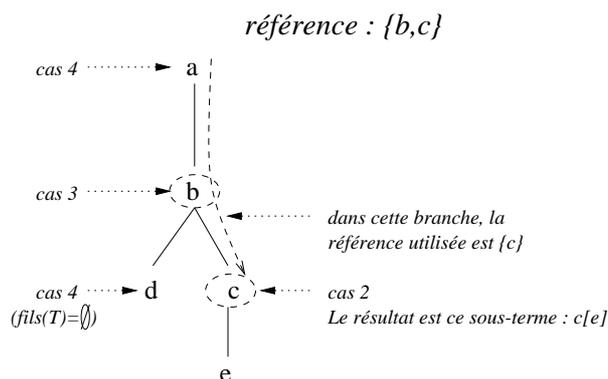
Remarques

1. La fonction récursive *calcRef* termine puisqu'elle « descend » dans l'arbre : la hauteur de T est strictement décroissante dans *calcRef*.
2. La fonction *calcRef* utilise l'ordre des attributs dans un terme. Ainsi, la référence $\{a, b\}$ ne désigne pas les mêmes termes que la référence $\{b, a\}$ (elles n'ont pas le même résultat).
3. Les références utilisées dans la fonction *calcRef* peuvent indifféremment être des termes flottants ou rattachés à une vue. En revanche, la fonction *calcRef* retourne uniquement des termes non-flottants, rattachés au terme passé en paramètre.

2.3.2.2 Exemples

Exemple 1

Considérons le terme $a[b[c, d[e]]]$. La référence $\{b, c\}$ a pour résultat le terme de concept c , comme le montre la figure ci-dessous :



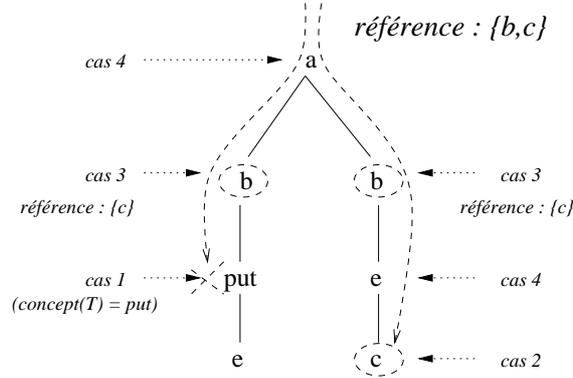
Le déroulement de l'algorithme de *calcRef* est le suivant :

1. Le concept du terme examiné (c'est-à-dire a) n'est pas le premier de la référence (qui est b) : nous sommes dans le cas 4. La fonction *calcRef* est donc appliquée aux attributs du terme, avec la même référence.
2. Il n'y a qu'un attribut (le terme de concept b). Ce terme est subsumé par le premier terme de la référence ($b \succeq b[c, d]$). Nous sommes alors dans le cas 3. La fonction *calcRef* est donc appliquée aux attributs du terme de concept b , mais avec la référence tronquée de son premier élément, c'est-à-dire $\{c\}$.
 - (a) Pour l'attribut de concept d , nous sommes dans le cas 4 avec $fils(T) = \emptyset$. Le résultat pour cette branche est donc \emptyset .
 - (b) Pour l'attribut de concept c , nous sommes dans le cas 2 (le terme est subsumé par le dernier élément de la référence). Le résultat de cette branche est donc le singleton composé du sous-terme atteint, *i.e.* $\{c[e]\}$.

Le résultat final est l'union des résultats pour toutes les branches, *i.e.* $\{c[e]\}$.

Exemple 2

Considérons maintenant un exemple un peu plus riche :



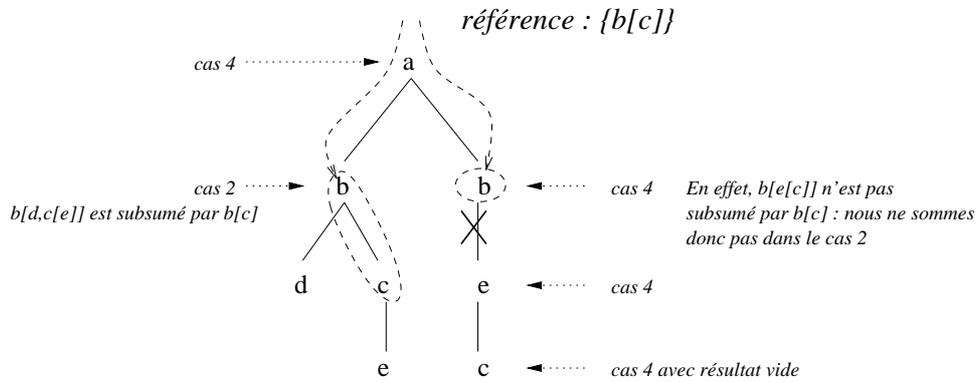
Dans cet exemple, la fonction *calcRef* est aussi appliquée aux attributs du terme de concept *a*. L'appariement se fait dans les deux branches et *calcRef* est alors appelée sur les attributs avec la référence « amputée » $\{c\}$.

- Dans la branche de gauche sur la figure, le concept tu terme est *put* qui appartient à $I_{calcRef}$. Nous sommes donc dans le cas 1 : le reste de la branche est ignorée et le résultat pour cette branche est \emptyset .
- Pour la branche de droite, l'algorithme est déroulé de manière similaire à l'exemple précédent, conduisant au résultat $\{c\}$.

Le résultat final est l'union des résultats des deux branches, *i.e.* $\{c\}$.

Exemple 3

Enfin, pour illustrer le mécanisme de subsumption, considérons l'exemple suivant :



- Dans la branche de gauche, le terme $b[d, c[e]]$ est bien subsumé par $b[c]$, qui est le premier (et le seul) élément de la référence. Nous sommes donc dans le cas 2, et le résultat est $\{b[d, c[e]]\}$.
- Dans la branche de droite, le terme $b[e[c]]$ n'est pas subsumé par $b[c]$. Nous sommes donc dans le cas 4 : la fonction *calcRef* est appliquée aux attributs du terme. Mais aucun sous-terme n'est subsumé par $b[c]$: le résultat pour la branche est donc \emptyset .

Le résultat de la référence $\{b[c]\}$ sur ce terme est donc $\{b[d, c[e]]\}$.

2.3.2.3 Construction de l'ensemble $I_{calcRef}$

L'ensemble $I_{calcRef}$ définit l'ensemble des termes qui ne sont pas pris en compte dans le calcul d'une référence. En effet, en VDL, les actions sont décrites comme des éléments de la vue. Nous sommes alors confrontés, comme en LISP, au théorème d'incomplétude de Gödel : lorsqu'un système est capable de faire référence à lui-même (nous parlons *d'auto-référence*), il peut permettre d'exprimer des éléments paradoxaux et il devient non calculables. L'exemple le plus célèbre est celui du crétois déclarant : « tous les crétois sont des menteurs ».

Dans un langage de programmation, comme VDL, nous ne voulons pas avoir d'élément non calculable. C'est la raison pour laquelle les références ne doivent pas pouvoir s'adresser elles-mêmes. Par conséquent, la fonction *calcRef* ne doit pas rechercher dans les termes dont le concept est un modifieur de référence : *path*, *get*, *"event get"*, *"size ref"*.⁶

De même, les attributs des actions décrivent les modifications à effectuer. Par exemple, nous pouvons avoir une action qui ajoute un terme i dans la vue. Ce terme i , décrit dans les actions, ne doit pas être pris en compte dans les références (de même qu'en LISP, les actions qui ne doivent pas être interprétées sont « quotées »). C'est la raison pour laquelle les termes représentant des actions élémentaires (*i.e.* ceux de concept *add*, *put* et *del*) sont ignorés par *calcRef*.

Enfin, les événements externes envoyés à la vue, décrits dans les attributs des termes de concept *event*, peuvent prendre n'importe quelle forme. C'est la raison pour laquelle ces termes sont ignorés par *calcRef*.

Remarque

Dans la version 0.1 de VDL [121], J.-P. Sansonnet utilise le modifieur *script* pour inhiber les éléments procéduraux et explique que cela revient à considérer les actions VDL comme des constantes, à la manière du *code quoté* LISP.

Ce modifieur *script* introduit un verrou *sémantique* de manière *syntaxique*, ce

⁶Nous pouvons illustrer ce problème de manière plus formelle directement en VDL : supposons que *get[i]* fasse référence à tous les termes de la vue dont le concept est i , y compris les références. Alors le i de *get[i]* est aussi référencé. S'il est utilisé dans une action (par exemple, augmenter la valeur de tous les termes de concept i), il se modifie lui-même.

qui est contraire à la notion d'observateur et de modifieur séparant syntaxe et sémantique que nous avons évoqué précédemment.

De plus, l'utilisation de verrous syntaxiques pose le problème suivant : si *script* est utilisé pour indiquer à la fonction *calcRef* qu'elle doit ignorer certains termes, il faut aussi un concept *script_x* pour les autres fonctions (la fonction d'interprétation, la fonction d'exécution des actions, la fonction de construction de l'interface, etc). Ce qui est verrouillé pour une fonction ne l'étant pas nécessairement pour une autre, nous ne pouvons pas regrouper ces verrous en un seul (par exemple, il ne faut pas faire référence aux références, mais il faut pouvoir les interpréter pour avoir un modèle Turing-complet). La multiplication des verrous rend le code impossible à écrire sans connaître en détail la sémantique opérationnelle du langage.

C'est pourquoi nous proposons ici une autre solution qui consiste à simplement attribuer une sémantique de verrou à chaque terme lorsque cela est nécessaire. Nous avons ainsi un verrou sémantique, qui n'apparaît pas de manière syntaxique.

2.3.3 Interprétation canonique

Dans notre modèle, les actions élémentaires permettent d'ajouter des termes dans la vue. Certains sous-termes de ces actions élémentaires décrivent les termes à ajouter, généralement construits à partir d'autres termes en utilisant les opérations arithmétiques.

L'interprétation des concepts VDL représentant des opérations arithmétique est définie par la fonction $\zeta : \Upsilon \times \mathcal{P}(\Upsilon) \longrightarrow \mathcal{P}(\Upsilon)$, appelée *interprétation canonique*.

NB : La fonction ζ retourne toujours des termes flottants. Nous verrons section 2.3.4 comment ceux-ci sont rattachés à la vue lors de l'exécution des actions élémentaires.

2.3.3.1 Définitions

Pour l'interprétation des modifieurs représentant des valeurs arithmétiques, nous définissons les deux ensembles suivants :

- l'ensemble \mathbb{R}_{vdl} des atomes correspondant aux nombres décimaux dans leur représentation usuelle (par exemple : -2.63). Si t est un terme, nous dirons que $t \in \mathbb{R}_{vdl}$ si et seulement si t n'a pas d'attribut et *concept*(t) est la représentation d'un nombre décimal (ou entier) sous sa forme canonique.
- l'ensemble $\mathbf{B}_{vdl} = \{true, false\}$ des atomes correspondant aux booléens (\top , \perp).

Soit alors $t = c[t_1, \dots, t_n]$ un terme de concept c et d'attributs t_1, \dots, t_n et E un ensemble de termes⁷ éventuellement vide.

⁷Nous verrons dans la suite du document que E est l'ensemble des événements en cours de traitement, manipulables par le modifieur "*event get*". Cet ensemble est construit par la fonction d'exécution des actions élémentaires.

L'interprétation $\zeta(t, E)$ du terme t est définie récursivement par :

2.3.3.2 Termes procéduraux

- Si $c \in I_\zeta = \{add, put, del, guard, event\}$, $\zeta(t, E) = \{t\}$.

En effet, comme en LISP, il est possible d'ajouter dans la vue un terme qui décrit un fonctionnement. En LISP, ces actions doivent être « quotées » pour ne pas être exécutées et conservées sous la forme d'une liste. En VDL, elle ne sont simplement pas interprétées par la fonction ζ (ce qui revient à les conserver sous la forme initiale).

Le résultat de la fonction ζ est alors le terme flottant t initialement passé en paramètre.

2.3.3.3 Termes utilisant des références

- Si $c = get$, $\zeta(t, E) = \cup fils(t')$ pour $t' \in calcRef((\bigcup_{i=1}^n \zeta(t_i, E)), vue(t))$

Autrement dit, l'interprétation d'un terme de concept *get* est l'ensemble des *attributs* du ou des terme(s) référencé(s) par l'interprétation des attributs du *get*. Par exemple, considérons la vue suivante :

```
view[ 2[a], get[ plus[1,1] ] ]
```

Dans cette vue, la référence contient le terme *plus[1,1]*, qui sera interprété par ζ en 2. Le résultat de cette référence est donc l'attribut du terme de concept 2, c'est-à-dire l'atome *a*.

Le résultat d'un *get* est alors un ensemble de termes *flottants*, copies conformes des termes rattachés à la vue obtenus comme résultats de la référence. Dans notre exemple, la fonction ζ retourne un terme flottant *a*, et non le terme *a* rattaché à la vue (ce sont deux termes différents).

- Si $c = "event\ get"$,
 - si $n > 0$, $\zeta(t, E) = \cup fils(t')$ pour $t' \in \bigcup_{e \in E} calcRef((\bigcup_{i=1}^n \zeta(t_i, E)), e)$.
 - si $n = 0$, $\zeta(t, E) = E$.

L'interprétation d'un terme de concept "*event get*" est l'ensemble des *attributs* des termes référencés par l'interprétation des attributs du "*event get*", dans les événements en cours de traitement, donnés par l'ensemble *E*. Si la référence est vide, le résultat est l'ensemble des événements externes en cours de traitement. Comme pour *get*, le résultat d'un "*event get*" est un ensemble de termes *flottants*, copies conformes des termes résultats de la référence.

- Si $c = "size\ ref"$, soit $s = |calcRef((\bigcup_{i=1}^n \zeta(t_i, E)), vue(t))|$ et s_{vdl} l'atome de \mathbb{R}_{vdl} lui correspondant. Alors $\zeta(t, E) = \{s_{vdl}\}$.

Autrement dit, l'interprétation d'un terme de concept "*size ref*" est le nombre de termes référencés par l'interprétation des attributs du "*size ref*".

2.3.3.4 Termes arithmétiques

- Si $c = plus$,
 - si $\forall i, \zeta(t_i, E) \in \mathbb{R}_{vdl}$, soit n_i l'interprétation canonique de $\zeta(t_i, E)$ dans \mathbb{R} , soit $res = n_1 + \dots + n_n$ et res_{vdl} l'atome de \mathbb{R}_{vdl} lui correspondant. Alors $\zeta(plus[t_1, \dots, t_n], E) = \{res_{vdl}\}$.
 - Sinon, l'algorithme retombe dans le cas général (voir ci-après).
Autrement dit, l'interprétation canonique d'un terme de concept *plus* est la somme des interprétations canoniques des ses attributs, dans la mesure où cela a un sens du point de vue mathématique. Par exemple, quelque soit $E \in \mathcal{P}(\Upsilon)$, $\zeta(plus[1, 2], E) = \{3\}$.
- Si $c = times$,
 - si $\forall i, \zeta(t_i, E) \in \mathbb{R}_{vdl}$, soit n_i l'interprétation canonique de $\zeta(t_i, E)$ dans \mathbb{R} , soit $res = n_1 \times \dots \times n_n$ et res_{vdl} l'atome de \mathbb{R}_{vdl} lui correspondant. Alors $\zeta(times[t_1, \dots, t_n], E) = \{res_{vdl}\}$.
 - Sinon, l'algorithme retombe dans le cas général.
Autrement dit, même chose avec \times au lieu de $+$.
- Si $c = or$,
 - si $\forall i, \zeta(t_i, E) \in \mathbf{B}_{vdl}$, soit b_i l'interprétation canonique de $\zeta(t_i, E)$ dans \mathbf{B} , soit $res = b_1 \vee \dots \vee b_n$ et res_{vdl} l'atome de \mathbf{B}_{vdl} lui correspondant. Alors $\zeta(or[t_1, \dots, t_n], E) = \{res_{vdl}\}$.
 - Sinon, l'algorithme retombe dans le cas général.
Autrement dit, l'interprétation canonique d'un terme de concept *or* est la disjonction des interprétations canoniques des ses attributs, dans la mesure où cela a un sens du point de vue logique. Par exemple, quelque soit $E \in \mathcal{P}(\Upsilon)$, $\zeta(or[true, false], E) = \{true\}$.
- Si $c = and$,
 - si $\forall i, \zeta(t_i, E) \in \mathbf{B}_{vdl}$, soit b_i l'interprétation canonique de $\zeta(t_i, E)$ dans \mathbf{B} , soit $res = b_1 \wedge \dots \wedge b_n$ et res_{vdl} l'atome de \mathbf{B}_{vdl} lui correspondant. Alors $\zeta(and[t_1, \dots, t_n], E) = \{res_{vdl}\}$.
 - Sinon, l'algorithme retombe dans le cas général.
Autrement dit, même chose avec \wedge au lieu de \vee .
- Si $c = minus$,
 - si $n = 1$ et $\zeta(t_1, E) \in \mathbb{R}_{vdl}$, soit n_1 l'interprétation canonique de $\zeta(t_1, E)$ dans \mathbb{R} , et res_{vdl} l'atome de \mathbb{R}_{vdl} correspondant à $-n_1$. Alors $\zeta(minus[t_1, \dots, t_n], E) = \{res_{vdl}\}$.

- Sinon, l’algorithme retombe dans le cas général.
Autrement dit, l’interprétation canonique d’un terme de concept *minus* est l’opposé de l’interprétation de son attribut, s’il est unique et dans la mesure où cela a un sens du point de vue mathématique.
- Si $c = inverse$,
 - si $n = 1$ et $\zeta(t_1, E) \in \mathbb{R}_{vdl}$, soit n_1 l’interprétation canonique de $\zeta(t_1, E)$ dans \mathbb{R} . Si $n_1 \neq 0$, soit res_{vdl} l’atome de \mathbb{R}_{vdl} correspondant à n_1^{-1} .
Alors $\zeta(inverse[t_1, \dots, t_n], E) = \{res_{vdl}\}$.
 - Sinon, l’algorithme retombe dans le cas général.
Autrement dit, l’interprétation canonique d’un terme de concept *inverse* est l’inverse de l’interprétation de son attribut, s’il est unique et dans la mesure où cela a un sens du point de vue mathématique.
- Si $c = not$,
 - si $n = 1$ et $\zeta(t_1, E) \in \mathbf{B}_{vdl}$, soit b_1 l’interprétation canonique de $\zeta(t_1, E)$ dans \mathbf{B} , et res_{vdl} l’atome de \mathbf{B}_{vdl} correspondant à $\neg b_1$.
Alors $\zeta(minus[t_1, \dots, t_n], E) = \{res_{vdl}\}$.
 - Sinon, l’algorithme retombe dans le cas général.
Autrement dit, l’interprétation canonique d’un terme de concept *not* est la négation de l’interprétation de son attribut, s’il est unique et dans la mesure où cela a un sens du point de vue logique.
- Si $c = equals$,

$\zeta(equals[t_1, \dots, t_n], E) = \{true\}$ si $\forall t_j \in \bigcup_{i=1}^n \zeta(t_i, E), t'_j = t'_1$ et $\{false\}$ sinon ou si $card(\bigcup_{i=1}^n \zeta(t_i, E)) < 2$.

Autrement dit, l’interprétation canonique d’un terme de concept *equals* est un singleton contenant le terme flottant *true* si tous les termes résultats de l’interprétation des attributs du *equals* sont égaux, et *false* sinon.
- Si $c = "is a"$,

$\zeta(is a[t_1, \dots, t_n], E) = \{true\}$ si $\forall t_j \in \bigcup_{i=1}^n \zeta(t_i, E), concept(t'_j) = concept(t'_1)$ et $\{false\}$ sinon ou si $card(\bigcup_{i=1}^n \zeta(t_i, E)) < 2$.

Autrement dit, l’interprétation canonique d’un terme de concept *"is a"* est un singleton contenant le terme flottant *true* si les concepts de tous les termes résultats de l’interprétation des attributs du *"is a"* sont égaux, et *false* sinon. La relation de spécialisation « *is a* » est en réalité beaucoup plus complexe dans les versions typées et structurées de VDL. Il ne s’agit ici que d’un artefact pour modéliser une notion, nécessaire pour la définition de certains composants, dans le cadre structurel restreint qu’est VDLp.

- Si $c = \text{"greater than"}$,
 - si $n = 2$ et $\forall i \in [1, 2], \zeta(t_i, E) \in \mathbb{R}_{vdl}$, soit n_i l'interprétation canonique de $\zeta(t_i, E)$ dans \mathbb{R} .
Alors $\zeta(\text{"greater than"}[t_1, t_2], E) = \{true\}$ si $n_1 \geq n_2$ et $\{false\}$ sinon.
 - Sinon, l'algorithme retombe dans le cas général.
- Par exemple, quelque soit $E \in \mathcal{P}(\Upsilon)$, $\zeta(\text{"greater than"}[2, 1], E) = \{true\}$. L'interprétation des termes de concept "greater than" utilise l'ordre des attributs. Ainsi, "greater than"[1, 2] ne sera pas interprété de la même manière que "greater than"[2, 1].

2.3.3.5 Modifieur *random*

- Si $c = \text{random}$, soit x un nombre tiré aléatoirement dans $[0, 1]_{\mathbb{R}}$ et res_{vdl} l'atome correspondant à x . Alors $\zeta(\text{random}[t_1, \dots, t_n], E) = \{res_{vdl}\}$

Autrement dit, l'interprétation canonique d'un terme de concept *random* est la représentation en VDL d'un nombre entier compris entre 0 et 1, tiré aléatoirement.

2.3.3.6 Modifieurs structurels

- Si $c = \text{size}$,
 - si $n \geq 1$, soit $s = |\bigcup_{i=1}^n \zeta(t_i, E)|$ et s_{vdl} l'atome de \mathbb{R}_{vdl} lui correspondant.
Alors $\zeta(\text{size}[t_1, \dots, t_n], E) = \{s_{vdl}\}$.
 - Sinon, l'algorithme retombe dans le cas général.

Autrement dit, l'interprétation canonique d'un terme de concept *size* est la représentation en VDL du nombre de ses attributs, après interprétation, à condition qu'il y en ait au moins un. Par exemple, pour tout $E \in \mathcal{P}(\Upsilon)$:

$$\begin{aligned} \zeta(\text{size}[a, b, c], E) &= 3 \\ \zeta(\text{size}[\text{plus}[1, 1]], E) &= |\{1\}| = 1 \\ \zeta(\text{size}, E) &= \text{size} \text{ (et non } 0) \end{aligned}$$

En effet, l'interprète dynamique de VDLp considère que *size* a un sens s'il possède au moins un attribut. Sinon, il ignore ce modifieur, qui peut être utilisé par exemple dans une référence pour parler de la « taille » d'une figure, comme nous le verrons dans la section 2.4.

- Si $c = \text{node}$,

$$\zeta(\text{node}[t_1, \dots, t_n], E) = \begin{cases} \emptyset & \text{si } n = 0 \\ \text{node}[I] & \text{si } C = \emptyset \\ \bigodot_{t_i \in C} f_{text}(t_i)[I \setminus C] & \text{sinon} \end{cases}$$

où $I = \bigcup_{t_i} \zeta(t_i, E)$, $C = \{t'_i \in I \text{ tq } \text{concept}(t'_i) = \text{concept}\}$, \odot est la concaténation de chaînes sur A^* et f_{text} est la fonction de construction d'une représentation textuelle d'un terme VDL, définie section 2.3.3.9.

Autrement dit, l'interprétation canonique d'un terme de concept *node* est un terme :

- Dont le concept est construit à partir de l'interprétation des attributs t_i de concept *concept*, en utilisant la fonction f_{text} ;
- Dont les attributs sont exactement les attributs (interprétés) dont le concept n'est pas *concept*.

L'utilisation du modifieur *concept*, qui alourdit l'écriture du code VDL et des algorithmes d'interprétation, est nécessaire en raison de l'aspect non ordonné des nœuds de l'arbre.

2.3.3.7 Cas général

- Dans tous les autres cas (*i.e.* si l'interprétation de t n'est pas définie par l'une des règles précédentes), nous avons :

$$\zeta(t, E) = \{c[\zeta(t_1, E), \dots, \zeta(t_n, E)]\}$$

Par exemple, quelque soit $E \in \mathcal{P}(\Upsilon)$, $\zeta(\text{plus}[a, 2], E) = \{\text{plus}[a, 2]\}$, l'atome a ne pouvant pas être interprété comme un élément de \mathbb{R} .

Ainsi, dans le cas général, le résultat est le singleton composé du terme dont les attributs ont été à leur tour interprétés, en utilisant la convention :

$$c[t_1, \dots, t_n, \{t_{n+1}, \dots, t_m\}] \equiv_{def} c[t_1, \dots, t_m]$$

2.3.3.8 Construction de l'ensemble I_ζ

L'ensemble $I_\zeta = \{\text{add}, \text{put}, \text{del}, \text{guard}, \text{event}\}$ définit l'ensemble des termes qui ne sont pas interprétés par la fonction ζ . En effet, ces termes sont exactement les termes procéduraux définissant des actions (actions élémentaires ou actions composées). Par conséquent, ils ne doivent être interprétés que lorsque la fonction d'exécution les atteint, et conformément aux algorithmes d'exécution présentés section 2.3.4.

Remarquons que l'introduction de ce verrou sémantique \mathcal{I}_ζ permet de définir des actions dont l'effet est d'ajouter ou de retirer des actions de la vue, c'est-à-dire de modifier le fonctionnement du composant actif (la description de ces modifications, c'est-à-dire les actions à ajouter n'étant alors pas interprétées, puisque qu'elle ne sont pas atteintes par l'algorithme d'exécution).

2.3.3.9 Fonction f_{text}

Nous notons \odot la concaténation de chaînes.

Tout terme VDL $t \in \Upsilon$ est associé à une représentation textuelle par la fonction $f_{text} : \Upsilon \longrightarrow A^*$ définie de la manière suivante :

$$f_{text}(t) = \begin{cases} \text{concept}(t) & \text{si } \text{fils}(t) = \emptyset \\ \text{concept}(t) \odot \text{""} \odot \bigodot_{t_i \in \text{fils}(t)} f_{text}(t_i) \odot \text{""} \odot \text{""} & \text{sinon} \end{cases}$$

2.3.4 Exécution des actions

2.3.4.1 Principe général

Nous définissons en VDL trois *actions élémentaires* permettant de modifier la structure de la vue, chacune associée à un modifieur :

- *add* pour ajouter un terme comme attribut d'un autre.
- *put* pour remplacer les attributs d'un terme par un nouveau terme.
- *del* pour effacer un terme

Nous noterons $Act = \{add, put, del\} \subset A^*$ l'ensemble des concepts des actions élémentaires. Nous verrons section 2.3.6.3 que chacun de ces trois opérations sur le corps des vues ne peut pas s'écrire comme combinaison des deux autres (elles forment une base des modifications possibles d'un arbre VDL).

Soit a un terme représentant une action élémentaire. L'ensemble des termes modifiés par l'action élémentaire est défini par l'ensemble P des éléments référencés par les attributs de concept *path* de a ⁸. Si $\text{concept}(a) \in \{add, put\}$, l'ensemble des « nouveaux attributs » des termes modifiés est défini par le complémentaire de P dans l'ensemble des attributs de a , c'est-à-dire par l'ensemble des attributs dont le concept n'est pas *path*.

Chaque action peut être munie de deux sortes de préconditions :

- Des *gardes* booléennes, englobées dans un concept *guard* :
Les actions sont exécutées si et seulement si les attributs du terme de concept *guard* doivent être interprétés à $\{true\}$ par la fonction ζ (nous dirons que la garde est *valide*) ;
- Des *événements externes*, englobés dans un concept *event* :
Les actions sont exécutées si et seulement si la vue a reçu au cours du cycle précédent un événement externe appariable (par subsomption) avec l'un des attributs du terme de concept *event*.

La fonction d'exécution des actions ne parcourt un terme donné t que si **toutes** les préconditions (représentées par des termes de concept *guard* ou *event*) parmi ses attributs sont vérifiées.

⁸Par analogie avec les langages de programmation classiques, *get* est un accès en lecture à un terme alors que *path* décrit un accès en écriture (dans les actions). Par exemple, pour modéliser une instruction $dest := src$ en langage Pascal, nous utiliserons en VDL l'instruction : $put[path[dest], get[src]]$.

File des événements

Tous les événements externes envoyés au composant au cours de l'exécution sont mémorisés dans une liste de termes $\varepsilon \in \mathcal{P}(\Upsilon)$. A chaque instant de l'exécution, la liste ε contient l'ensemble des événements externes envoyés à la vue au cours du cycle d'exécution immédiatement précédent. La file des événements externe est vidée à la fin du cycle d'exécution, même si certains événements n'ont pas pu être traités par le composant. Ce principe est illustré sur la figure 2.4.

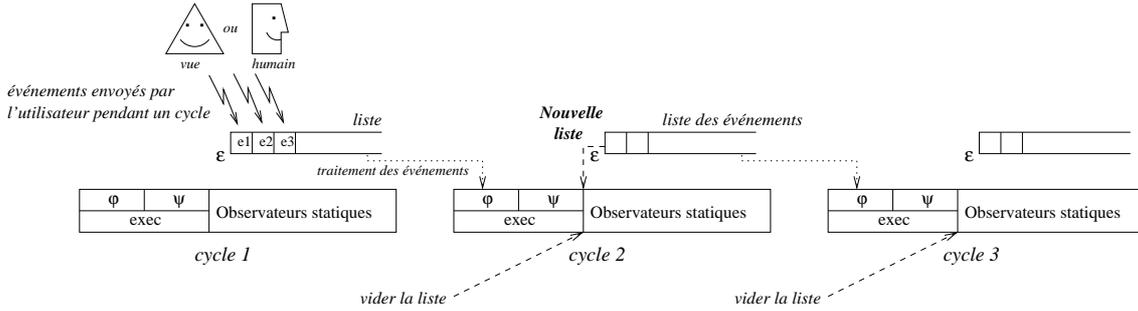


FIG. 2.4 – Gestion de la liste des événements

Le mécanisme d'interaction avec le monde extérieur à l'aide d'événements externes est fortement dépendant de l'implémentation des vues. Il ne sera pas présenté ici.

2.3.4.2 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** VDL ;
- Si a est une action VDL, nous dirons que $concept(a)$ est le **nom** de l'action.⁹
- 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 « frères », « oncles », « grand-oncles », *etc.* de a :

$$\pi(a) = \{t \text{ tq } pere(a) = pere(t)\} \cup \pi(pere(a))$$

Nomenclature

- Le **comportement** d'un composant est défini par l'ensemble des actions ;
- L'**activité** d'un composant est l'ensemble de ses actions dont les préconditions sont vérifiées ;

⁹L'utilisation du *nom* de l'action sera au coeur de nos algorithmes de construction de réponses aux requêtes, dans le MRF, présentés au chapitre 4.

- L' **état** du composant est donné par la vue (*i.e.* l'arbre de concepts) à l'instant t .¹⁰

Classification des actions

Nous pouvons alors classer les actions VDL en fonction de leurs préconditions et, à un instant donné, de la valeur de ces préconditions. Nous dirons que :

- Toute action a telle que $\forall t \in \Pi_a, \text{concept}(t) = \text{guard}$ est un **processus**. Une action sans précondition est donc un processus. Un processus est **actif** ssi toutes ses gardes sont vérifiées et *inactif* sinon.

Les processus expriment la notion de changement continu : les actions sont toujours exécutées tant que la garde est vraie. Ils sont donc effectués de façon *non modale* : les actions des processus ne dépendent pas de l'interaction avec l'utilisateur.

- Toute action a telle que $\exists t \in \Pi_a, \text{concept}(t) = \text{event}$ est une **réaction**. Une réaction est **possible** ssi toutes ses gardes sont vérifiées. Dans le cas contraire, *même si l'événement externe a été reçu*, elle ne sera pas exécutée.

Les réactions aux événements externes permettent d'introduire la notion d'interaction entre la vue et le monde extérieur. Elles sont donc effectuées de façon *modale* : les actions sont effectuées uniquement lorsque la vue reçoit un événement.

Remarquons que, suivant cette définition, une action qui n'est munie d'aucune précondition est un processus et qu'elle est toujours effectuée. Nous parlerons de *processus non gardé*.

2.3.4.3 Notion de variable

Compte tenu de leur structure arborescente, les vues n'utilisent pas de notion de variable. Cependant, l'utilisation de variable est un élément *nécessaire* dès lors que l'on souhaite *programmer* un composant.

En VDLp, il est possible de modéliser des variables au sens classique du terme. Nous verrons section 2.3.6.1 qu'il est même possible de modéliser des variables structurées, au sens où l'entendent les langages de programmation classiques.

Définition

Une *variable* VDL est un terme contenant un attribut atomique dans $\mathbf{R}_{vdl} \cup \mathbf{B}_{vdl}$. Nous appelons *valeur* cet attribut.

Par rapport à la définition classique d'une variable dans un langage de programmation, seules les variables (au sens classique du terme) dont le type est *nombre* ou *booléen* sont des variables VDL. Par exemple, une cellule d'un tableau est une variable, mais un tableau n'est pas une variable VDL.

¹⁰Il correspond donc à la notion de *situation* dans la terminologie du calcul des situations [112].

Les variables VDL sont en fait les *variables terminales* des langages de programmation classiques.

2.3.4.4 Notion de concurrence

A chaque cycle d'exécution, la fonction ϕ_{exec} parcourt la vue pour y rechercher les actions élémentaires dont les préconditions sont vérifiées. *Toutes* ces actions élémentaires sont alors effectuées :

Si a est une action élémentaire dans la vue v telle que toutes les préconditions $p \in \Pi_a$ sont vérifiées, alors a sera *effectuée* par $\phi_{exec}(v)$.

Soit $P(a) = \{t' \in fils(a) \cdot concept(t') = path\}$ l'ensemble des attributs de concept $path$. Soit $T(a)$ l'ensemble des termes qui seront modifiés par a :

$$T(a) = \{t \in v \cdot \exists t' \in P(a) \cdot calcRef(fils(t'), v) = t\}$$

Pour tout $t \in T(a)$, nous dirons que a sera *exécuté sur* t .

Plusieurs actions élémentaires peuvent donc être sélectionnées pour être effectuées sur un même terme t pendant le même cycle d'exécution. Il faut donc traiter un problème d'actions concurrentes.

Contraintes

Lors de l'exécution des actions élémentaires, nous voulons que :

1. Le résultat ne dépende pas de l'ordre dans lequel ces actions sont trouvées dans la vue par la fonction ϕ_{exec} ;
 - Pour les actions concurrentes, cela signifie qu'il faut un mécanisme déterministe de sélection des actions effectuées sur un terme t donné¹¹ ;
 - Dans le cas général, cela signifie que les attributs ajoutés à un terme par les actions *add* et *put* ne dépendent pas du parcours, c'est-à-dire qu'ils sont calculés de manière déterministe une fois que l'ensemble des actions à effectuer est connu.
2. La sémantique opérationnelle et le mécanisme de gestion de la concurrence soient conformes à la sémantique explicative des actions, afin de pouvoir répondre à des requêtes d'explications construites par le MLN :
 - une action *del* doit permettre de supprimer un terme de la vue ;
 - une action *add* doit permettre d'ajouter un terme dans la vue ;
 - une action *put* doit permettre de remplacer les attributs d'un terme par d'autres attributs.¹²

¹¹ Contrairement aux EA [54] où la modification d'un *lieu* donné obéit à la règle du « non-déterminisme local ».

¹² Soulignons que, suivant cette définition, *put* permet de modéliser l'instruction de base des langages de programmation : $x := y$, où x et y sont des variables terminales (*i.e.* des variables VDL).

3. Le plus d'action élémentaires possibles (parmi celles dont les préconditions sont vérifiées) soient effectuées à chaque cycle, tout en satisfaisant aux points précédents.

En particulier, l'une des conséquences de la seconde contrainte est que les actions de suppression (*del*) sur un terme donné t sont prioritaires sur les autres actions élémentaires, en cas de concurrence. En effet, pour rester conforme à sa sémantique explicationnelle, l'action *del* **doit** supprimer le terme. Cette suppression ne n'influe pas sur la sémantique explicationnelle des autres actions *add* et *put* puisque si un terme n'existe pas, rien ne *peut* lui être ajouté, conformément au bon sens.

La principale difficulté est alors de proposer un mécanisme qui prenne en considération le fait qu'un terme t peut être supprimé non pas en tant que tel, mais comme sous-terme d'un terme t' supprimé par une action *del*.

2.3.4.5 La liste des modifications

Pour gérer le problème des actions concurrentes et en particulier pour satisfaire la première contrainte ci-dessus, nous proposons d'utiliser un mécanisme d'exécution en deux temps :

1. Nous constituons dans un premier temps une liste μ des actions à effectuer. Dans cette liste, chaque action est mémorisée sous la forme d'un quadruplet $(t, a, R, E) \in \Upsilon \times Act \times \mathcal{P}(\Upsilon) \times \mathcal{P}(\Upsilon)$ où :
 - $t \in \Upsilon$ est le terme *non flottant*¹³ devant être modifié par l'action ;
 - $a \in Act$ est le concept de l'action ;
 - R est l'ensemble des attributs de l'action dont le concept n'est pas *path*, c'est-à-dire les termes qu'il faudra interpréter pour les ajouter à t lors de l'exécution réelle de l'action (lorsque $a = del$, $R = \emptyset$) : R est un donc ensemble de termes non flottants ;
 - E est l'ensemble des événements en cours de traitement pour cette action.
Nous notons $\varphi : \Upsilon \times \mathcal{P}(\Upsilon) \rightarrow \mathcal{P}(\Upsilon \times Act \times \mathcal{P}(\Upsilon) \times \mathcal{P}(\Upsilon))$ la fonction qui construit la liste μ à partir d'une vue v et d'une liste d'événements externes E .

Pour des raisons algorithmiques, la liste μ est indexée sur les $a_i.t$ des actions $a_i \in \mu$.

2. Nous effectuons les actions dans un ordre déterministe qui satisfait aux 3 contraintes énoncées et permet de gérer les actions concurrentes. Nous notons $\psi : \Lambda \times \mathcal{P}(\Upsilon \times Act \times \mathcal{P}(\Upsilon) \times \mathcal{P}(\Upsilon)) \rightarrow \Lambda$ la fonction qui effectue les actions, c'est-à-dire qui construit la vue v_{t+1} à partir de la vue v_t et de la liste des actions μ pour cette vue.

Ainsi, $\phi_{exec}(v) = \psi(v, \varphi(v, \emptyset))$ (nous dirons abusivement que $\phi_{exec} = \psi \circ \varphi$) :

$$v_t \xrightarrow{\varphi} \left\{ \begin{array}{c} v_t \\ \mu \end{array} \right\} \xrightarrow{\psi} v_{t+1}$$

¹³Nous verrons que, dans l'implémentation, t est un pointeur vers un sous-terme de la vue.

Ce mécanisme d'exécution est illustré sur la figure 2.5.

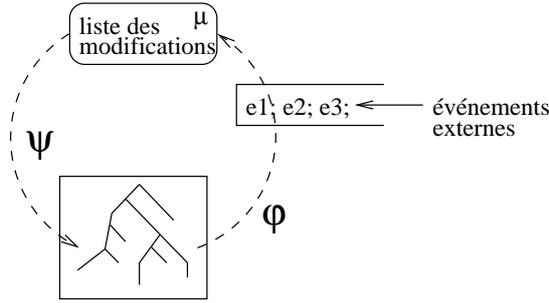


FIG. 2.5 – Mécanisme d'exécution en deux temps

2.3.4.6 Gestion de la concurrence dans le cas direct

Définition

Deux actions a_1 et a_2 de μ sont en concurrence si et seulement si $a_1.t = a_2.t$, c'est-à-dire si elles agissent sur le même terme de la vue.

Étude de cas

Soit t un terme et $(a_i)_{i \in [1, n]}$ les actions en concurrence pour la modification de t . Nous pouvons envisager 5 cas distincts :

1. $\forall i \in [1, n], a_i.act = add$: toutes les actions en concurrence sont des *add*.
Il n'y a alors pas à proprement parler de « problème » de concurrence puisque l'ordre dans lequel les attributs sont ajoutés dans la vue n'intervient pas. Il suffit donc « d'effectuer » toutes les actions a_i dans un ordre quelconque.
2. $\forall i \in [1, n], a_i.act = del$: toutes les actions en concurrence sont des *del*.
Il n'y a alors pas à proprement parler de « problème » de concurrence puisque toutes les actions ont exactement le même effet : supprimer t . Il suffit d'effectuer l'une des actions a_i .
3. $\exists i \in [1, n] tq a_i.act \neq del$ et $\exists i \in [1, n] tq a_i.act = del$: des actions *del* entrent en concurrence avec d'autres actions.
Comme nous l'avons montré section 2.3.4.4, les actions de suppression (*del*) sur un terme donné t sont prioritaires sur les autres actions élémentaires. Par conséquent, nous devons simplement de supprimer le terme.
4. $\forall i \in [1, n], a_i.act = put$: toutes les actions en concurrence sont des *put*.
5. $\forall i \in [1, n], a_i.act \in \{add, put\}$ et $\exists (i, j) \in [1, n]^2 tq a_i.act = add$ et $a_j.act = put$: des actions *add* et *put* entrent en concurrence entre elles.

Le problème de gestion de la concurrence apparaît uniquement dans ces deux derniers cas, *i.e.* lorsque des actions *put* sont en concurrence entre elles ou avec des actions *add*.

	add	put	del
add	1 	5 ?	3 
put	5 ?	4 ?	3 
del	3 	3 	2 

FIG. 2.6 – Différents cas de concurrences directes sur un terme

Traitement du cas 5

Pour être conforme à la sémantique explicationnelle, nous devons construire un nouveau terme contenant tous les attributs ajoutés par les actions *add*. Mais nous ne pouvons pas obtenir un terme qui soit conforme à la sémantique opérationnelle de plusieurs actions *put*, puisque chacune « détruit » ce qu'a fait la précédente en supprimant l'ensemble des attributs (c'est un problème de *sémaphore*).

Nous utiliserons donc la règle suivante :

- Les attributs des actions sont calculés sur la vue initiale v_t ;
- Une seule action *put* est effectuée, choisie de manière non déterministe (non-déterminisme local) ;
- Les actions *add* sont effectuées après l'action *put*. Les attributs qu'ils doivent ajouter sont calculés en utilisant la vue v_t .

Par conséquent, un programme VDL déterministe ne devrait pas contenir des actions *put* concurrentes entre elles et avec des *add*.

Utilisation de la notion de variable

Cependant, nous pouvons (et nous verrons dans le chapitre 5 que c'est même une nécessité) mettre à profit la possibilité pour une action *put* de représenter une affectation de variables, comme nous l'avons évoqué.

En effet, si un terme est modifié à la fois par des actions *add* et par des actions *put* (cas 5), ce ne peut pas être une variable, puisqu'après exécution des actions *add*, ce terme aurait plusieurs attributs. Mais lorsque seules des actions *put* interviennent (cas 4), nous pouvons considérer qu'il s'agit d'instructions $x := y$. Pour rester conforme à la sémantique explicationnelle des actions élémentaires, nous pouvons alors effectuer **toutes** ces « instruction », **en séquence**, dans un ordre déterministe.

Nous prendrons l'ordre lexicographique sur les termes VDL écrits dans leur représentation textuelle sans espace pour déterminer l'ordre d'exécution des *put*. Les attributs ajoutés par chaque *put* sont calculés non pas sur la vue initiale mais sur le résultat de l'action *put* précédente :

$$v_t : t_0 \xrightarrow{put_1} t_1 \xrightarrow{put_2} t_2 \xrightarrow{\dots} t_{n-1} \xrightarrow{put_n} t_n : v_{t+1}$$

2.3.4.7 Fonction φ

La fonction récursive φ prend en entrée un terme $t = c[t_1, \dots, t_n]$ (initialement la vue v_t) et un ensemble d'événements E en cours de traitement (initialement \emptyset). Elle retourne une liste de modifications μ . Elle est définie de la manière suivante :

Cas général

- Dans le cas général, $\varphi(c[t_1, \dots, t_n], E) = \bigcup_i \varphi(t_i, E)$:

L'algorithme « descend » dans la structure.

Actions élémentaires

- Si $c = add$ ou $c = put$,

soit $P = \{t_i \text{ tq } concept(t_i) = path\}$ et $\bar{P} = \{t_i \text{ tq } concept(t_i) \neq path\}$.¹⁴ Pour chaque $t_j \in P$, soit $ref(t_j) = calcRef(\left(\bigcup_{t' \in fils(t_j)} \zeta(t', E)\right), v_t)$ le résultat de la référence que constituent les attributs de t_j . Soit alors $Ref = \bigcup_{t \in ch} ref(t)$, c'est-à-dire l'ensemble des termes référencés par les attributs de t de concept $path$.¹⁵

Alors :

$$\begin{aligned}\varphi(add[t_1, \dots, t_n]) &= \{(t, add, \bar{P}, E), t \in Ref\} \\ \varphi(put[t_1, \dots, t_n]) &= \{(t, put, \bar{P}, E), t \in Ref\}\end{aligned}$$

- Si $c = del$,

de la même manière :

$$Ref = \bigcup_{t_i | concept(t_i) = path} \left(calcRef\left(\left(\bigcup_{t' \in fils(t_j)} \zeta(t', E)\right), vue(t)\right) \right)$$

Alors :

$$\varphi(del[t_1, \dots, t_n]) = \{(t, del, \emptyset, E), t \in Ref\}$$

L'ensemble de termes à ajouter est alors l'ensemble vide : il sera ignoré par la fonction ψ .

¹⁴ P est l'ensemble des attributs de concept $path$, dont les attributs constituent une *référence* ayant pour résultat les termes à modifier et \bar{P} est l'ensemble des autres attributs, qu'il va falloir ajouter.

¹⁵Les éléments de R et de \bar{P} sont des termes (non flottants) de la vue v_t .

Préconditions

- Si $c = guard$,

$$\varphi(c[guard[t_1, \dots, t_n], t_{n+1}, \dots, t_m], E) =$$

$$\begin{cases} si \forall t' \in \bigcup_{i=1}^n \zeta(t_i, E), t' = true, & \bigcup_{i=n+1}^m \varphi(t_i, E) \\ sinon & \emptyset \end{cases}$$

Si $n = 0$, la garde **est vérifiée** : les termes sont interprétés comme s'il n'y avait pas de garde (\top (*true*) étant élément neutre pour \vee).

- Si $c = event$,

$$\varphi(c[event[t_1, \dots, t_n], t_{n+1}, \dots, t_m], E) =$$

$$\begin{cases} si \exists t' \in \bigcup_{i=1}^n \zeta(t_i, E) \text{ et } t'' \in \varepsilon \text{ tq } t' \succeq t'', & \bigcup_{i=n+1}^m \varphi(t_i, E \cup \{t''\}) \\ sinon & \emptyset \end{cases}$$

Si $n = 0$, les termes **ne sont pas interprétés** (il n'existe pas de t_i tel que...). Soulignons que le ou les événements externes t'' correspondant à la précondition sont ajoutés à la liste des événements en cours de traitement ($E \cup \{t''\}$) pour cette branche de l'arbre.

Exemple abstrait

Considérons la vue suivante :

```
view[ guard[g1], a1, event[e], foo[ guard[g2], a2 ]
```

Les actions a_1 et foo sont effectuées si et seulement si $\zeta(g, \emptyset) = \{true\}$ et il existe $e' \in \varepsilon$ tel que $e' \preceq e$. L'action a_2 est effectuée si et seulement si $\zeta(g, \{e'\}) = \{true\}$.

2.3.4.8 Fonction ψ

La fonction ψ prend en entrée la liste des modifications produite par φ ($\mu = \varphi(v_t)$) ainsi que la vue v_t à modifier et effectue toutes les actions données par la liste des modifications μ sur v_t pour obtenir v_{t+1} . Le résultat de ψ est la vue v_{t+1} obtenue.

L'algorithme de ψ est le suivant :

```
vt+1 = vt
soit T={a.t, a∈μ}
pour tout t∈T
  soit A={a∈μ tq a.t=t}
  soit A'={a.act, a∈A}
  si del∈A'                               /* cas 2 et 3 */
```

```

    t = correspondant (t, vt+1)
    supprimer (t)
sinon
  si put ∈ A
    si add ∉ A          /* cas 4 */
      v'' = vt
      pour chaque a ∈ A, dans l'ordre lexicographique
        v' = v''
        t = correspondant (t, v')
        pour tout t' ∈ fils(t)
          supprimer (t')
        pour tout r ∈ a.R
          r = correspondant (r, v')
          ajouter_fils (t, ζ(a.R, a.E))
    sinon              /* cas 5 */
      soit a0 ∈ A tq a0.act=put choisie aléatoirement
      pour tout t' ∈ fils(t)
        supprimer (t')
      t = correspondant (t, vt+1)
      ajouter_fils (t, ζ(a0.R, a0.E))
  si add ∈ A          /* cas 3 et 5 */
    pour tout a ∈ A
      si a.act=add,
        t = correspondant (t, vt+1)
        ajouter_fils (t, ζ(a.R, a.E))
retourner vt+1

```

L'algorithme de cette fonction fait appel à deux actions importantes :

- **supprimer**(Υ) qui retire un terme flottant de la vue à laquelle il est rattaché;
- **ajouter_fils**($\Upsilon, \mathcal{P}(\Upsilon)$) qui ajoute un ensemble de termes flottants à l'ensemble des attributs d'un terme donné. Les termes flottants sont alors rattachés à la vue, via leur nouveau parent. Si le terme à modifier est \surd , aucune modification n'est effectuée.¹⁶

NB : Ces deux actions dépendant fortement de l'implémentation, elles ne seront pas détaillées ici.

L'algorithme de ψ fait aussi appel à la fonction *correspondant* : $\Upsilon \times \Lambda \longrightarrow \Upsilon$ qui prend en argument un terme **non flottant** t et une vue v et retourne le terme t' qui *correspond* à t dans v , c'est-à-dire le terme ayant le même chemin dans v que t dans la vue à laquelle il est rattaché. Si aucun terme t' ne correspond à t dans v , la fonction retourne \surd .

¹⁶Ce cas ne se produit pas dans nos algorithmes, puisque tous les termes passés en paramètre à *ajouter_fils* sont non-flottants.

Explication du cas 4

Pour mieux comprendre ces notions assez complexes, considérons le cas le plus difficile : celui des actions *put* concurrentes (cas 4). L'algorithme utilise deux vues annexes v' et v'' initialement égales à v_t . Pour chaque *put* (dans l'ordre lexicographique pour avoir une exécution déterministe), ψ supprime les attributs du terme correspondant à t dans v'' . Il calcule alors l'interprétation dans v' des éléments à ajouter, donnés par $a.R$ et ajoute ces éléments aux attributs du terme correspondant à t dans v'' . Puis il remplace v' par v'' et recommence avec l'action *put* suivante.

Remarques

1. Par construction de μ , les termes modifiés dans ψ (passés en paramètres à *supprimer* et *ajouter_fils*) sont les correspondants dans v_{t+1} des termes *non flottants* référencés dans les actions de la vue v_t (c'est-à-dire retournés par *calcRef* lors du calcul de l'ensemble *Ref* dans φ). Ces termes étant eux-mêmes rattachés à la vue v_{t+1} , c'est bien v_{t+1} qui est modifiée. Les termes non flottants sont donc des points d'entrée dans la vue : nous accédons au tout (la vue) par ses parties (les termes référencés dans les actions élémentaires).
2. L'algorithme de ψ , traduisant notre mécanisme de gestion de la concurrence, met en évidence qu'une action *put* ne peut pas être modéliser par une action *add* et une action *del*. Nous le prouvons de manière plus explicite section 2.3.6.3.

Gestion de la concurrence dans le cas indirect

L'algorithme que nous avons proposé pour ψ gère correctement les deux cas de concurrence indirecte pouvant se produire en VDLp :

1. Si t est un sous-terme d'un terme t' qui est supprimé dans la vue v_{t+1} par une action *del*, les actions qui « utilisent » t dans leurs calculs peuvent toujours être appliquées, puisqu'elles travaillent sur v_t et non sur v_{t+1} .
2. De même, si t est un terme modifié par une action *put* dans la vue v_{t+1} , les actions *add* en concurrence avec ce *put* qui utiliseraient t utiliseront bien l'ancienne valeur (celle de v_t).

Conséquence

La principale conséquence de ce mode d'exécution est que les actions sont toutes exécutées en parallèle. Dans ces langages où l'on décrit un composant à chaque instant, toutes les actions décrites par la vue à l'instant t seront exécutées *en parallèle* au cours du cycle t pour construire la vue à l'instant $t + 1$. Et inversement, toute action décrite à l'instant t est exécutée en séquence avec toute action décrite à l'instant $t + 1$, comme nous le voyons sur la figure 2.7.

La liste μ joue ici le même rôle que les tableaux auxiliaires dans les automates cellulaires.

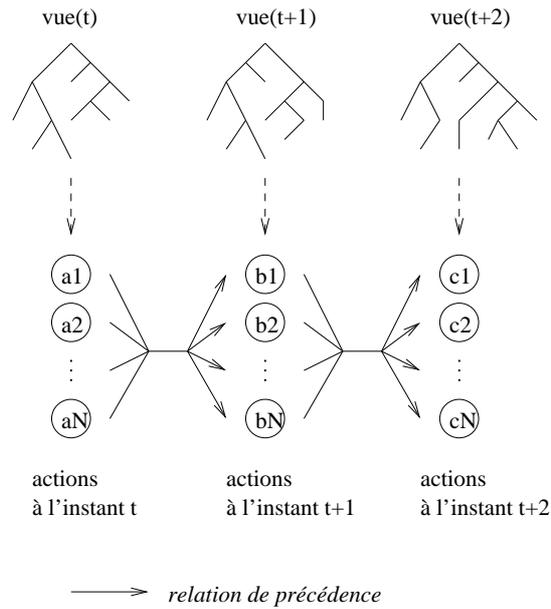


FIG. 2.7 – Exécutions des actions en séquence et en parallèle en VDL.

2.3.5 Exemples

2.3.5.1 Un exemple complet : *Coco*

Nous pouvons illustrer l'essentiel de la sémantique opérationnelle du langage VDL sur un exemple simple. Il s'agit d'un compteur appelé *Coco* dont la vitesse peut être modifiée à l'aide d'événements externes. Cette vue a été introduite par J.-P. Sansonnet dans la version 0.1 de VDL [121]. Sa description en VDLp 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]] ]
]

```

Coco est muni d'un processus *count* qui incrémente l'attribut du terme de concept *value* (il définit son comportement proactif) et de quatre réactions (non munies de gardes) : *stop*, *start* pour l'arrêter ou le redémarrer, *speed up* et *slow down* pour modifier la vitesse.

La vue *Coco* comprend :

- Une variable numérique¹⁷ *value* correspondant à la valeur du compteur, que l'on imprime à chaque cycle ;
- Une variable numérique *speed* correspondant à la vitesse du compteur ;
- Une variable booléenne¹⁸ *running* pour contrôler l'activité du compteur ;
- Un processus dont le nom est *count* et qui ajoute, à chaque cycle, la valeur de la vitesse (*speed*) à la valeur du compteur (*value*) tant que le compteur est en marche (*running*) ;
- Quatre actions de réaction à des événements externes :
 - *stop* pour mettre la valeur de *running* à *false* lorsque la vue reçoit l'événement *stop* ;
 - *start* pour mettre la valeur de *running* à *true* lorsque la vue reçoit l'événement *start* ;
 - *speed up* pour augmenter la vitesse (*speed*) de 1 lorsque la vue reçoit l'événement *faster* ;
 - *slow down* pour diminuer la vitesse de 1 lorsque la vue reçoit l'événement *slower*.

Dans une version implémentée, nous pourrions supposer qu'un observateur statique écrit à l'écran la valeur du terme *value* à chaque cycle d'exécution. Nous verrons dans la section 2.4 comment munir *Coco* d'une interface plus riche.

Exécution

Voici une exécution possible de *Coco* sur 10 cycles :

cycle	ε	<i>value</i>	<i>speed</i>	<i>running</i>
1		1	1	<i>true</i>
2		2	1	<i>true</i>
3	stop	3	1	<i>true</i>
4		3	1	<i>false</i>
5	start	3	1	<i>false</i>
6		4	1	<i>true</i>
7	faster	5	1	<i>true</i>
8	faster,stop	7	2	<i>true</i>
9		10	3	<i>false</i>
10		10	3	<i>false</i>

2.3.5.2 Le Yale Shooting Problem

Nous pouvons écrire en VDLp une vue englobant le petit monde opératoire dans lequel se situe le YSP [57] :

¹⁷Une variable *numérique* est une variable VDL dont l'attribut appartient à \mathbf{R}_{vdl} .

¹⁸Une variable *booléenne* est une variable VDL dont l'attribut appartient à \mathbf{B}_{vdl} .

```

view[dead[false], loaded[false]
    load[ event[load], put[path[loaded],true] ],
    shoot[ event[shoot], guard[get[loaded]],
          put[path[loaded],true] ],
    wait[ event[wait] ]
]

```

2.3.5.3 Modifier le fonctionnement

Comme nous l'avons expliqué précédemment, VDLp est un langage dans lequel les actions peuvent manipuler des actions, comme dans la plupart des langages orientés IA (comme PROLOG et LISP par exemple). Par exemple, la vue suivante est une variante de *Coco* dans laquelle le processus *count* n'est pas muni d'une garde, mais est retiré (resp. ajouté) lorsque la vue reçoit l'événement *stop* (resp. *start*) :

```

view[print [value [0]],
     count [put [path [value], plus [get [value], 1]],
           stop [event [stop], del [path [count]]],
           start [event [start], add [path [view],
                                   count [put [path [value], plus [get [value], 1]]]]
]

```

2.3.6 Propriétés du langage

2.3.6.1 Turing complétude

Le langage VDLp est Turing-complet (toute machine de Turing peut être modélisée par un programme en VDLp et réciproquement) et linéaire en temps et en espace par rapport à une machine de Turing équivalente.

Preuve

Soit M une machine de Turing caractérisée par un ensemble de transitions $M \equiv_{def} \{\delta_i : E, B \mapsto E', B', S\}_{1 \leq i \leq m}$ de cardinal m . Une transition $\delta_i : E, B \mapsto E', B', S$ signifie que si la machine est dans l'état E et si la valeur lue sur la bande est B , δ_i fait passer la machine dans l'état E' , met la valeur B' dans la case et fait bouger la tête suivant le mouvement $S \in \{Droite, Gauche, Rien\}$.

Soit $b = (b_i)_{i>0}$ une suite (infinie) de valeur initiale de la bande de la machine de Turing : $\forall i, b_i \in \{\top, \perp\}$. Nous supposons que la machine M est à entrée finie, c'est à dire qu'il existe $i_0 \in \mathbf{N}$ tel que $\forall i > i_0, b_i = \perp$.

Nous allons construire une vue v dont l'exécution correspond à celle de la machine M pour l'entrée b :

- L'état de la machine sera représenté par un terme de concept *etat* dont l'unique attribut donne le numéro de l'état courant de la machine (par exemple, *etat*[3]). Initialement, nous avons *etat*[1].

- La bande de la machine sera représentée par un terme de concept *bande*. Puisqu'il est impossible de représenter en informatique une bande semi-infinie, nous utiliserons une bande contenant à tout instant un nombre fini de cases et nous lui ajouterons des cases au fur et à mesure de l'exécution lorsque cela est nécessaire.

Ces cases seront représentées par des termes dont le concept est le numéro de la case, de 1 à n (1 étant la première case de la bande semi-infinie), et ne possédant qu'un seul attribut \top ou \perp , selon la valeur de la case de la bande. Initialement, la bande de la machine VDL est définie par le terme suivant, correspondant à l'entrée b : $bande [1[b_1] \ 2[b_2] \ \dots \ i_0[b_{i_0}]]$. Par exemple, nous aurons :

```
bande [1[ $\top$ ], 2[ $\perp$ ], 3[ $\top$ ], 4[ $\top$ ], 5[ $\perp$ ]]
```

- La position de la tête de lecture de la machine sur la bande sera représentée par un terme de concept *pos* dont l'unique attribut donne le numéro de la case où se situe la tête de lecture (par exemple, $pos[3]$).

Initialement, nous avons $pos[1]$.

- L'ensemble des transitions de la machine M sera représenté par un terme de concept *machine* muni des sous-termes de concept $delta\ 1, \dots, delta\ m$ représentant respectivement les instructions $\delta_1, \dots, \delta_m$ de la machine de Turing.

Soit $\delta_i : E, B \rightarrow E', B', S$ une transition : δ_i sera représentée par l'action $delta\ i$ de la manière suivante :

```
"delta i "[guard[and[equals[get[etat],E],
                    equals[get[bande,get[pos]],B]]],
           put[path[etat],E'],
           put[path[bande,get[pos]],B']],
si S = Gauche :
   proc[guard[not[equals[get[pos],1]]],
        put[path[pos],plus[get[pos],-1]]]
]
si S = Droite :
   proc[guard[equals[get[pos],size[get[bande]]]],
        put[path[bande],
            node[concept[plus[get[pos],1]], $\perp$ ]]
],
   put[path[pos],plus[get[pos],1]]
sinon : (aucune action)
]
```

Toute transition de la machine de Turing initiale est alors effectuée en un et un seul cycle VDLp. Notre langage est donc linéaire en temps par rapport à une machine de Turing. Il est aussi linéaire en espace puisque la description de la machine est de taille constante et la bande est complétée au fur et à mesure que cela est nécessaire.

Signalons que nous n'avons pas exprimé les conditions d'arrêt de la machine, qui ne sont pas nécessaires ici.

Remarque

Nous voyons dans la construction d'une machine de Turing que VDLp permet de modéliser des données structurées :

- Des enregistrements (pour chaque opérations de la machine de Turing) ;
- Des tableaux (pour l'ensemble de la machine) ;
- Des listes (pour la bande).

De manière générale, bien qu'étant un langage non typé dont la structure est réduite à sa plus simple expression (un arbre), VDLp permet de modéliser des structures de données plus complexes. Nous verrons au chapitre 6 que VDLp peut être intégré dans un langage structuré et type comme XML.

2.3.6.2 Opérateurs CSP

Le langage CSP de Hoare [63] définit un ensemble d'opérateurs de compositions d'actions dans les processus communicants qui permettent de modéliser l'exécution de tout programme itératif. Soit a et b deux actions et c une formule logique :

- $seq(a, b)$ exprime la composition séquentielle de deux actions, c'est-à-dire l'action obtenue en effectuant a puis en effectuant b ;
- $alt(a, b)$ exprime le choix non déterministe entre deux actions, c'est-à-dire l'action obtenue en effectuant a ou en effectuant b ;
- $par(a, b)$ exprime la composition parallèle de deux actions, c'est-à-dire l'action obtenue en effectuant a et b simultanément ;
- $cond(c, a)$ exprime le test conditionnel, c'est-à-dire l'action obtenue en effectuant a si et seulement si c est vraie dans la situation courante ;
- $while(c, a)$ exprime la boucle conditionnelle, c'est-à-dire l'action obtenue en effectuant a tant que c est vraie dans la situation courante ;

Nous pouvons montrer que, comme GOLOG [79], VDLp permet de modéliser chacun de ces opérateurs, et par conséquent que VDL est aussi expressif que les langages de programmation itératifs.

Séquence : $seq(a, b)$

Soient a et b deux actions VDLp. L'action $r = seq(a, b)$ est définie par le terme suivant en VDL :

```
actionSeq[ tmp[a_pas_faite] ,
           proc1[guard[equals[get[tmp], a_pas_faite]],
                a, put[path[actionSeq, tmp], a_faite]],
           proc2[guard[equals[get[tmp], a_faite]],
                b, put[path[actionSeq, tmp], a_pas_faite]] ]
```

Composition parallèle : $par(a, b)$

Les actions VDLp sont toujours effectuées en parallèle. Par conséquent, l'action $r = par(a, b)$ est définie simplement par :

```
actionPar[ a , b ]
```

Choix non déterministe : $alt(a, b)$

Nous pouvons simuler le *alt* en deux cycles en VDLp, en utilisant le modifieur *random* et les autres opérateurs CSP :

```
actionAlt[ tmp,
  seq(put [path [tmp], random],
    par(cond("greater than" [l [get [tmp]], h [0.5]], a),
      cond("greater than" [l [0.5], h [get [tmp]]], b))
  ) ]
```

Itération : $while(c, a)$

Les processus VDLp sont effectués *tant que* leurs gardes sont valides. Ils correspondent donc aux les boucles conditionnelles :

```
ActionWhile[guard [c], a]
```

Test conditionnel : $cond(c, a)$

Un test conditionnel est une boucle *while* exécutée une seule fois. Elle est donc modélisable en VDL par :

```
ActionIf[ tmp[true], guard[and [c, get [tmp]]],
  a, put [path [tmp], false]]
```

Conclusion

La partie procédurale du formalisme de description des vues (VDL) permet donc d'avoir des séquences, des tests conditionnels, des boucles et des actions en parallèle. Il est donc possible d'exprimer en VDL n'importe quel programme impératif classique.

2.3.6.3 Actions élémentaires

Les actions élémentaires VDL forment une base des modifications possibles d'une vue.

Preuve

Rappelons que l'action *add* permet simplement d'ajouter des termes, *put* d'en retirer et d'en ajouter et *del* uniquement d'en retirer. Il existe deux « clefs de voûte » de cette propriété d'indépendance entre les actions élémentaires :

- Les références (*path* ou *get*) permettent d'accéder à un ensemble de termes, sans distinction entre ceux-ci. Si une référence R a pour résultat un ensemble de termes T , il n'existe pas de référence $R' \subset R$ dont le résultat est $T' \subsetneq T$, compte tenu de l'algorithme de *calcRef*.¹⁹
- Les actions sont effectuées en parallèle dans ϕ_{exec} et calculent leur « résultat » sur la vue v_{t+1} .

Ces deux propriétés permettent de montrer l'indépendance entre les action élémentaires :

1. *add* ne peut pas être modélisé à l'aide de *put* et de *del* dans le cas général.

Soit $t = add[path[t_1, \dots, t_k], t_{k+1}, \dots, t_n]$ dans la vue v .

Soit $R = calcRef(\{t_1, \dots, t_k\}, v)$. t ajoute l'interprétation des t_{k+1}, \dots, t_n aux attributs des éléments de R . Il ne supprime aucun terme.

Seule l'action *put* peut être utilisée pour ajouter des attributs.

Si $|R| \leq 1$, cette action peut « remettre » les attributs qu'elle a retiré, obtenus par un *get* :

$$put[path[t_1, \dots, t_k], t_{k+1}, \dots, t_n, get[t_1, \dots, t_{k+1}]]$$

En revanche, si $|R| > 1$, le *get* retourne les attributs de *tous* les éléments de R . Il n'est donc pas possible de « restituer » à chaque élément de R ses propres attributs.

2. *del* ne peut pas être modélisé à l'aide de *put* et de *add* dans le cas général.

Soit $t = del[path[t_1, \dots, t_n]]$ dans la vue v . Plaçons nous dans le cas simple où $calcRef(\{t_1, \dots, t_k\}, v) = \{x\}$, $calcRef(\{t_1, \dots, t_k\}, v) = \{y\}$ et $x \in fils(y)$ mais $|fils(y)| > 1$ (x n'est pas le seul attribut de y). Seule l'action *put* permet de supprimer des termes, mais elle supprime *tous* les attributs d'un terme, sans distinction, et, à l'aide d'un *get*, une action *add* ou une action *put* peut seulement ajouter *tous* les attributs du terme modifié, sans distinction.

3. *put* ne peut pas être modéliser à l'aide de *add* et *del* dans le cas général.

Elle peut l'être en séquence (*put* est un *del* suivit (*seq* au sens de CSP) d'un *add*), mais *put* permet d'effectuer ces deux opérations **en 1 seul cycle**. En particulier, il permet de modéliser l'affectation de variables en programmation impérative. Puisque *add* permettant seulement de retirer, *del* seulement d'ajouter et compte tenu de notre mécanisme de concurrence qui fait prévaloir *del* sur les autres actions, *add* et *del* ne peuvent pas être composés pour modéliser un *put*.

¹⁹En effet, « réduire » la référence donne au mieux les termes parents. Pour obtenir un sous ensemble de T , il faudrait « préciser » certains éléments de la référence. Par exemple, dans la vue :

```
view[ a[x[y[1]]], a[z[y[1]] ]
```

$R = \{a, y\}$ a pour résultat les deux termes de concept y alors que $R' = \{a[z], y\}$ a pour résultat $\{y[2]\}$. Mais R' n'est pas inclus dans R .

2.4 L'observateur d'interface

Pour munir les vues d'un interface, nous proposons d'utiliser un observateur statique, appelé *interf*, qui construit une visualisation de la vue en fonction de certains modifieurs.

2.4.1 Principe général

L'observateur *interf* construit une interface graphique de la vue dans lequel sont placés les composants graphiques (boutons, zones de texte, etc) en fonction des modifieurs graphiques, puis affiche ce composant.

Dans l'implémentation de VDLp en Java, présentée au chapitre 7, l'interface graphique construite est un *java.awt.Panel*, affiché dans une applet. Lorsque l'utilisateur agit sur les composants graphiques, l'applet envoie des événements externes à la vue.

Ce schéma de principe est illustré sur la figure 2.8.

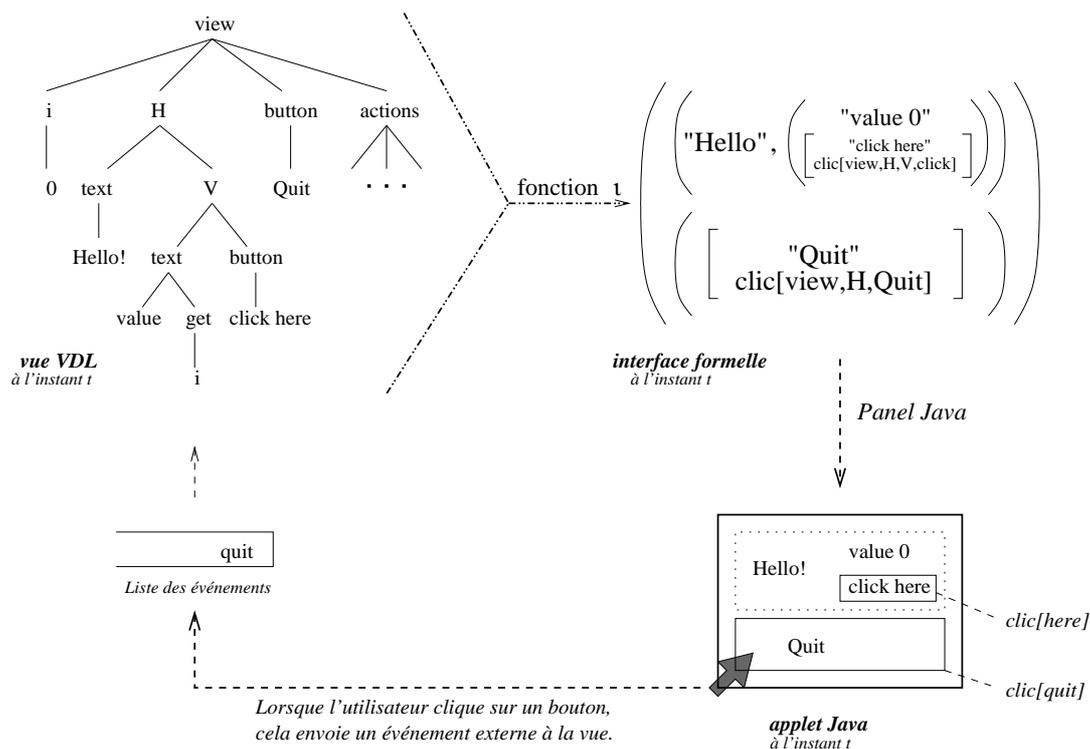


FIG. 2.8 – Gestion de l'interface graphique des vues.

Dans cette section, nous présentons uniquement la construction de l'interface formelle. La génération de l'interface en Java est présentée au chapitre 7.

Liste des modifieurs de *interf*

Les modifieurs utilisés par l'observateur statique *interf* sont les suivants :

- **H** et **V** pour placer horizontalement et verticalement (respectivement) les composants graphiques ;
- **button** définir des boutons, **clik** pour définir l'événement qu'il faut envoyer lorsque l'utilisateur clique dessus ;
- **text** pour visualiser du texte ou définir le texte affiché dans un bouton ;
- "**colored text**" pour visualiser du texte en couleur ;
- **figure** pour définir des figures géométriques ;
- **shape** pour donner la forme d'une figure, parmi **square** (carré), **circle** (rond) et **triangle** ;
- **size** pour donner la taille d'une figure, parmi **big** (grand), **small** (petit), **tiny** (très petit) ou un entier dans sa représentation canonique en VDLp ;
- **color** pour donner la couleur d'une figure, d'un bouton ou d'un texte, parmi **black** (noir), **white** (blanc), **red** (rouge), **green** (vert), **blue** (bleu), **yellow** (jaune) et **gray** (gris) ;
- **cell** pour dessiner une case vide pouvant contenir des figures géométriques ;
- "**blank space**" pour dessiner des zones vide d'une taille donnée ;
- **component** pour utiliser directement du code Java dans la description de l'interface graphique.

De plus, l'observateur *interf* utilise l'interprétation canonique ζ définie dans *exec*. Il utilise donc les modifieurs interprétés par ζ .

2.4.2 Sémantique

VDLp est indépendant de tout langage de programmation. C'est la raison pour laquelle la fonction caractéristique ϕ_{interf} présentée ici construit une interface formelle. C'est cette interface qui sera envoyée à une applet Java pour que celle-ci construise et affiche le *Panel* correspondant.

2.4.2.1 Définitions

Nous notons :

- Ω l'ensemble des composants graphiques, décrits de manière formelle, utilisés dans l'interface formelle construite par ϕ_{interf} ;
- $Coul$ l'ensemble des couleurs possibles pour les composant graphiques : $Coul = \{noir, blanc, rouge, vert, bleu, jaune, gris\}$.

Pour positionner les éléments au moyen de matrices holophrastiques, nous utiliserons des vecteurs orientés verticalement ou horizontalement. Nous avons donc deux sortes d'espaces (ou *contenants*) : les espaces verticaux et les espaces horizontaux.

L'ensemble Ω est défini de la manière suivante :

composant (Ω) ::= *contenant* | *bouton* | *zone_texte* | *figure* | \square
contenant ::= *espace_vertical* | *espace_horizontal*

$$\begin{aligned}
\text{espace_horizontal} &::= (\text{composant} \ \dots \ \text{composant}) \\
\text{espace_vertical} &::= \begin{pmatrix} \text{composant} \\ \text{composant} \\ \dots \\ \text{composant} \end{pmatrix} \\
\text{zone_texte} &::= \begin{bmatrix} \text{"texte"} \\ \text{Coul} \end{bmatrix} \quad \text{bouton} ::= \begin{bmatrix} \text{"texte"} \\ \text{Coul} \\ \Upsilon \end{bmatrix} \quad \text{figure} ::= \begin{bmatrix} \text{forme} \\ \text{Coul} \\ \mathbb{N} \end{bmatrix} \\
\text{forme} &::= \text{rond} \mid \text{carré} \mid \text{triangle}
\end{aligned}$$

Les boutons sont des éléments graphiques munis d'un titre et capables de produire des événements externes. Un bouton VDL est donc composé d'un titre ("*texte*") et d'un événement externe ($evt \in \Upsilon$) envoyé à la vue lorsque l'utilisateur clique sur le bouton dans l'interface graphique.

Exemple

Par exemple, voici une interface formelle possible :

$$\left(\begin{array}{c} \left(\text{"coucou"} \ \square \ \begin{bmatrix} \text{rond} \\ \text{rouge} \\ 40 \end{bmatrix} \right) \\ \begin{bmatrix} \text{"blabla"} \\ \text{vert} \end{bmatrix} \\ \left(\begin{bmatrix} \text{"stop"} \\ \text{clic}[\text{stop}] \end{bmatrix} \ \begin{bmatrix} \text{"start"} \\ \text{clic}[\text{start}] \end{bmatrix} \right) \end{array} \right)$$

2.4.2.2 Algorithme

La fonction $\phi_{\text{interf}} : \Upsilon \longrightarrow \Omega$ prend en argument un terme, initialement égal à la vue et construit une interface formelle en recherchant dans le terme les modifieurs graphiques.

Soit $t = c[t_1, \dots, t_n]$. La fonction ϕ_{interf} est définie récursivement par :

Cas général

– Dans le cas général :

$$\phi_{\text{interf}}(t) = \bigcup_{i \in [1, n]} \phi_{\text{interf}}(t_i, \text{coul})$$

- Si $c \in I_t = \{add, put, del, path, get, "event\ get", "size\ ref"\}$:

$$\phi_{interface}(c[t_1, \dots, t_n]) = \emptyset$$

En effet, les composants graphiques décrits dans les actions élémentaires (*add*, *put* et *del*) sont des composants qui doivent être ajoutés à la vue lors de l'exécution d'une action. Ils ne sont donc pas encore réellement dans la partie graphique de la vue. Ils doivent donc être ignorés.

Quant aux références, rien n'interdit *syntactiquement* d'utiliser un modifieur graphique dans une référence²⁰. Comme pour les actions élémentaires, ces modifieurs ne décrivent pas un composant réel. Il faut donc les ignorer.

Positionnement des éléments

- Si $c = view$,

$$\phi_{interface}(view[t_1, \dots, t_n]) \equiv_{def} \phi_{interface}(V[t_1, \dots, t_n], coul)$$

L'espace est donc orienté verticalement par défaut (ce choix est arbitraire).

- Si $c = V$,

$$\phi_{interface}(t, coul) = \left\{ \begin{array}{c} \phi_{interface}(t_1)_1 \\ \dots \\ \phi_{interface}(t_1)_{k_1} \\ \vdots \\ \phi_{interface}(t_n)_1 \\ \dots \\ \phi_{interface}(t_n)_{k_n} \end{array} \right\}$$

où $k_i = card(\phi_{interface}(t_i))$.

- Si $c = H$,

$$\phi_{interface}(t) = \left\{ \left(\phi_{interface}(t_i)_{k_{ij}}, i \in [1, n], j \in [1, k_i] \right) \right\}$$

où $k_i = card(\phi_{interface}(t_i))$.

²⁰En revanche, sémantiquement, c'est moins facilement concevable : cela signifierait que l'on attribue à un même concept deux sémantiques explicationnelles (celle de composant graphique et celle de discriminant dans la référence). Dans une vue bien construite, nous utiliserions deux modifieurs, l'un englobé dans l'autre.

Zones de texte et boutons

- Si $c = \text{text}$,

$$\phi_{interf}(t) = \left\{ \begin{bmatrix} \varphi_{text}(t) \\ \text{noir} \end{bmatrix} \right\}$$

où $\varphi_{text}(t) = \bigodot_{t' \in \bigcup_{i=1}^n \zeta(t_i, \emptyset)} f_{text}(t)$.

Remarquons que nous ne traitons pas au sens de l’affichage (*i.e.* par ϕ_{interf}) les attributs t_1, \dots, t_n du terme. Sémantiquement, un *text* ne devrait pas avoir de fils graphiques. S’il y en a, ils sont ignorés.

- Si $c = \text{”colored text”}$,

soit $C = \{t_i \text{ tq } \text{concept}(t_i) = \text{color}\}$. Soit $C' = \bigcup_{t' \in \text{fils}(t_i), t_i \in C} \zeta(t')$. Si $|C'| = 0$, $\text{coul}' = \text{noir}$. Sinon, soit $c' \in C'$ choisi aléatoirement²¹. Alors $\text{coul}' = \varphi_{coul}(\text{concept}(c'))$ où $\varphi_{coul} : A^* \rightarrow \text{Coul}$ est la fonction qui associe une couleur formelle à un modifieur (*noir* pour *black*, *etc*) et qui retourne *noir* le concept passé en paramètre n’est pas un modifieur de couleur.

Dans la suite, pour tout terme t contenant des attributs de concept *color*, nous dirons que coul' construite suivant cet algorithme est la couleur associée canoniquement à t .

Soit $T = \{t_i \text{ tq } \text{concept}(t_i) = \text{text}\}$ et $\text{text} = \bigodot_{t_i \in T} \varphi_{text}(t_i)$. Alors :

$$\phi_{interf}(t) = \left\{ \begin{bmatrix} \text{text} \\ \text{coul}' \end{bmatrix} \right\}$$

- Si $c = \text{button}$,

soit coul' la couleur associée canoniquement à t . Soit $T = \{t_i \text{ tq } \text{concept}(t_i) = \text{text}\}$ et $\text{text} = \bigodot_{t_i \in T} \varphi_{text}(t_i)$. Soit $K = \{t_i \text{ tq } \text{concept}(t_i) = \text{clic}\}$. Si $K = \emptyset$, evt est un terme de concept *clic* ayant un attribut atomique, de concept *text*. Sinon, $\text{evt} = \text{clic}[\bigcup_{t_i \in K} \text{fils}(t_i)]$. Alors :

$$\phi_{interf}(t) = \left\{ \begin{bmatrix} \text{text} \\ \text{coul}' \\ \text{evt} \end{bmatrix} \right\}$$

NB : Les événements envoyés par les boutons sont des termes flottants.

²¹Si le programmeur donne plusieurs attributs de concept *color*, le choix de la couleur est donc non déterministe.

- Si $c = \text{"blank space"}$,

soit $I = \bigcup_{i \in [1, n]} \zeta(t_i)$. Si $|I| = 1$ et $I_1 \in \mathbb{R}_{val}$, alors k est l'interprétation canonique de I_1 dans \mathbb{R} . Sinon, $k = 1$. Alors :

$$\phi_{interf}(t) = \left\{ \left(\begin{array}{c} \text{" " } \\ \text{noir} \end{array} \right) \quad \begin{array}{c} k \text{ fois} \\ \dots\dots\dots \end{array} \quad \begin{array}{c} \text{" " } \\ \text{noir} \end{array} \right\}$$

Figures géométriques

- Si $c = \text{figure}$,

soit $coul'$ la couleur associée canoniquement à t . Soit $T = \{t_i \text{ tq } concept(t_i) = size\}$ et $T' = \bigcup_{t' \in fils(t_i), t_i \in T} \zeta(t')$. Si $|T'| = 0$, $taille = 30$. Sinon, soit $t' \in T'$ choisi aléatoirement. Si $concept(t') \in \mathbb{R}_{val}$, $taille$ est la partie entière de l'interprétation canonique de $concept(t')$ dans \mathbb{R} ; si $concept(t') = big$, $taille = 40$; si $concept(t') = small$, $taille = 20$; si $concept(t') = tiny$, $taille = 10$. Sinon, $taille = 30$.

De même, soit $F = \{t_i \text{ tq } concept(t_i) = shape\}$ et $F' = \bigcup_{t' \in fils(t_i), t_i \in F} \zeta(t')$. Si $|F'| \neq 0$, soit $f' \in F'$ choisi aléatoirement. Si $concept(f') = square$, $forme = carré$; si $concept(f') = circle$, $forme = rond$; si $concept(f') = triangle$, $forme = triangle$. Alors :

$$\phi_{interf}(t) = \left\{ \begin{array}{c} forme \\ coul' \\ taille \end{array} \right\}$$

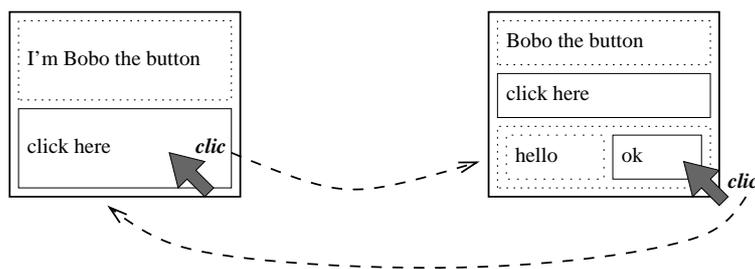
Sinon, c'est-à-dire si $|F'| = 0$ ou si $concept(f') \notin \{square, triangle, circle\}$, $\phi_{interf}(t) = \{\square\}$. La « figure par défaut » est une case vide.

- Si $c = cell$, $\phi_{interf}(t) = \{\square\}$.

2.4.3 Exemples

2.4.3.1 Bobo le bouton

Bobo est une vue contenant initialement une zone de texte « *I'm Bobo the button* » et, en dessous, un bouton « *click here* ». Lorsque l'utilisateur clique sur ce bouton, deux nouveaux composants sont ajoutés en dessous : une zone de texte « *hello* » et, à côté, un bouton « *ok* ». Si l'utilisateur clique alors sur *ok*, ces composants disparaissent et la vue se retrouve dans l'état initial.



Le code VDL de ce composant est le suivant :

```
view [text ["I'm Bobo the button"],
      button [text ["Click here"], clic [here]],
      proc1 [event [clic [here]],
            add [path [view], H [text [Hello],
                          button [text [OK], color [blue]]]]],
      proc2 [event [clic [OK]], del [path [H]]]
    ]
```

Ce qui correspond à la vue donnée figure 2.9.

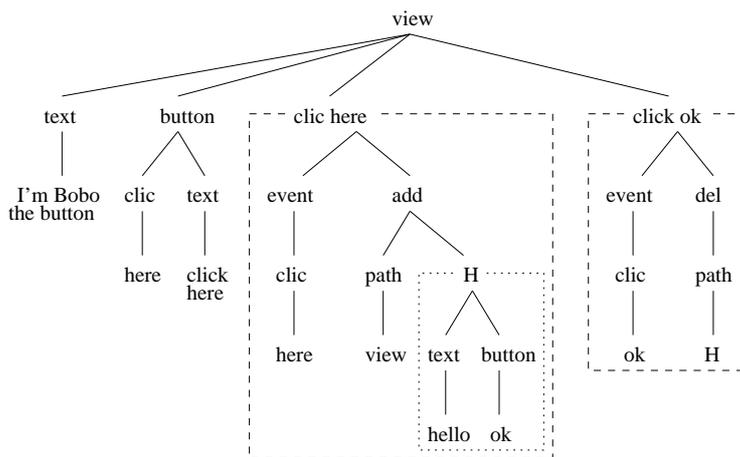


FIG. 2.9 – *Bobo le bouton.*

2.4.3.2 Jojo le jeu (inspiré du monde de cubes [149])

Jojo est une vue englobant le petit monde opératoire présenté sur la figure 2.10, dans lequel l'utilisateur peut manipuler des figures géométriques de couleur à l'aide d'un bras articulé ne pouvant contenir qu'un seul objet à la fois. Ce monde est composé de :

- une grille 3×3 dont chaque case peut contenir un objet
- une main permettant de saisir un objet
- une pioche contenant les objets restant

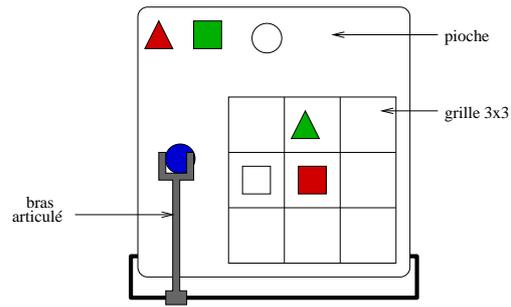


FIG. 2.10 – *Jojo* le Jeu

Le code de *Jojo* en VDLp est le suivant :

```

view[ out[H[cell[
  figure[color[white],shape[square]],
  figure[color[white],size[small],shape[circle]],
  figure[color[red],size[big],shape[triangle]],
  figure[color[red],size[small],shape[square]],
  figure[color[blue],shape[circle]],
  figure[color[green],shape[square]],
  figure[color[green],size[tiny],shape[triangle]] ]]],

H[text[taken :],taken[cell],"blank space"[6]],

grid[ upper[H["blank space"[2],
  left[cell],middle[cell],right[cell],
  "blank space"[3] ]],
  middle[H["blank space"[2],
  left[cell],middle[cell],right[cell],
  "blank space"[3] ]],
  lower[H["blank space"[2],
  left[cell],middle[cell],right[cell],
  "blank space"[3] ]], ],

drop[ event[drop],
  guard["is a"[get["event get"[drop],cell]],
  guard[or[equals[size[get["event get"[drop],cell]],0],
  equals["event get"[drop],out]]],
  add[path["event get"[drop],cell],get[taken,cell]],
  put[path[taken,cell]] ],

take[ event[take[figure]],
  guard[equals[size[get[taken,cell]],0]],

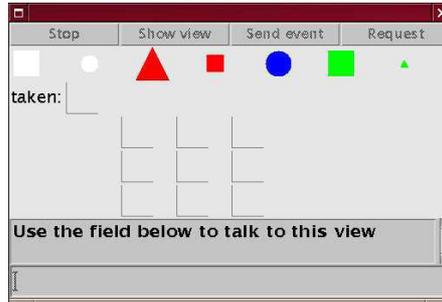
```

```

guard[equals["size ref"["event get"[take]],1]],
del[path["event get"[take]]],
put[path[taken,cell],"event get"[take]] ] ]

```

Dans l'implémentation de VDLp proposée au chapitre 7, l'interface Java de *Jojo* est la suivante :



2.5 Bilan

Le cadre formel que nous avons proposé pour la représentation des composants actifs intègre au sein d'une même représentation la description de la structure, très simplifiée, et la description du fonctionnement.

La sémantique opérationnelle des actions VDLp présentée dans ce chapitre est une extension de ce que nous avons proposé dans [115]. Elle a été définie de telle manière à pouvoir servir de support pour les mécanismes de construction de réponses aux requêtes produites par le MLN, à partir des questions de l'utilisateur sur le fonctionnement du composant avec lequel il interagit. En effet, dans ce modèle d'exécution basé sur la réécriture d'arbre, le fonctionnement est décrit explicitement dans la vue et peut donc être manipulé pour répondre à des requêtes sur le fonctionnement, en utilisant la sémantique opérationnelle du langage définie par la fonction caractéristique ϕ_{exec} . Chaque concept est associé à une sémantique explicative « de bon sens » qui sera utilisé dans la construction de la réponse.

Nous n'avons pas défini ici la sémantique opérationnelle des modificateurs *script*, *global* et *component* qui permettent d'inclure directement du code Java dans les vues. Celle-ci est en effet fortement liée à l'implémentation que nous avons fait des observateurs *exec* et *interf* dans un interprète VDL en Java. Elle sera présentée et expliquée au chapitre 7.

Le langage VDLp est un langage de programmation à part entière et il vérifie les propriétés intéressantes pour notre étude. En utilisant des observateurs statiques comme l'observateur *interf*, il est possible de définir facilement une très large classe de composants actifs dans ce langage.

Cependant, nous pouvons envisager plusieurs extensions pour rendre le langage VDLp encore plus expressif :

1. Dans un premier temps, nous voudrions intégrer VDLp dans un modèle structurel riche, afin de pouvoir manipuler des variables structurées et typées en VDL. Nous monterons que c'est possible au chapitre 6 en utilisant XML comme modèle support.
2. À l'aide des ensembles \mathbb{R}_{vdl} et \mathbf{B}_{vdl} , l'interprétation canonique ζ peut manipuler des nombres et des booléens. Les concepts étant des chaînes de caractères, nous pourrions aussi facilement munir l'observateur *exec* de modifieurs permettant la manipulation de chaîne de caractères (par exemple : *concat*, *substring*, *etc*). Cette extension ne présente aucune difficulté, ni théorique ni technique, et n'apporte rien à la sémantique opérationnelle de VDLp. Elle permet seulement de modéliser facilement une plus large classe de composants actifs, comprenant les agents manipulant des données textuelles. C'est la raison pour laquelle nous ne l'avons pas présenté ici.
3. Le langage VDLp ne permet pas de définir des fonctions paramétrées : l'ensemble du calcul doit être répété dans la vue à chaque fois qu'il doit être effectué. Nous envisageons de proposer un mécanisme de fonctions en VDL en utilisant les modifieurs suivants :
 - **function** pour déclarer une fonction. Les termes de concept *function* sont ignorés par l'interprétation canonique et la fonction φ ;
 - **param** pour déclarer les paramètres de la fonction, **result** pour le résultat ;
 - **call** pour appeler une fonction, utilisé par ζ .

Par exemple :

```
view[ f[ function[param[x],
          put[path[result], plus[get[x], 2]]],
      ...
      print[call[path[f], x[4]]] ]          → 6
```

Cette extension est relativement simple du point de vue théorique, mais elle l'est moins du point de vue technique et n'est pas nécessaire pour notre étude, c'est-à-dire la construction de réponses à des questions de bon sens sur le fonctionnement. C'est la raison pour laquelle nous ne la détaillerons pas ici.

4. Nous pouvons remarquer que la sémantique opérationnelle de VDL est fortement parallélisable : toutes les actions sont recherchées par φ (parcours en profondeur) puis effectuées par ψ en parallèle. C'est la raison pour laquelle nous pourrions paralléliser l'exécution des vues : construction de μ par φ en parallèle, puis distribution de μ sur tous les processeurs, puis exécution ψ en parallèle et réunion de toutes les sous-vues obtenues en une seule. Cette extension constitue un travail d'ingénieur intéressant mais relativement conséquent.
5. Dans l'algorithme de ϕ_{interf} , nous reconstruisons l'interface graphique à chaque cycle. C'est une méthode coûteuse. Nous pourrions envisager une extension dans laquelle la fonction ϕ_{interf} mémorise l'interface au cycle précédent et se

contente de la *modifier* en fonction des composants ayant été ajoutés, enlevés ou modifiés dans la vue.

Cette extension est relativement complexe à mettre en œuvre car, du point de vue théorique, la vue peut être entièrement modifiée à chaque cycle.

Dans le chapitre suivant, nous nous intéressons aux questions qu'un utilisateur peut poser concernant le fonctionnement d'un composant actif décrit en VDL. Nous proposons un langage de requête qui permet de modéliser la plupart de ces questions puis, dans le chapitre 4, nous présentons des algorithmes de traitement.

Chapitre 3

Langage de requêtes

Le langage de description de composants que nous avons présenté au chapitre précédent permet de modéliser et d'implémenter des composants actifs capables de se représenter leur fonctionnement, afin de pouvoir raisonner dessus et répondre *en cours d'exécution* aux questions d'un utilisateur portant sur leurs actions et leur exécution.

Dans le projet *InterViews*, les questions de l'utilisateur sont traduites en requêtes formelles par le module de traitement du langage naturel (MLN). Ces requêtes sont ensuite traitées par un module de raisonnement au niveau formel pour produire une réponse, effectuer une action, modifier la base de connaissances, *etc.* La réponse construite par ce module est elle-même une requête formelle, qui peut-être utilisée pour produire une réponse en langue naturelle. La figure 1.1 page 13 illustre ce schéma de principe.

Dans ce chapitre, nous présentons le langage de requête que nous utilisons pour modéliser les questions qu'un utilisateur humain peut poser concernant le fonctionnement d'un composant actif et les réponses que le système peut donner. Dans la première section, nous définissons les besoins de ce modèle de requêtes et les contraintes qu'il doit respecter. Nous proposons une catégorisation des différents types de questions qu'un utilisateur humain peut poser à partir d'un corpus sur un composant simple. Dans la deuxième section, nous présentons un modèle de requêtes simple, fondé sur cette catégorisation, qui répond à notre besoin. Nous montrons dans la troisième section qu'il permet de modéliser toutes les questions de notre corpus. Nous étudions ces limites et nous montrons que ce modèle est bien adapté pour notre problématique.

3.1 Étude du problème

3.1.1 Contexte de l'étude

Dans notre étude, nous essayons de modéliser et d'implémenter des composants actifs qui seraient capables de répondre en cours d'exécution aux questions de l'utilisateur portant sur leurs actions et leur exécution.

3.1.1.1 Rappel de l'architecture

Notre travail se place dans le contexte du projet *InterViews*. Dans ce projet, les questions de l'utilisateur sont traduites en requêtes formelles à l'aide d'outils de traitement de la langue naturelle et en s'appuyant sur la description des composants en VDL, qui permet de connaître à tout instant leur structure statique et dynamique. Ces requêtes sont ensuite traitées par un Module de Raisonnement sur le Fonctionnement (MRF) qui construit une réponse à partir de l'analyse de la description des actions des actions du composant. Ce schéma de principe est illustré sur la figure 1.1 page 13.

L'idée que nous soutenons dans notre 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 et de mieux traiter les problèmes spécifiques au TALN. La construction de réponses dans un environnement intégré comme le projet *InterViews* résulte alors d'une interaction entre le MLN et le MRF.

3.1.1.2 Le traitement du langage naturel

Dans cette thèse, nous n'aborderons pas les problèmes liés au TALN. Nous nous intéressons uniquement à la construction de réponses formelles à des requêtes formelles.

Le MLN réalisé au sein du projet par S. Gérard et G. Pitel [52, 125, 102] est trop complexe pour être présenté ici dans son intégralité. Toutefois, nous pouvons en présenter les grandes lignes, afin de mieux situer notre travail par dans le projet *InterViews* et de mettre en évidence la séparation des problèmes dans la construction d'une réponse à une question.

Lorsque l'utilisateur pose une question sur les données ou sur le fonctionnement du composant actif, le MLN apparie les *concepts VDL* dans la vue avec les mots ou les groupes de mots de la question de l'utilisateur, à l'aide d'une classification sémantique comme celle proposée par *WordNet* [38].

Il construit ainsi une requête formelle dont les arguments sont les éléments de la description procédurale du composant, sélectionnés à partir de la question en langue naturelle de l'utilisateur. Le mécanisme mis en œuvre, reposant sur un ensemble de règles bien définies, ne peut pas être présenté ici.

3.1.1.3 Traitement de la référence

Le travail d'association des éléments formels avec les mots de la phrase de l'utilisateur est appelé en TALN *traitement de la référence* [103]. Dans le cadre d'une interaction avec un utilisateur ordinaire, nous pouvons mettre en évidence plusieurs difficultés dans le traitement de la référence.

Considérons par exemple le composant actif *Coco* défini au chapitre précédent, section 2.3.5.1 page 73. Si l'utilisateur lui demande « *Quelle est la valeur de ta*

vitesse ? », la question porte sur le terme *speed* et non sur *value*. Le traitement de cette référence fait uniquement appel à des notions linguistiques.

Référence structurelle

Considérons maintenant une extension du composant graphique *Jojo* défini section 2.4.3.2 page 86 dans laquelle les données sont structurées et typées.

Supposons par exemple que ce composant contient une variable v_1 de type *Figure*, où *Figure* est un enregistrement à 3 champs :

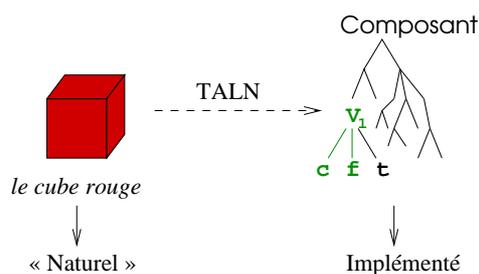
- f de type *Forme* = {*Cube*, *Boule*} ;
- c de type « entier hexadécimal à 6 chiffres » ;
- $taille$ de type « entier entre 0 et 50 ».

Supposons que $v_1.f = \textit{Cube}$ et $v_1.c = \#FF0000$ ¹. L'interface utilisateur graphique (GUI) utilise cette description pour afficher une représentation graphique de v_1 sous la forme d'un cube rouge.

L'utilisateur n'a pas accès aux variable : il voit simplement un cube rouge à l'écran et c'est ainsi qu'il va référencer l'objet, suivant son bon sens. Par exemple, il pourra poser la question suivante : « *Le cube rouge est-il plus gros que la boule verte ?* ». Au contraire, le système ne connaît que la représentation interne (en langage de programmation) de cet objet, qu'il référence par v_1 . Il ne peut que travailler au niveau des propriétés formelles de l'objet ($v_1.c = \textit{Cube}$). Il existe donc une différence entre le perçu et l'implémenté [49, 98]. Pour pouvoir répondre à la question, il doit donc pouvoir passer de la représentation naturelle à la représentation formelle :

$$v_1.taille \geq v_2.taille ?$$

L'objectif du traitement de la *référence structurelle* est de faire correspondre la référence « cube rouge » avec la variable v_1 en utilisant la structure et les valeurs des variables :



Dans le projet *InterViews*, c'est l'utilisation des « sens *WordNet* »² associés à chaque terme de la vue qui permet de traiter la référence structurelle. Dans notre

¹ $\#FF0000$ est la représentation usuelle du rouge pur en notation hexadécimale rouge-vert-bleu (RVB).

²Un « sens *WordNet* » est un identifiant dans la classification *WordNet*. Par exemple, « *(red, 1): red, redness – (the quality or state of the chromatic color resembling the hue of blood)* » pour la couleur rouge.

exemple, si les couleurs sont associés à leur sens *WordNet*³, et s'appuyant sur la description en VDL qui permet de connaître à tout instant de l'exécution la structure statique du composant, le système peut facilement identifier un enregistrement dont les attributs ont des valeurs associées à $(cube, 1)$ et à $(red, 1)$ (en l'occurrence v_1), et donc relier « le cube rouge » à cet enregistrement.

De plus, le mécanisme mis en œuvre dans le MLN permet de faire des correspondances indirectes entre les mots de la phrase et les éléments VDL. Par exemple, si l'utilisateur parle d'un « *bloc rouge* », aucune variable n'a pour valeur *Bloc* dans notre composant. En utilisant la classification *WordNet*, le système étend la recherche de terme pour relier le terme « bloc » de la phrase à une variable dont le sens *WordNet* est « le plus proche possible » d'un sens de « bloc ». La mesure de distance peut être calculée de différentes manières, mais une première méthode simple consiste à partir de *bloc* et à rechercher parmi l'ensemble de ses synonymes, hypéronymes et hyponymes, en réitérant jusqu'à aboutir à un sens correspondant à celui d'une variable dans le composant, chaque itération ayant un coût de 1.⁴

Référence spatiale

Dans les composants munis d'une GUI, le traitement de la référence peut aussi faire appel au positionnement des objets dans l'interface; nous parlons alors de référence spatiale. Par exemple, l'utilisateur peut parler de « *l'objet qui est au dessus du cube rouge; l'objet du haut; etc* ».

Comme pour la référence structurelle, le module de traitement de la référence doit pouvoir interpréter de manière formelle les relations spatiales. Par exemple, « *au dessus de* » peut correspondre à une relation d'ordre sur les coordonnées dans le repère cartésien utilisé pour positionner les objets ($v_1.x \geq v_2.x$). Les outils de conception d'interfaces graphiques faisant de plus en plus appel à des « *widgets* », c'est-à-dire des composants graphiques prédéfinis, le module de traitement de la référence doit pouvoir travailler avec ces outils, et non directement en coordonnées cartésiennes.

Référence sur les actions

Dans cette thèse, nous n'aborderons pas le traitement de la référence structurelle ou spatiale, qui est un problème difficile non encore résolu aujourd'hui, dans le cadre d'un dialogue non contraint. Nous nous intéressons uniquement au fonctionnement du composant. C'est le module de traitement de la référence structurelle, par extension

³Par exemple, $sens(x) = (red, 1)$ si et seulement si $x_r > \#E0$, $x_v > \#2A$ et $x_b > \#2A$ où x_r est le nombre correspondant aux deux premiers chiffres de x ($x_r = x \div \#010000$), x_v les deux suivants ($x_v = (x - x_r) \div \#000100$) et x_b les deux derniers ($x_b = (x - x_r - x_v)$), en base hexadécimale. Autrement dit : beaucoup de composante rouge, pas trop de vert ni de bleu.

⁴Cette mesure de distance n'est pas appropriée puisqu'elle ne prend en compte que les relations sémantiques proposées par *WordNet*. Par exemple, les noms « *church* » et « *bell* » ne sont pas reliés dans *WordNet* alors que le bon sens les associe.

Merci à Jean-Baptiste Berthelin pour cet exemple.

de sens *WordNet*, qui associe les mots de la question de l'utilisateur avec des actions du composant, quand c'est possible.

Par exemple, dans le composant *Coco*, si l'utilisateur demande « *Est-ce que tu comptes ?* », le module de traitement de la référence associe *compter* avec le terme de concept *count*.

Lorsque la correspondance n'est pas possible, le MLN construit une requête dans laquelle les actions sont décrites par un concept représentant le sens *WordNet* extrait de la phrase. Par exemple, si l'utilisateur demande à *Coco* : « *Tu dors ?* », le MLN construira une requête contenant l'action *sleep*, non définie dans *Coco*.

Toute la difficulté pour le MRF est alors de pouvoir construire des réponses formelles pertinentes, en fonction de la requête construite et malgré la présence d'actions non définies.

3.1.1.4 Objectifs

Notre objectif dans ce chapitre est de définir un modèle de requêtes formelles pour faire le lien entre les outils de traitement de la langue naturelle et les formalismes de raisonnement sur les actions dans les composants actifs. Dans ce cadre, nous voudrions pouvoir satisfaire les contraintes suivantes :

1. Le langage de requêtes doit permettre de modéliser la plus large classe possible de questions qu'un utilisateur humain peut poser sur le fonctionnement d'un composant actif : le langage doit être le plus expressif possible ;
2. Le MLN doit pouvoir traduire les questions sur le fonctionnement sous la forme de requêtes : le modèle doit être suffisamment peu complexe pour pouvoir être utilisé par des algorithmes de TALN ;
3. Le MRF doit pouvoir, à partir d'une requête formelle, construire de manière automatique une réponse sous la forme d'une requête, utilisée par le MLN pour produire une réponse en langage naturel : le modèle doit être bien structuré pour permettre de séparer les différents types de traitement possibles.

3.1.2 Un corpus de questions

Afin de proposer un modèle de requêtes aussi expressif que possible (contrainte 1), nous avons constitué un corpus de questions qu'un utilisateur humain peut poser concernant le fonctionnement d'un composant actif simple.

3.1.2.1 Protocole

La méthode employée habituellement pour constituer un tel corpus est le « magicien d'Oz » : l'utilisateur croit interagir avec un système informatique alors qu'il interagit seulement avec une interface contrôlée par un humain (le magicien). L'objectif de cette technique est d'observer le comportement des utilisateurs en présence d'un système avant son implémentation.

Dans notre étude, nous avons proposé à des utilisateurs humains d'interagir avec deux composants simples munis d'une interface graphique :

1. Le composant *Coco*, dont le code en VDL avec l'interface graphique est donnée en annexe page A-1 ;
2. Le composant *Jojo* présenté section 2.4.3.2 page 86.

L'interface Java que nous avons utilisée, présentée au chapitre 7, contient une zone de dialogue dans laquelle l'utilisateur pouvait poser ses questions et lire les réponses données par le magicien, comme le montre la figure 3.1.

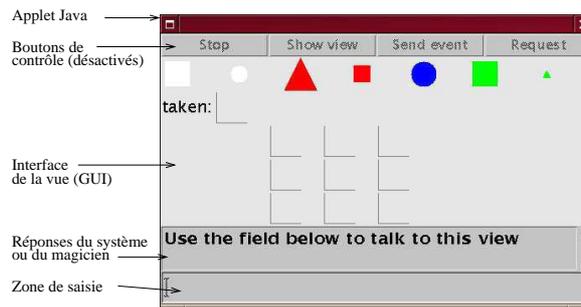


FIG. 3.1 – Interface Java des composants VDL

Dans cette expérience, les composants utilisés sont véritablement des vues implémentées en VDL et en cours d'exécution. En revanche, les questions formulées dans la boîte en langues naturelles sont traitées par le magicien pour :

- Soit envoyer un événement externe au composant ;
- Soit donner une réponse en langue naturelle à l'utilisateur.

Nous mémorisons l'ensemble des questions posées par l'utilisateur, les réponses fournies par le magicien ainsi que tous les événements externes reçus par la vue, ce qui permet de retrouver l'intégralité de l'interaction « effective » avec l'utilisateur⁵. Nous obtenons ainsi un ensemble de « traces d'interaction effectives ».

3.1.2.2 Construction du corpus

Pour constituer notre corpus, nous avons extrait des « traces d'interaction effective » toutes les questions concernant le *fonctionnement* du composant et les réponses envisagées par le magicien. Nous avons aussi ajouté quelques questions, non envisagées par les utilisateurs, qui nous sont venues à l'esprit soit lors de la construction du corpus lui-même, soit plus tard, lors de la conception du MRF.

⁵Nous appelons « interaction effective » toutes les interactions suivies d'effet. Par exemple, si l'utilisateur clique dans l'interface graphique mais en dehors d'un bouton, rien ne se passe et cela n'est pas mémorisé.

Un sous-ensemble représentatif⁶ de ces questions et de ces réponses pour le composant *Coco* est donnée en annexe page A-2. Bien que toutes ces questions portent sur le fonctionnement du composant, nous allons montrer que certaines ne sont pas du ressort du MRF.

Remarque

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 le comportement d'un composant actif simple.

Lors d'autres expériences sur d'autres composants, nous avons vu apparaître d'autres types de questions qui n'ont pas été prises en compte dans l'étude initiale. Nous ne les présentons pas ici mais nous évoquons quelques extensions possibles dans la conclusion de ce chapitre.

3.1.3 Classement des questions du corpus

Les questions proposées dans le tableau en annexe page A-2 ont été regroupées partiellement en respectant l'ordre d'apparition dans le corpus et partiellement de manière à faire ressortir quelques similitudes. Dans l'ordre, nous avons :

- Les questions portant sur les actions du composant ;
- Les questions portant sur les variables ;
- Les commandes ;
- Les questions faisant intervenir des aspects temporels ;
- Les questions un peu absurdes.

Ce regroupement « intuitif » ne définit pas pour autant un classement des questions possibles. Nous allons montrer dans cette section qu'il existe plusieurs classements possibles de ces questions. L'étude de ces différents type de modèles permet de faire ressortir d'une part les éléments de la question en langue naturelle qu'il est nécessaire de représenter dans notre modèle et d'autre part les informations qu'il faut donner au MRF pour définir le type de traitement nécessaire.

3.1.3.1 Commande et contrôle

Dans notre corpus, nous pouvons distinguer :

- Les requêtes de commande (*Arrête-toi !*) ;
- Les questions de contrôle (*Qu'est-ce qu'il se passe ? Qu'est-ce que je peux faire ?*), pouvant elle-même porter sur les commandes possibles.

Du point de vue du TALN, certaines phrases sont ambiguës. Par exemple, si l'utilisateur demande « *Est-ce que tu peux t'arrêter ?* », le MLN ne peut pas nécessairement

⁶Certaines questions sont en effet redondantes. Pour notre étude, il n'est pas nécessaire de les faire toutes figurer. En revanche, l'étude des différentes formulations possibles d'une même question est primordiale pour la conception du MLN.

déterminer s'il s'agit d'une question (*Es-tu capable de le faire ?*) ou d'une commande poliment formulée (*Fais-le*).

Par contre, du point de vue de l'interaction entre une vue et un l'utilisateur, ces deux catégories sont bien séparées : le contrôle correspond à des questions pour lesquelles le système doit fournir une explication, construite à partir de l'analyse du fonctionnement du composant ; la commande correspond l'exécution d'une action dans la vue.

C'est la raison pour laquelle nous avons séparé ces deux problèmes : ces différentes catégories de questions correspondront à différentes catégories de requêtes. Cela signifie qu'elles devront être déterminées par le MLN, en faisant un choix arbitraire lorsqu'il n'arrive pas à déterminer à quelle catégorie appartient une question.

3.1.3.2 Questions de contrôle et de commande

Nous pouvons classer les questions de notre corpus de quatre manières différentes :

1. En fonction du type de fonctionnement considéré par la question. Nous pouvons distinguer dans notre corpus les quatre types suivants :
 - Les question portant sur l'activité du composant (*cf.* section 2.3.4.2 page 63), comme « *Est-ce que tu es en train de compter ?* » ;
 - Les question portant sur le comportement du composant : « *Est-ce que tu peux aller plus vite ?* » ;
 - Les questions portant sur l'état du composant : « *Quelle est ta valeur ?* » ;
 - Les questions portant sur les capacités d'interaction du composant : « *Qu'est-ce que je peux te demander de faire ?* ».
2. En fonction de la classe de mécanisme qu'il faut mettre en œuvre pour construire une réponse. Dans notre corpus, nous avons pu dégager les traitements suivants :
 - Les questions portant sur un élément particulier du composant, comme « *Est-ce que tu comptes ?* ». Le système doit alors analyser l'élément concerné pour construire une réponse ;
 - Les questions ne portant pas sur un élément particulier du composant, comme « *Qu'est-ce que tu fais ?* ». Le système doit alors donner une réponse globale, prenant en compte l'étude du composant dans son ensemble ;
 - Les questions de la forme « *Pourquoi est-ce que tu comptes ?* », correspondent à la construction d'un diagnostic ou d'une explication ;
 - Les questions de la forme « *Comment faire pour t'arrêter ?* », correspondant à la construction d'un plan ;
 - Les questions de la forme « *Quand ais-je changé ta vitesse ?* » pour lesquelles le MRF doit rechercher une date.

Nous pouvons aussi souligner que les commandes correspondent à une autre classe de mécanisme de traitement.

Enfin, nous pouvons remarquer que la plupart de ces questions peuvent être formulées :

- Au présent : « *Est-ce que tu comptes ?* » ;
- Au passé : « *Quand ais-je changé ta vitesse ?* » ;
- Au futur : « *Quelle sera ta valeur dans 3 minutes ?* ».

Et que, lorsqu'elle est exprimée, la « date » de la question peut prendre plusieurs formes :

- Une date directe : « *... dans trois minutes...* ».
- Une date par rapport à un événement : « *... quand j'ai dit stop...* ».
- Une date par rapport à un état de la vue : « *... quand j'ai changé ta vitesse à 2...* ».

Modèle de requêtes

Dans notre modèle de requêtes, nous devons donc pouvoir prendre en compte à la fois le type de fonctionnement considéré, le traitement nécessaire pour la construction de la réponse et la possibilité de représenter des questions portant sur le passé ou le futur de la vue.

3.1.3.3 Commandes

Lorsque le système reçoit une commande de l'utilisateur (formulée en langue naturelle), il doit construire un événement externe à partir de l'analyse de la commande. Dans notre corpus, nous pouvons mettre en évidence deux types de commandes :

1. Les commandes *prévues* ou *indirectes*, correspondant à l'envoi d'un événement externe. Par exemple : « *Va plus vite !* » doit être interprété comme l'envoi de l'événement externe *faster* ;
2. Les commandes *imprévues* ou *directes*, ne correspondant à aucun événement externe mais pouvant s'interpréter en une ou plusieurs actions élémentaires à effectuer sur la vue. Par exemple : « *Recommence à zéro !* » peut être interprété comme l'action VDL `put[path[value], 0]` dans la vue *Coco*.

Commandes indirectes

La construction des événements externes dans le cas des commandes indirectes est du ressort du MLN, dès lors qu'il connaît la liste des événements externes possibles. Le MLN doit détecter que la phrase de l'utilisateur est une commande et qu'elle correspond à un événement externe prévu dans la vue. Dans un cas simple comme *Coco*, ce problème de référence est relativement simple à traiter. En revanche, dans un composant comme *Jojo*, il faut pouvoir construire un événement externe « bien formé », qui sera traité par la vue. Par exemple, si l'utilisateur commande : « Prends le carré rouge » qui doit être traduite en `take[t1]` où t_1 est le terme obtenu par traitement de la référence structurelle « carré rouge » dans la vue :

$$t_1 = \text{figure}[\text{shape}[\text{square}], \text{color}[\text{red}], \text{size}[\text{small}]]$$

Dans une version plus structurée de VDLp, les termes décrivant des événements externes sont munis d'informations sur la structure des événements, utilisées par le MLN pour le traitement de la référence.

Dans notre étude, nous considérerons que les événements externes envoyés par le MLN dans les requêtes formelles sont bien formés.

Commandes directes

La construction d'une commande directe, c'est-à-dire d'une ou plusieurs action(s) VDL à partir d'une commande en langue naturelle est un problème qui est à la fois du ressort du MLN, pour le traitement de la référence par exemple, et du ressort du MRF, puisqu'elle nécessite des connaissances sur la sémantique opérationnelle des actions VDLp.

Par exemple, supposons que l'utilisateur demande à *Coco* : « *Met ta valeur à 0!* ». Le MLN peut déterminer qu'aucun événement externe ne correspond à cette commande, et donc qu'il est nécessaire de construire une action spécifique. Il peut aussi traiter la référence au terme *value* dans la phrase. Par contre, construire l'instruction `put[path[value], 0]` est un problème autrement plus complexe.

De plus, dans le cas général, les commandes ne se réduisent pas à de simples actions élémentaires. Par exemple, si l'utilisateur dit : « Je veux que tu recommence à 0 quand tu arrives à 10 », il faut *ajouter l'action suivante dans la vue* :

```
reset[ guard["greater than"[get[value],10]],  
        put[path[value],0] ]
```

C'est-à-dire qu'il faut effectuer l'action élémentaire `put[path[view], reset[...]]`.

Dans notre thèse, nous n'avons pas traité le problème des commandes directes, en raison de la difficulté de la construction d'une requête formelle contenant une action VDL bien formée (du point de vue de la sémantique opérationnelle ϕ_{exec} , définie dans le chapitre précédent). Cependant, le modèle de requêtes que nous proposons permet aussi de représenter des commandes directes.

3.1.3.4 Conclusion

Notre objectif est de proposer un modèle de requêtes qui permette à la fois de modéliser toutes les questions de notre corpus et de prendre en compte l'étude du fonctionnement par le MRF. L'étude de ces différents types de classements possibles des questions du corpus permet de mettre en évidence des *traits de caractères* dans les questions, c'est-à-dire des éléments caractéristiques permettant de catégoriser la question.

C'est la raison pour laquelle nous proposons dans la section suivante un modèle de requête dans lequel chaque trait de caractère est associé à un critère particulier du langage de requête.

3.1.4 Modèles de requêtes existants

Nous pouvons mettre en évidence plusieurs grandes classes de modèles de requêtes existant.

3.1.4.1 Approche « logique »

Dans les modèles de représentation des connaissances fondés sur la logique, les requêtes sont généralement des expressions syntaxiquement correctes du langage de représentation des connaissances contenant des variables libres. Par exemple, en PROLOG, nous écrivons :

```
pere(paul,marie).      Connaissance : Paul est le père de Marie.
:- pere(X,marie)      Requête : Qui est le père de Marie ?
```

Le traitement des requêtes consiste alors en l'instanciation des variables libres de la requête ($X = paul$ dans notre exemple).

L'avantage de cette approche est que le langage de requête est aussi expressif que le langage de représentation des connaissances (puisque c'est le même). L'inconvénient est que, pour construire une requête, l'utilisateur (agent ou humain) doit connaître exactement la syntaxe et la sémantique du langage. Cette approche est donc particulièrement adaptée pour retrouver dans une base de connaissances l'ensemble des données qui vérifient des contraintes données, exprimées dans la requête [80].

Dans notre schéma d'interaction, le MLN ne connaît pas la sémantique des termes VDLp. Nous ne pouvons donc pas utiliser l'approche logique pour notre modèle de requête. Cependant, nous conserverons la possibilité, pour décrire les éléments dans la requête, d'utiliser des termes VDLp dont la syntaxe et la sémantique sont bien définis.

3.1.4.2 Approche « SQL »

Dans le domaine des bases de données relationnelles, le langage de requêtes SQL, fondé sur la théorie des ensembles, s'est imposé comme modèle de référence. Cette approche est caractérisée par la définition d'un langage orthogonal au modèle de représentation des connaissances et puissant, aussi bien en expressivité qu'en efficacité. C'est aussi l'approche utilisée par le langage OQL⁷, utilisé maintenant dans les bases de données orientées objet.

L'avantage de cette approche est de disposer d'un langage complet (permettant d'exprimer toutes les requêtes envisagées). Son inconvénient réside dans la difficulté à définir les algorithmes de traitement, liée à la complexité syntaxico-sémantique du modèle.

⁷Le modèle OQL utilise aussi des éléments de l'approche logique puisqu'il est possible d'exprimer des structures de données orientées objet (c'est-à-dire de la même nature que les données de la base) et des variables quantifiées existentiellement (par l'opérateur `exists`) dans un champ `select`.

Nous avons envisagé initialement d'utiliser cette approche pour définir notre modèle de requêtes, en nous appuyant sur la syntaxe VDL, c'est-à-dire sur une arborescence de concepts. Cela nous a conduit à proposer le langage orthogonal et complet appelé VQL pour *View Query Language*. Une requête VQL est un arbre de concepts, ce qui nous permet de proposer au moins une modélisation pour toute question en langage naturel. Par exemple, la question : « *Est-ce que tu comptes ?* » se représente en VQL par :

```
Ask[do[view,count]]
```

Toutefois, ce modèle est source d'une grande complexité syntaxico-sémantique, tant au niveau de la construction des requêtes par le MLN qu'au niveau de leur traitement par le MRF. C'est la raison pour laquelle nous avons retenu un autre modèle de requêtes, de type *vecteur*.

3.1.4.3 Modèles « vecteur »

Les modèles de type « vecteurs » sont plus ad-hoc que l'approche logique ou que les langages complets comme SQL, mais ils se rencontrent dans de nombreuses applications lorsqu'il s'agit de faire communiquer deux modules séparés, pour structurer les données échangées.

Dans ces modèles, la requête est un enregistrement de valeurs typées, attendues par le module de traitement de la requête. L'inconvénient de ce type de modèle est que, si l'on souhaite avoir un langage suffisamment expressif, certaines combinaisons de valeurs dans l'enregistrement sont inconsistantes. En revanche, les requêtes sont plus faciles à construire que dans les autres modèles (seule la syntaxe du modèle vecteur est nécessaire) et à traiter (chaque critère correspondant à un élément du traitement).

L'analyse des questions qu'un utilisateur humain peut poser concernant les actions et le fonctionnement d'un composant actif et en particulier l'étude des différents caractères présents correspondant aux classements possibles de ces requêtes nous permet de proposer un modèle de requêtes de type vecteur, dans lequel chaque trait de caractère correspond à un champ de la requête, qui est à la fois :

- Suffisamment expressif pour représenter les différentes questions de notre corpus ;
- Suffisamment simple pour que le MLN puisse construire ces requêtes de manière automatique, en utilisant des techniques de TALN ;
- Et qui permet de prendre en compte l'étude du fonctionnement par le MRF pour un traitement efficace des requêtes.

3.1.4.4 Expressions régulières

Pour compléter la présentation des modèles de requêtes existants, nous devons parler aussi des modèles utilisant les expressions régulières pour définir les éléments à rechercher dans un ensemble de données. Par exemple, de nombreux langages de

commande de systèmes d'exploitation utilisent des expressions régulières dans les noms de fichiers à rechercher.

Toutefois, les requêtes possibles dans ces modèles se limitent à la recherche d'un sous-ensemble dans un ensemble donné. La syntaxe et la sémantique du langage de représentation des connaissances doit donc être connue par l'utilisateur du langage de requêtes. C'est la raison pour laquelle nous ne pourrions pas utiliser d'expressions régulières dans notre modèle de requêtes. En revanche, nous verrons au chapitre suivant que l'utilisation de l'opérateur de subsomption nous permet d'utiliser des sous-arbres VDL comme élément de nos requêtes pour définir un sous-ensemble des termes possibles.

3.1.4.5 Conclusion

Nous avons donc choisi pour notre modèle de requêtes d'utiliser une approche « vecteur » dans laquelle certains éléments peuvent être des termes VDL, munis d'une sémantique opérationnelle et explicationnelle bien définie.

Dans notre modèle d'interaction, les réponses fournies par le MRF sont des requêtes formelles construites suivant *le même modèle*, que nous présentons dans la section suivante.

3.2 Présentation du modèle de requêtes

Dans cette section, nous présentons le modèle de requêtes utilisé par le MLN pour représenter les questions de l'utilisateur et par le MRF pour construire des réponses formelles à ces questions.

3.2.1 Syntaxe

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

$$\rho = \langle \alpha, \tau, \nu, \sigma, \omega, \delta \rangle$$

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.

Les critères α et τ correspondent aux deux traits de caractères mis en évidence par l'analyse des classements possibles pour les questions de contrôle et de commande. Les valeurs possibles pour ces deux critères proviennent des travaux effectués initialement sur VQL : elles ont été choisies à partir des modèles de questions associées à la valeur dans le classement correspondant.

Nous présentons dans les sous-sections suivantes chaque critère plus en détail.

3.2.2 L'acte : α

Le critère α est *l'acte* de la requête. De même que les *actes de langage* (au sens de Austin et Searle [129, 130], en traitement de la langue naturelle) définissent les prédicats opératifs des illocutions dans un dialogue, l'acte de requête définit donc le cadre opératif de la question. Chaque acte de requête correspond donc à un mécanisme de construction de réponse différent.

Dans notre modèle, l'acte α peut prendre les valeurs suivantes :

- *What* pour indiquer que la question ne porte pas sur un élément particulier de la vue ou, plus généralement, que l'objet de la question est inconnu. Par exemple : « *Que fais-tu ?* » ou « *Que comptes-tu ?* »⁸ ;
- *Ask* pour indiquer que la question porte sur un élément particulier, c'est-à-dire que l'objet de la question est donné par le critère ω ⁹. Par exemple : « *Est-ce que tu vaux 42 ?* » ou « *Est-ce que tu comptes ?* » ;
- *Why* pour les demandes de *diagnostic* et d'explication concernant 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 d'explication sur le fonctionnement (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 ?* ».

Et pour les réponses construites par le MRF, nous avons les actes :

- *Assert* pour affirmer un résultat. Par exemple : « *Je compte.* » ;
- *Error* pour signifier que la requête fournie est syntaxiquement incorrecte ;
- *Unknown* pour indiquer qu'aucun sous-terme de la vue ne correspond à l'un des arguments proposés dans le critère ω ($\forall t' \in \omega, t' \notin \text{attributs}(v_t)$), c'est-à-dire que l'objet de la question (donné ou recherché selon l'acte de la requête-question) n'a pas pu être trouvé.

Enfin, pour les commandes (directes ou indirectes), nous utiliserons l'acte :

- *Order* pour modéliser les commandes.

3.2.3 Le type : τ

τ est le *type procédural* de la requête. Il définit le type de fonctionnement considéré par la question. Dans notre modèle, le type τ peut prendre les valeurs suivantes :

- *is* pour indiquer que la question porte sur un l'état du composant ; par exemple :

⁸Une analyse plus poussée met en évidence qu'il existe deux actes *What* différents : *What*₁ dans lequel l'objet est inconnu et *What*₂ dans lequel l'action objet de la question est connue, mais les objets de cette action restent à déterminer. Dans notre modèle de requêtes, compte tenu de la structure et des valeurs choisies pour les critères, nous pouvons regrouper ces deux actes en un seul.

⁹Cette classe de question correspond en KQML [140] à l'acte performatif « *ask-if* ».

- La question de contrôle « *Pourquoi vaux-tu 47 ?* » sera modélisée par une requête dans laquelle $(\alpha, \tau) = (Why, is)$;
- La commande directe « *Met ta valeur à 0.* » sera modélisée par une requête dans laquelle $(\alpha, \tau) = (Order, is)$;
- *do* pour indiquer que la question porte sur l'activité de la vue ; par exemple :
 - La question « *Comment faire pour que tu comptes ?* » sera modélisée par une requête dans laquelle $(\alpha, \tau) = (How, do)$;
 - La commande directe « *Je veux que tu t'arrêtes quand tu arrives à 10.* » sera modélisée par une requête dans laquelle $(\alpha, \tau) = (Order, do)$;
- *order* pour indiquer que la question porte sur les capacités d'interaction du composant ; par exemple :
 - La question « *Qu'est-ce que je peux te demander de faire ?* » sera modélisée par une requête dans laquelle $(\alpha, \tau) = (What, order)$;
 - La commande indirecte « *Va plus vite !* », correspondant à l'envoi de l'événement externe *faster*, sera modélisée par une requête dans laquelle $(\alpha, \tau) = (Order, order)$;
- *can* pour indiquer que la question porte sur le comportement de la vue. Il n'est utilisé que pour les questions (de contrôle ou de commande). Par exemple, la question « *Est-ce que tu peux compter ?* » sera modélisée par une requête dans laquelle $(\alpha, \tau) = (Ask, can)$.

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

3.2.4 Le booléen ν

Le critère $\nu \in \{\top, \perp\}$ est un marqueur booléen, utilisé uniquement dans les questions de contrôle pour de modéliser les question portant sur des « non-actions » ou des « non-états ». Sa sémantique exacte dépend du cadre de la question, défini par (α, τ) .

Par exemple, lorsque $\tau = do$ (questions portant sur l'activité du composant), ν utilisé pour préciser si la question porte sur des processus actifs ($\nu = \top$) ou inactifs ($\nu = \perp$). Ainsi, la question « *Pourquoi est-ce que tu ne comptes plus ?* » sera modélisée par une requête dans laquelle $\nu = \perp$.

3.2.5 Le sujet : σ

$\sigma \in \mathcal{P}(\Upsilon)$ est une référence VDL dont le résultat est le sous-terme de la vue sur lequel porte la question, appelé *sujet* de la question. Par exemple, pour « *Est-ce que ta valeur est 47 ?* », $\sigma = \{value\}$.

C'est le module de traitement de la référence dans le MLN qui est sensé garantir l'unicité et l'exactitude de σ , en utilisant les mécanismes de désambiguïsation en contexte présentés section 3.1.1.3. Nous verrons au chapitre suivant qu'en cas d'échec, le MRF considère la requête comme mal formée et retourne une requête $\langle Unknown, is, \sigma, \emptyset, \surd \rangle$.

Pour les commande, le sujet est l'événement externe envoyé à la vue.

Enfin, pour les réponses construites par le MRF dans les cas d'erreur, σ contient un terme atomique définissant le type d'erreur rencontrée. Par exemple :

$$\langle Error, is, \{ "Not Yet Implemented" \}, \emptyset, \surd \rangle$$

pour indiquer au MLN que la fonctionnalité demandée au MRF n'est pas encore implémentée.

Remarque :

Les requêtes, qu'il s'agisse de questions 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.

3.2.6 L'objet : ω

$\omega \in \mathcal{P}(\Upsilon)$ est un ensemble de termes VDL correspondant aux « paramètres » de la question (commande ou contrôle). Ils sont construits par le MLN et ne sont pas nécessairement présents dans la vue elle-même.

Pour les requêtes de type $\tau = is$, c'est-à-dire portant sur l'état du composant, le sujet est une variable et l'objet ω de la requête donne un ensemble de valeurs possibles. Par exemple, pour modéliser la commande « *Mets ta valeur à 47!* », nous aurons $(\sigma, \omega) = (\{value\}, \{47\})$.

Pour les requêtes portant sur les actions et le fonctionnement (c'est-à-dire telles que $\tau \in \{do, can, order\}$), le sujet est le sous-terme de la vue dans lequel l'action est recherchée (généralement, c'est la vue elle-même) et l'objet donne un ensemble d'actions possibles dans ce sous-terme. Par exemple, pour modéliser la question de contrôle « *Tu comptes ou tu dors ?* », nous aurons $(\sigma, \omega) = (\{view\}, \{count, sleep\})$.

3.2.7 La date : δ

La plupart des questions de contrôle sur le fonctionnement peuvent porter sur une date présente, passée ou même future de l'exécution du composant. Nous avons retenu plusieurs formes pour les dates :

- $\delta = \surd$ représente le temps présent ; par exemple, c'est la valeur qui sera utilisée pour modéliser la question « *Est-ce que tu comptes ?* » ;
- $\delta = \Delta$ où $\Delta \in \{past, future\}$ pour préciser dans les requêtes que la question porte sur le passé ou le futur de la vue ;

Par exemple, la question « *Quand est-ce que va valeur sera 47 ?* » sera modélisée par une requête dans laquelle $\delta = future$;

- $\delta = global[x]$ (où $x \in \mathbf{N}$) représente une date exprimée en nombre de cycles depuis la création de la vue,

- $\delta = date[x]$ (où $x \in \mathbf{Z}$) représente une date dans le passé (si $x < 0$), le présent ($x = 0$) ou le futur ($x > 0$), exprimée en cycles d'exécution ;
Par exemple, la question « *Que faisais-tu il y a trois minutes ?* » sera modélisée par une requête dans laquelle $\delta = date[-\frac{3}{T}]$ où T est la période d'un cycle d'exécution ;
- $\delta = date[x, \rho]$ (où $x \in \mathbf{Z}$ et ρ une requête avec l'acte $\alpha = Assert$ et une date $\delta \in \{past, future\}$) représente une date décalée de x cycles par rapport à la date associée à la requête ρ ;
Par exemple, la question « *Que faisais tu 3 minutes avant que ta valeur soit 24 ?* » sera modélisée par une requête dans laquelle :

$$\delta = date[-\frac{3}{T}, \langle Assert, is, \{value\}, \{24\}, past \rangle]$$

Contraintes

Les requêtes d'acte $\alpha = When$ auront toujours une date $\delta \in \{past, future\}$. Les commandes et les questions d'acte $\alpha = How$ auront toutes une date $\delta = \surd$.

3.2.8 Notions de requête multiple

Certaines questions de l'utilisateur concernant le fonctionnement du composant actif ou certaines réponses souhaitées (en particulier pour les demandes de plan ou de diagnostic) ne peuvent pas être modélisées par une seule requête. Par exemple, la réponse « *Non, je ne compte plus parce que je suis arrêté.* » se représente à l'aide de deux requêtes : ρ_1 qui modélise « *Je ne compte pas* » et ρ_2 qui modélise « *Je suis arrêté* ».

Nous parlons alors de *requête multiple*.

Les éléments de notre modèle de requêtes sont donc des listes de requêtes, généralement réduites à un singleton. Nous notons $\{\rho_1, \dots, \rho_n\}$ une requête contenant n tuples-requêtes et lorsque $n = 1$, nous noterons simplement ρ_1 pour $\{\rho_1\}$. Par abus de langage, nous appellerons requête un élément ρ_i et requête multiple une liste de requêtes.

3.3 Modélisation des questions du corpus

Toutes les questions de notre corpus sont représentables en utilisant le modèle de requêtes proposé dans la section précédente, puisque c'est à partir de l'étude de ce corpus que nous l'avons défini. Nous avons fait en sorte que la construction des requêtes par le MLN soulève le moins de problème possible. Cependant, plusieurs problèmes persistent, outre le traitement de la référence structurelle et spatiale.

Dans cette section, nous discutons l'expressivité de notre langage de requêtes. Nous donnons tout d'abord les requêtes correspondant à chacune des questions et

réponses proposées dans le tableau représentatif donné en annexe page A-2. Nous présentons à chaque fois les problèmes de traitement auxquels le MLN est confronté et nous indiquons le mécanisme de construction de réponse mis en œuvre par le MRF. Nous montrons ensuite quelques exemples de questions sur le fonctionnement qui ne sont pas représentables dans notre modèle.

Nous montrons ainsi que notre modèle de requêtes est bien adapté à notre problématique de construction de réponses aux questions qu'un utilisateur humain peut poser concernant le fonctionnement d'un composant actif.

Structure des requêtes

Nous utiliserons la notation suivante (où n donne le numéro de la requête dans le tableau) :

n . **Question de l'utilisateur en langue naturelle.** *Réponse du composant.*

Q : requête-question
R : requête-réponse

Éventuellement, remarques concernant le MRF et le MLN.

Les requêtes seront données de la manière suivante :

α =acte, τ =type, σ ={référence},
 ω ={liste d'objets}, ν =booléen, δ =date

Et les requêtes multiples seront données entre accolades

3.3.1 Requêtes d'acte $\alpha \in \{Ask, What\}$ au présent

3.3.1.1 Requêtes de type $\tau = is$

Ces questions ne sont pas à proprement parler des questions sur le fonctionnement du composant. Toutefois, leur interprétation nécessite des connaissances sur la sémantique opérationnelle de VDLp. Elles sont donc du ressort du MRF :

4. **Qui es-tu ?** *Je suis Coco le compteur.*

Q : $\alpha=What$, $\tau=is$, $\sigma=\{view\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\surd$
R : $\alpha=Assert$, $\tau=is$, $\sigma=\{view\}$, $\omega=\{ "Coco the counter" \}$,
 $\nu=\top$, $\delta=\surd$

Nous verrons au chapitre suivant que, lorsqu'un terme t est muni d'un attribut de concept *definition*, c'est cet attribut qui est utilisé pour réponse aux questions de type $\tau = is$ dont le sujet est le terme t .

17. **Quelle est ta valeur ?** 47.

Q : $\alpha=What$, $\tau=is$, $\sigma=\{value\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\surd$
R : $\alpha=Assert$, $\tau=is$, $\sigma=\{value\}$, $\omega=\{47\}$, $\nu=\top$, $\delta=\surd$

Dans le cas général, le MRF doit donner l'attribut de la variable référencée par σ . Le MLN peut produire une réponse directe (« *Ma valeur est 47.* ») ou une réponse plus naturelle du point de vue du dialogue homme-machine (« *47.* »).

23. **A quelle vitesse vas-tu ?** *Ma vitesse est 3.*

Q : $\alpha=\text{What}$, $\tau=\text{is}$, $\sigma=\{\text{speed}\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\sqrt{\quad}$
 R : $\alpha=\text{Assert}$, $\tau=\text{is}$, $\sigma=\{\text{speed}\}$, $\omega=\{3\}$, $\nu=\top$, $\delta=\sqrt{\quad}$

Pour construire cette requête, le MLN n'utilise pas de connaissance sur le fonctionnement du composant. L'heuristique mise en œuvre consiste à déterminer que c'est une valeur qui est demandée et qu'un terme de concept associé au sens *WordNet* à (*speed*, 1) est présent dans la vue.

18. **Est-ce que ta valeur est 52 ?** *Non. Ma valeur est 47.*

Q : $\alpha=\text{Ask}$, $\tau=\text{is}$, $\sigma=\{\text{value}\}$, $\omega=\{52\}$, $\nu=\top$, $\delta=\sqrt{\quad}$
 R : { $\alpha=\text{Assert}$, $\tau=\text{is}$, $\sigma=\{\text{value}\}$, $\omega=\{52\}$, $\nu=\perp$, $\delta=\sqrt{\quad}$
 $\alpha=\text{Assert}$, $\tau=\text{is}$, $\sigma=\{\text{value}\}$, $\omega=\{47\}$, $\nu=\top$, $\delta=\sqrt{\quad}$ }

La réponse est composée de deux requêtes : la première modélisant « *ma valeur n'est pas 52* » et la seconde : « *ma valeur est 47* ». Le MLN peut alors produire une réponse directe (« *Ma valeur est 47.* ») ou une réponse plus naturelle du point de vue du dialogue homme-machine (« **Non**, *ma valeur est 47.* ») en utilisant l'acte de la requête-question $\alpha = \text{Ask}$ et en comparant l'objet de la question avec celui de la réponse.

19. **Est-ce que ta valeur est plus grande que 10 ?** *Oui.*

Q : $\alpha=\text{Ask}$, $\tau=\text{is}$, $\sigma=\emptyset$, $\omega=\{\text{"greater than"}[\text{get}[\text{value}],10]\}$,
 $\nu=\top$, $\delta=\sqrt{\quad}$
 R : $\alpha=\text{Assert}$, $\tau=\text{is}$, $\sigma=\emptyset$, $\omega=\{\text{"greater than"}[\text{get}[\text{value}],10]\}$,
 $\nu=\top$, $\delta=\sqrt{\quad}$

Pour les questions d'état dans lesquelles l'objet de la question est une relation entre des termes, plusieurs problèmes se posent :

- La relation peut être directement exprimée et évaluée en VDL, en utilisant l'interprétation canonique sur les termes ;
- La question peut comporter plusieurs sujets (dans notre exemple : {*value*} et {10}) ;
- Ces sujets doivent être donnés dans le relation VDL pour qu'elle puisse être évaluée.

C'est pourquoi nous aurons $\sigma = \emptyset$ pour ces requêtes. Le MRF utilisera cet élément pour savoir qu'il lui faut non pas rechercher une valeur dans la vue mais évaluer une relation VDL.

Pour construire la requête, le MLN utilise la sémantique explicationnelle des opérateurs arithmétiques VDLp.

3.3.1.2 Requêtes de type $\tau = do$

Ce sont les questions portant sur l'activité de la vue :

1. **Qu'est-ce que tu fais ?** *Je compte.*

Q : $\alpha=What, \tau=do, \sigma=\{view\}, \omega=\emptyset, \nu=\top, \delta=\surd$
R : $\alpha=Assert, \tau=do, \sigma=\{view\}, \omega=\{count\}, \nu=\top, \delta=\surd$

Lorsque $\omega = \emptyset$, le MRF doit rechercher les processus actifs dans la vue. Si aucun processus actif n'est trouvé, l'objet de la réponse sera $\omega = \{nothing\}$.

2. **Qu'est-ce que tu comptes ?** *Je compte ma valeur.*

Q : $\alpha=What, \tau=do, \sigma=\{view\}, \omega=\{count\}, \nu=\top, \delta=\surd$
R : $\alpha=Assert, \tau=do, \sigma=\{view\}, \omega=\{count[path[value]]\},$
 $\nu=\top, \delta=\surd$

Lorsque $\omega \neq \emptyset$, le MRF doit rechercher les termes modifiés par les actions données dans ω .

9. **Est-ce que tu es en train de compter ?** *Oui.*

Q : $\alpha=Ask, \tau=do, \sigma=\{view\}, \omega=\{count\}, \nu=\top, \delta=\surd$
R : $\alpha=Assert, \tau=do, \sigma=\{view\}, \omega=\{count\}, \nu=\top, \delta=\surd$

Le MLN utilise les noms des actions, associé à un concept pour construire l'ensemble ω (cf. section 2.3.4.2 page 63).

Comme dans le cas $\tau = is$, le MLN peut construire une réponse directe (« *Je compte.* ») ou une réponse plus naturelle (« *Oui.* »), en comparant l'objet de la question et celui de la réponse.

27. **Tu ne comptes plus ?** *Non. Effectivement.*

Q : $\alpha=Ask, \tau=do, \sigma=\{view\}, \omega=\{count\}, \nu=\perp, \delta=\surd$
R : $\alpha=Assert, \tau=do, \sigma=\{view\}, \omega=\{count\}, \nu=\perp, \delta=\surd$

Cette question illustre l'utilisation du booléen ν . Le MLN doit donc pouvoir déterminer si les questions portent sur l'activité ou l'inactivité du composant.

16. **Tu es arrêté, là ?** *Oui.*

Q : $\alpha=Ask, \tau=do, \sigma=\{view\}, \omega=\{stop\}, \nu=\top, \delta=\surd$
R : $\alpha=Assert, \tau=do, \sigma=\{view\}, \omega=\{stop\}, \nu=\top, \delta=past$

L'action *stop* n'est pas un processus (c'est une réaction). Mais le MLN ne connaît pas la nature des actions (il ne connaît que leur *nom*). C'est le MRF qui détermine que *stop* est une réaction et lance un traitement différent de celui utilisé pour les processus : il doit alors déterminer si le composant est dans un état correspondant au résultat de la réaction. Ce procédé est entièrement transparent pour le MLN, mais la réponse a pour date $\delta = past$ pour indiquer que l'action n'est pas *en train* de s'exécuter (ce n'est pas un processus actif).

11. **Tu es en train de compter ou de dormir ?** *De compter.*

Q : $\alpha=\text{Ask}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{count},\text{sleep}\}$, $\nu=\top$, $\delta=\surd$
R : $\alpha=\text{Assert}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{count}\}$, $\nu=\top$, $\delta=\surd$

Comme nous l'expliquons dans la section 3.3.7.1 page 121, notre modèle de requêtes ne permet pas de différencier les opérations *ou* et *et* utilisées en langage naturel pour associer plusieurs éléments. En effet, ces opérateurs ne sont pas nécessaires pour le traitement de ces requêtes au niveau formel.

Le MRF construit une réponse naturelle en privilégiant les processus actifs. Ce choix a été guidé par l'analyse de notre corpus, dans lequel nous trouvons :

Q : Tu comptes ou tu dors ?
R : Je comptes.
Q : Et tu ne dors pas ?
R : Je ne sais pas ce que dormir signifie.

3.3.1.3 Requêtes de type $\tau = \text{can}$

Les requêtes de type *can* et d'acte *What* sont traitées de manière similaire aux requêtes de type *do* :

32. **Qu'est-ce que tu pourrais faire ?** *Je peux compter.*

Q : $\alpha=\text{What}$, $\tau=\text{can}$, $\sigma=\{\text{view}\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\surd$
R : $\alpha=\text{Assert}$, $\tau=\text{can}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{count}\}$, $\nu=\top$, $\delta=\surd$

En revanche, les requêtes d'acte *Ask* sont porteuses d'une ambiguïté. En effet, les requêtes et de de type *can* portent sur le comportement du composant, c'est-à-dire sur les actions *possibles*. Or avoir la possibilité de 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. Nous verrons au chapitre suivant que la recherche d'action dans la vue permet en effet de déterminer quelles sont les actions connues et que le calcul des préconditions permet à la fois de déterminer quelles sont les actions possibles, pour *définir le cadre de la question* (physique ou moral)

21. **Est-ce que tu peux compter ?** *Non parce que je ne suis pas en marche.*

Q : $\alpha=\text{Ask}$, $\tau=\text{can}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{count}\}$, $\nu=\top$, $\delta=\surd$
R : { $\alpha=\text{Assert}$, $\tau=\text{can}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{count}\}$, $\nu=\perp$, $\delta=\surd$
 $\alpha=\text{Assert}$, $\tau=\text{is}$, $\sigma=\{\text{running}\}$, $\omega=\{\text{true}\}$, $\nu=\perp$, $\delta=\surd$ }

Lorsque la réponse est négative ($\nu = \perp$), le MRF construit une requête multiple telle que chaque $\rho_{i \geq 2}$ représentent une précondition invalide de l'action, comme dans l'exemple ci-dessus. Au contraire, lorsque la réponse est positive ($\nu = \top$), la réponse est une requête simple.

Ce mécanisme permet de désambiguïser la possibilité physique et la possibilité morale.

10. **Est-ce que tu peux compter moins vite ?** *Oui. Tu veux ?*

Q : $\alpha=\text{Ask}$, $\tau=\text{can}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{"slow down"}\}$, $\nu=\top$, $\delta=\surd$
 R : $\alpha=\text{Assert}$, $\tau=\text{can}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{"slow down"}\}$, $\nu=\top$, $\delta=\surd$

La requête peut aussi être modélisée de la manière suivante :

Q : $\alpha=\text{Ask}$, $\tau=\text{can}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{slower}\}$, $\nu=\top$, $\delta=\surd$

En effet, le MRF peut travailler aussi bien sur les réactions que sur les événements externes. Dans les deux cas, le MLN doit être capable d'associer « *count less fast* » (« *compter moins vite* ») avec soit « *slow down* », soit « *slower* », en utilisant la classification *WordNet* et l'extension de requêtes présentée section 3.1.1.3. Ce problème difficile ne sera pas abordé ici.

Lorsque le MLN associe la question de l'utilisateur avec un événement externe (ici, *slower*), pour obtenir un dialogue plus naturel [83, 67], il peut proposer d'envoyer cet événement externe à la vue (ici : « *Tu veux ?* »).

Impossibilité physique

L'utilisateur peut parfaitement demander à la vue si elle a la possibilité de faire des actions qui ne sont pas définies dans la vue. Dans ce cas, la réponse sera une requête d'acte *Unknown* :

43. **Est-ce que tu peux me faire un café ?** *Je ne sais pas faire un café.*

Q : $\alpha=\text{Ask}$, $\tau=\text{can}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{"make coffee"}\}$,
 $\nu=\top$, $\delta=\surd$
 R : $\alpha=\text{Unknown}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{"make coffee"}\}$,
 $\nu=\top$, $\delta=\surd$

Remarquons que le type de la réponse est $\tau = do$, bien qu'elle ne porte pas uniquement sur l'activité de la vue, mais sur son comportement en général. Nous verrons au chapitre suivant que les requêtes $(\alpha, \tau) = (\text{Unknown}, \text{can})$ sont utilisées pour modéliser d'autres types de réponses.

44. **Est-ce que tu peux augmenter mon salaire ?** *Je n'ai pas de salaire.*

Q : $\alpha=\text{Ask}$, $\tau=\text{can}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{increase}[\text{path}[\text{salary}]]\}$,
 $\nu=\top$, $\delta=\surd$
 R : $\alpha=\text{Unknown}$, $\tau=\text{is}$, $\sigma=\{\text{salary}\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\surd$

Dans cette réponse, $\tau = is$ car ce n'est pas l'action *increase* qui est inconnue (nous verrons au chapitre suivant qu'il s'agit d'une action « de bon sens », connue pour tous les composants), mais bien le terme *salary* objet de cette action. C'est donc sur l'état de la vue que porte l'inconnue.

3.3.1.4 Requêtes de type $\tau = order$

Ces requêtes portent sur les événements externes de la vue :

34. **Qu'est-ce que je peux faire ?** *Tu peux me demander de m'arrêter d'aller plus vite ou moins vite...*

Q : $\alpha=What$, $\tau=order$, $\sigma=\{view\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\surd$
R : $\alpha=Order$, $\tau=order$, $\sigma=\{start,stop,faster,slower\}$,
 $\omega=\emptyset$, $\nu=\top$, $\delta=\surd$

Le MRF donne l'ensemble des événements externes possibles. C'est le MLN qui doit construire une réponse plus naturelle. De plus, nous verrons que pour des composants plus complexes comme *Jojo*, les événements donnés par le MRF ne permettent pas toujours de construire directement une réponse. Le MRF doit alors utiliser des connaissances spécifiques pour donner une réponse naturelle.

Les requêtes d'acte $\alpha = Ask$ correspondent à des demandes de recherche d'événement externes dans les traces d'exécution de la vue. Par exemple (la question n'est pas présente dans notre corpus) :

0. **Est-ce que je t'ai arrêté ?** *Oui, il y a 3 minutes.*

Q : $\alpha=Ask$, $\tau=order$, $\sigma=\{view\}$, $\omega=\{stop\}$, $\nu=\top$, $\delta=past$
R : $\alpha=Assert$, $\tau=order$, $\sigma=\{view\}$, $\omega=\{stop\}$, $\nu=\top$,
 $\delta=date[-300]$

Nous aurons toujours $\delta = past$. La construction de la réponse s'appuie uniquement sur la mémorisation des dates d'événements externes.

3.3.2 Requêtes d'acte $\alpha = How$

Lorsque le type $\tau = do$, les requêtes d'acte *How* modélisent des demandes d'explication sur le corps d'une action¹⁰ :

8. **Comment fais-tu pour compter ?** *J'incrémente ma valeur avec ma vitesse.*

Q : $\alpha=How$, $\tau=do$, $\sigma=\{view\}$, $\omega=\{count\}$, $\nu=\top$, $\delta=\surd$
R : $\alpha=Assert$, $\tau=do$, $\sigma=\{view\}$,
 $\omega=\{count[increase[path[value],get[speed]}\}$,
 $\nu=\top$, $\delta=date[-300]$

Nous verrons au chapitre comme l'action *increase* est construite à partir de la description de l'action *count* dans le corps de la vue.

¹⁰En effet, nous avons considéré que les demandes d'explications de la forme « *comment fais-tu cela* » étaient plus du ressort de l'activité du composant et que les demandes de plan de la forme « *comment je pourrais faire pour...* » étaient plus du ressort du comportement, c'est-à-dire des actions possibles mais pas en cours d'exécution.

Lorsque le type $\tau \in \{can, is\}$, les requêtes d'acte *How* modélisent des demandes de plan :

15. **Comment faire pour t'arrêter?** *Clique sur le bouton « stop ».*

Q : $\alpha=How, \tau=can, \sigma=\{view\}, \omega=\{stop\}, \nu=\top, \delta=\surd$
 R : $\alpha=Order, \tau=order, \sigma=\{stop\}, \omega=\emptyset, \nu=\top, \delta=\surd$

La réponse est une requête multiple donnant une suite d'action permettant d'arriver au résultat décrit dans la question. Le MRF ne donne donc qu'un chemin dans l'espace des états, même si un plan peut décrire plusieurs chemins.

Considérons à titre d'exemple la question suivante (absente de notre corpus) pour $\tau = is$:

0. **Comment faire pour que ta vitesse soit négative?** *Clique deux fois sur le bouton « - ».*

Q : $\alpha=How, \tau=is, \sigma=\emptyset, \omega=\{negative[get[speed]]\}, \nu=\top, \delta=\surd$
 R : $\{ \alpha=Order, \tau=order, \sigma=\{slower\}, \omega=\emptyset, \nu=\top, \delta=\surd$
 $\alpha=Order, \tau=order, \sigma=\{slower\}, \omega=\emptyset, \nu=\top, \delta=\surd \}$

Remarque

Les requêtes d'acte $\alpha = How$ et de type $\tau = order$ sont considérées comme des requêtes mal formées par le MRF. C'est une combinaison « inconsistante » de notre modèle de requêtes.

3.3.3 Requêtes d'acte $\alpha = Why$ au présent

Les questions d'acte $\alpha = Why$ représentent des demandes d'explications. L'algorithme de construction de réponse, commun aux types *is*, *do* et *can*, est présenté au chapitre 4. Une explication dans notre modèle est la description d'une action ou d'un événement externe, par une requête.

La première question, donnée à titre d'exemple pour le cas $\tau = is$, n'est pas issue de notre corpus.

0. **Pourquoi est ce que ta vitesse vaut 2?** *Parce que tu as cliqué sur « + ».*

Q : $\alpha=Why, \tau=is, \sigma=\{speed\}, \omega=\{2\}, \nu=\top, \delta=\surd$
 R : $\alpha=Assert, \tau=order, \sigma=\{view\}, \omega=\{clic[-]\}, \nu=\top,$
 $\delta=date[-150]$

24. **Pourquoi est-ce que tu es arrêté?** *Parce que tu as cliqué sur « stop ».*

Q : $\alpha=Why, \tau=do, \sigma=\{view\}, \omega=\{stop\}, \nu=\top, \delta=\surd$
 R : $\alpha=Assert, \tau=order, \sigma=\{view\}, \omega=\{clic[stop]\}, \nu=\top,$
 $\delta=date[-320]$

3. **Pourquoi tu comptes ?** *Parce que je suis en marche.*

Q : $\alpha=Why$, $\tau=do$, $\sigma=\{view\}$, $\omega=\{count\}$, $\nu=\top$, $\delta=\surd$
R : $\alpha=Assert$, $\tau=is$, $\sigma=\{running\}$, $\omega=\{true\}$, $\nu=\top$, $\delta=\surd$

22. **Pourquoi ne peux-tu pas compter ?** *Parce que je suis arrêté.*

Q : $\alpha=Why$, $\tau=can$, $\sigma=\{view\}$, $\omega=\{count\}$, $\nu=\perp$, $\delta=\surd$
R : $\alpha=Assert$, $\tau=do$, $\sigma=\{view\}$, $\omega=\{stop\}$, $\nu=\perp$, $\delta=past$

La réponse donnée par le MRF n'est pas exactement celle envisagée par le magicien : le MLN produira plutôt la réponse : « *Parce que j'ai été arrêté.* », en utilisant $\delta = past$ dans la requête-réponse.

Dans notre modèle, le MRF peut construire les trois réponses suivantes :

– *Parce que je ne suis pas en marche.*

R : $\alpha=Assert$, $\tau=is$, $\sigma=\{running\}$, $\omega=\{true\}$, $\nu=\perp$, $\delta=\surd$

– *Parce que j'ai été arrêté.*

R : $\alpha=Assert$, $\tau=do$, $\sigma=\{view\}$, $\omega=\{stop\}$, $\nu=\perp$, $\delta=past$

– *Parce que tu as cliqué sur stop.*

R : $\alpha=Assert$, $\tau=order$, $\sigma=\{view\}$, $\omega=clic[stop]$, $\nu=\perp$,
 $\delta=date[-180]$

Nous montrerons que, de manière générale, pour toutes les questions d'acte $\alpha = Why$, le MRF peut construire plusieurs réponses, correspondant chacune à une explication différente.

12. **Pourquoi est-ce que tu ne fais plus rien ?** *Parce que suis arrêté.*

Q : $\alpha=Why$, $\tau=do$, $\sigma=\{view\}$, $\omega=\{nothing\}$, $\nu=\top$, $\delta=\surd$
R : $\alpha=Assert$, $\tau=do$, $\sigma=\{view\}$, $\omega=\{stop\}$, $\nu=\top$, $\delta=past$

La difficulté ici est de passer de la notion de « ne rien faire » au fait que l'action *count* est inactive. Nous montrerons au chapitre 4 comment cela est effectué dans le MRF, de manière transparente pour le MLN.

29. **Pourquoi est-ce que tu n'as pas recommencé à compter ?** *Parce que tu ne l'as pas demandé. Tu veux que je recommence ?*

Q : $\alpha=Why$, $\tau=do$, $\sigma=\{view\}$, $\omega=\{count\}$, $\nu=\perp$, $\delta=\surd$
R : $\alpha=Assert$, $\tau=order$, $\sigma=\{view\}$, $\omega=\{clic[stop]\}$, $\nu=\top$, $\delta=\surd$

La construction de la requête par le MLN pose le problème du traitement de la référence : est-ce que « *recommencer à compter* » doit être associé à *start* ou à *count*. Cette difficulté ne sera pas étudiée ici.

C'est le MLN qui doit proposer d'envoyer l'événement externe à la vue, lorsque le MRF retourne une requête telle que $(\alpha, \tau, \nu) = (Assert, order, \perp)$ ¹¹.

¹¹Remarquons que lorsque la requête-réponse est négative ($\nu = \perp$: l'événement n'a pas été envoyé), nous avons $\delta = \surd$.

14. **Pourquoi est-ce que tu comptes à l'envers ?** *Parce que ma vitesse est négative.*

Q : $\alpha=Why, \tau=do, \sigma=\{view\}, \omega=\{nothing\}, \nu=\top, \delta=\surd$
R : $\alpha=Assert, \tau=is, \sigma=\emptyset, \omega=\{negative[get[speed]]\}, \nu=\top,$
 $\delta=past$

La difficulté dans cette question n'est pas de construire l'explication, comme nous le verrons dans le chapitre suivant, mais de comprendre la notion de « compter à l'envers ».

13. **Pourquoi est-ce que tu ne comptes plus ?** *Je compte mais ma vitesse est nulle. Alors ma valeur augmente de 0.*

Q : $\alpha=Why, \tau=do, \sigma=\{view\}, \omega=\{count\}, \nu=\perp, \delta=\surd$
R : { $\alpha=Assert, \tau=do, \sigma=\{view\}, \omega=\{count\}, \nu=\top, \delta=\surd$
 $\alpha=Assert, \tau=is, \sigma=\{speed\}, \omega=\{0\}, \nu=\top, \delta=\surd$
 $\alpha=Assert, \tau=do, \sigma=\{view\}, \omega=\{increase[path[value],0\},$
 $\nu=\top, \delta=\surd$ }

La construction de cette explication nécessite des connaissances complexes : le MRF doit pouvoir se représenter que si un incrément est nul, alors la valeur ne change pas et que ce non-changement est perçu comme une non-action.

Dans cette thèse, nous ne mettrons pas en œuvre ces connaissances dans le MRF. En revanche, nous pourrions produire la réponse suivante :

– *Si, je compte !*

R : $\alpha=Assert, \tau=do, \sigma=\{view\}, \omega=\{count\}, \nu=\top, \delta=\surd$

Remarque

Les requêtes d'acte $\alpha = Why$ et de type $\tau = order$ sont considérées dans cette thèse comme des requêtes mal formées par le MRF.

3.3.4 Commandes

3.3.4.1 Commandes indirectes

Le MLN peut envoyer des événements externes à la vue en utilisant des commandes indirectes :

25. **Va plus vite ! OK.**

Q : $\alpha=Order, \tau=order, \sigma=\{faster\}, \omega=\emptyset, \nu=\top, \delta=\surd$
R : $\alpha=Order, \tau=order, \sigma=\{faster\}, \omega=\emptyset, \nu=\top, \delta=\surd$

La réponse donnée par le MRF donne l'ensemble des événements externes envoyés. En effet, le MLN peut utiliser non seulement des événements dans les commandes indirectes, mais aussi des *noms* de réactions. Par exemple :

0. Accélère ! OK.

Q : $\alpha=Order, \tau=order, \sigma=\{\text{"speed up"}\}, \omega=\emptyset, \nu=\top, \delta=\surd$
R : $\alpha=Order, \tau=order, \sigma=\{\text{faster}\}, \omega=\emptyset, \nu=\top, \delta=\surd$

26. Arrête-toi ! Stop ! OK.

Q : $\alpha=Order, \tau=order, \sigma=\{\text{stop}\}, \omega=\emptyset, \nu=\top, \delta=\surd$
R : $\alpha=Order, \tau=order, \sigma=\{\text{stop}\}, \omega=\emptyset, \nu=\top, \delta=\surd$

Lorsque deux commandes correspondent au même événement, le MLN n'envoie qu'une seule fois l'événement externe.

33. Tu peux compter moins vite ? OK.

Q : $\alpha=Order, \tau=order, \sigma=\{\text{slower}\}, \omega=\emptyset, \nu=\top, \delta=\surd$
[ou]
Q : $\alpha=Order, \tau=order, \sigma=\{\text{"slow down"}\}, \omega=\emptyset, \nu=\top, \delta=\surd$
R : $\alpha=Order, \tau=order, \sigma=\{\text{slower}\}, \omega=\emptyset, \nu=\top, \delta=\surd$

Comme nous l'avons vu pour les requêtes de la forme $\tau = can$, le fait que le MRF puisse prendre en considération des événements externes ou des actions permet au MLN de construire plus facilement la requête, en utilisant les relations *WordNet*.

3.3.4.2 Commandes directes

Dans cette thèse, nous n'abordons pas le problème de la construction d'actions VDL à partir de commandes directes en langage naturel. C'est la raison pour laquelle le magicien répond « *Je ne sais pas le faire.* ». Cependant, en utilisant le pouvoir d'expressivité de VDLp et en ajoutant des concepts spécifiques, connus par le MLN, pour la représentation des commandes directes, nous pouvons modéliser ces commandes dans notre langage de requêtes :

28. Recommence à zéro ! *Je ne sais pas le faire.*

Q : $\alpha=Order, \tau=is, \sigma=\{\text{reset[path[value]]}\}, \omega=\emptyset,$
 $\nu=\perp, \delta=\surd$
R : $\alpha=Error, \tau=is, \sigma=\{\text{"Not Yet Implemented"}\}, \omega=\emptyset,$
 $\nu=\perp, \delta=\surd$

L'action VDLp que devrait construire le MRF, en utilisant la mémorisation de l'exécution du composant, est alors la suivante :

put[0,path[value]]

La problématique de la construction de la requête formelle à partir de la question ne sera pas abordée.

30. Je veux que tu t'arrêtes quand tu arrives à 10. *Je ne sais pas le faire.*

Q : $\alpha=Order$, $\tau=do$, $\sigma=\{stop\}$, $\omega=\emptyset$, $\nu=\perp$, $\delta=is[value,10]$
 R : $\alpha=Error$, $\tau=is$, $\sigma=\{"Not Yet Implemented"\}$, $\omega=\emptyset$,
 $\nu=\perp$, $\delta=\surd$

L'action VDLp que devrait construire le MRF est alors la suivante :

```

add[ path[view],
      "stop when reach 10"[
        guard["greater than"[get[value],10]],
        put[path[running],false]      ] ]

```

La construction de la requête et de l'action VDLp associée est un problème trop difficile, dans le cas général, pour être étudié dans le cadre de cette thèse.

Lorsque le MLN ou le MRF ne peuvent pas construire de requête ou d'action VDLp, nous considérerons qu'il s'agit d'un cas d'erreur :

6. **Vas-t-en!** *Je ne comprends pas.*

La réponse donnée par le MRF en cas d'échec est la suivante :

R : $\alpha=Error$, $\tau=is$, $\sigma=\{"Unable to Perform"\}$, $\omega=\emptyset$,
 $\nu=\perp$, $\delta=\surd$

3.3.5 Requêtes temporelles

Nous étudions ici la modélisation des questions faisant intervenir le temps, c'est-à-dire telle que $\delta \neq \surd$ ou $\alpha = When$ dans la requête.

35. **Quelle était ta valeur il y a 10 minutes?** *Ma valeur était 45 il y a 10 minutes.*

Q : $\alpha=What$, $\tau=is$, $\sigma=\{value\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=date[-1000]$
 R : $\alpha=Assert$, $\tau=is$, $\sigma=\{value\}$, $\omega=\{45\}$, $\nu=\top$, $\delta=date[-1000]$

36. **Quelle sera ta valeur dans 3 minutes?** *Si tu ne changes rien, ma valeur sera 38.*

Q : $\alpha=What$, $\tau=is$, $\sigma=\{value\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=date[300]$
 R : $\alpha=Assert$, $\tau=is$, $\sigma=\{value\}$, $\omega=\{38\}$, $\nu=\top$, $\delta=date[300]$

Pour les réponse sur le futur, le MLN doit préciser que l'utilisateur ne doit rien changer. Il s'agit en fait d'une faiblesse de notre modèle.

41. **Est-ce que tu comptais il y a 3 minutes?** *Oui.*

Q : $\alpha=Ask$, $\tau=do$, $\sigma=\{view\}$, $\omega=\{count\}$, $\nu=\top$,
 $\delta=date[-300]$
 R : $\alpha=Assert$, $\tau=do$, $\sigma=\{view\}$, $\omega=\{count\}$, $\nu=\top$,
 $\delta=date[-300]$

42. **Qu'est-ce que tu faisais il y a 5 minutes?** *Rien.*

Q : $\alpha=\text{What}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{count}\}$, $\nu=\top$,
 $\delta=\text{date}[-500]$
R : $\alpha=\text{Assert}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{nothing}\}$, $\nu=\top$,
 $\delta=\text{date}[-500]$

40. **Quand ais-je changé ta vitesse ?** *Ma vitesse a été modifiée il y a 7 minutes.*

Q : $\alpha=\text{When}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{modify}[\text{path}[\text{speed}]]\}$,
 $\nu=\top$, $\delta=\text{past}$
R : $\alpha=\text{Assert}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{modify}[\text{path}[\text{speed}]]\}$,
 $\nu=\top$, $\delta=\text{date}[-700]$

Nous verrons au chapitre que l'action *modify* est une action partagée par tous les composants.

Remarquons aussi que le MRF construit toujours une réponse avec une date relative au temps présent.

37. **Quelle était ta valeur 5 minutes avant que je ne dise « stop » ?** *Il y a 8 minutes, ma valeur était 32.*

Q : $\alpha=\text{What}$, $\tau=\text{is}$, $\sigma=\{\text{value}\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\text{date}[-5,\rho]$
avec ρ : $\alpha=\text{Assert}$, $\tau=\text{order}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{stop}\}$, $\nu=\top$, $\delta=\text{past}$
R : $\alpha=\text{Assert}$, $\tau=\text{is}$, $\sigma=\{\text{value}\}$, $\omega=\{32\}$, $\nu=\top$, $\delta=\text{date}[-800]$

Dans les requêtes décrivant une date (ρ), $\tau = \text{is}$ est utilisé pour donner un état, $\tau = \text{do}$ pour une action, $\tau = \text{can}$ pour une précondition et $\tau = \text{order}$ pour l'envoi d'un événement externe.

39. **Combien vaudras-tu quand je changerai ta vitesse ?** *Je ne peux pas le prévoir.*

Q : $\alpha=\text{What}$, $\tau=\text{is}$, $\sigma=\{\text{value}\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\text{date}[0,\rho]$
avec ρ : $\alpha=\text{Assert}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{modify}[\text{path}[\text{value}]]\}$,
 $\nu=\top$, $\delta=\text{future}$
R : $\alpha=\text{Unknown}$, $\tau=\text{do}$, $\sigma=\{\text{value}\}$, $\omega=\{\text{modify}[\text{path}[\text{value}]]\}$,
 $\nu=\top$, $\delta=\text{future}$

Le MRF construit toujours une réponse modélisant « *Je ne sais pas quand cette action sera effectuée.* » lorsque la date de la question est de la forme $\text{date}[x,\rho]$ et $\rho.\tau = \text{do}$ et $\rho.\tau = \text{future}$

38. **Quelle était ta valeur quand j'ai changé ta vitesse à 2 ?** *Il y a 2 minutes, ma valeur était 12.*

Q : $\alpha=\text{What}$, $\tau=\text{is}$, $\sigma=\{\text{value}\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\text{date}[0,\rho]$
avec ρ : $\alpha=\text{Assert}$, $\tau=\text{is}$, $\sigma=\{\text{speed}\}$, $\omega=\{2\}$, $\nu=\top$, $\delta=\sqrt{\quad}$
R : $\alpha=\text{Assert}$, $\tau=\text{is}$, $\sigma=\{\text{value}\}$, $\omega=\{12\}$, $\nu=\top$, $\delta=\text{date}[-200]$

Lorsque plusieurs dates ou intervalles de dates correspondent à la date exprimée dans la requête, le MRF choisit la première date du dernier intervalle :



Le MLN doit utiliser cette propriété pour construire la date de la requête.

Enfin, les demandes de diagnostic peuvent être formulées en utilisant des requêtes au passé. Par exemple :

20. **Pourquoi est-ce que la valeur a changé ?** *Parce que je suis en marche.*

Q : $\alpha=\text{Why}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{modify}[\text{path}[\text{value}]]\}$
 $\nu=\top$, $\delta=\text{past}$

R : $\alpha=\text{Assert}$, $\tau=\text{is}$, $\sigma=\{\text{running}\}$, $\omega=\{\text{true}\}$, $\nu=\top$, $\delta=\text{past}$

La réponse fournie par le MRF en utilisant le mécanisme de diagnostic n'est pas aussi bonne que celle fournie par le magicien (« *Parce que je compte.* »). Cependant, elle constitue une bonne explication de l'action.

3.3.6 Requêtes non traitées

Trois questions du tableau donné en annexe page A-2 ne sont pas du directement du ressort du MRF. Il s'agit de question qui peuvent sembler parler du fonctionnement du composant, mais qui en réalité sont des questions générales, sur le composant en général.

Nous avons choisi de ne pas pouvoir les traiter, mais elles sont tout de même représentables dans notre modèle de requêtes : premières :

5. **Pourquoi es-tu là ?** *Je ne sais pas.*

Q : $\alpha=\text{Why}$, $\tau=\text{is}$, $\sigma=\{\text{view}\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\surd$

7. **Tu sers à quoi ?** *A compter.*

Q : $\alpha=\text{Why}$, $\tau=\text{is}$, $\sigma=\{\text{view}\}$, $\omega=\emptyset$, $\nu=\top$, $\delta=\surd$

La requête suivante est plus problématique : si le MLN ne détecte pas qu'il s'agit d'une question extérieure au composant, il produit une requête de demande d'explication :

31. **Pourquoi fais-tu tout ce que je dis ?** *Je suis là pour ça.*

Q : $\alpha=\text{Why}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{everything}\}$, $\nu=\top$, $\delta=\surd$

R : $\alpha=\text{Assert}$, $\tau=\text{is}$, $\sigma=\{\text{running}\}$, $\omega=\{\text{true}\}$,
 $\nu=\top$, $\delta=\surd$

Le MRF construit alors une réponse inappropriée : « *Parce que je suis en marche* ». Si *Coco* avait été arrêté, le MRF aurait construit la réponse :

R : { $\alpha=\text{Assert}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{everything}\}$, $\nu=\perp$,
 $\delta=\surd$
 $\alpha=\text{Assert}$, $\tau=\text{do}$, $\sigma=\{\text{view}\}$, $\omega=\{\text{count}\}$, $\nu=\perp$, $\delta=\surd$ }

Modélisant « *Je ne fais pas tout puisque je ne compte pas !* ».

3.3.7 Limites de notre modèle

Notre modèle de requêtes permet de représenter toutes les questions de notre corpus sur *Coco* et *Jojo*. Cependant, nous avons pu envisager par la suite plusieurs questions qui ne sont pas représentables correctement en utilisant notre modèle de requêtes.

3.3.7.1 Opérateurs booléens

Lorsqu'un utilisateur demande : « *Est-ce qu'il est 3h ou 4h ?* », ce n'est pas la disjonction de *heure = 3* et *heure = 4* qu'il souhaite, mais bien savoir l'heure qu'il est, sachant qu'il suppose qu'il est ou bien 3 heures, ou bien 4 heures (le fait de faire part de cette supposition dans la question est un acte de langage, au sens de Searle [130]).

Ainsi, la conjonction de coordination « ou » de la langue naturelle est complètement différente de l'opérateur logique *ou*. C'est pour cette raison que les requêtes *Yes-No* de KQML ou les requêtes utilisées dans les modèles logiques ne sont pas des questions « naturelles », c'est-à-dire telles que pourraient en poser un utilisateur humain : l'opérateur *ou* n'y prend pas la même signification qu'en langage naturel.

Ce problème a été initialement mis en évidence par Shapiro [132], qui souligne la difficulté de « formaliser l'anglais » à l'aide de la logique du premier ordre (LPPO). Il propose d'utiliser une solution, implémentée dans le modèle SNePS [133], pour modéliser ce type de notions de bon sens issues du langage naturel.

Dans notre étude, nous souhaitons définir un modèle de requêtes permettant de représenter les questions naturelles sur le fonctionnement. Dans ce cadre, l'objet ω de la requête contient l'ensemble des « littéraux » de la question, qu'il s'agisse de questions sur l'état de la vue (*is*) ou de questions sur ce que la vue fait (*do*). Éventuellement, s'il y a des littéraux positifs et négatifs (par exemple : « *Tu comptes, là ? Tu ne dors pas ?* »), nous utiliserons une requête multiple :

$$\{ \alpha=\text{Assert}, \tau=\text{do}, \sigma=\{\text{view}\}, \omega=\{\text{count}\}, \nu=\top, \delta=\sqrt{} \\ \alpha=\text{Assert}, \tau=\text{do}, \sigma=\{\text{view}\}, \omega=\{\text{sleep}\}, \nu=\perp, \delta=\sqrt{} \}$$

Requêtes d'état

Pour les questions portant sur l'état du composant ($\tau = is$), nous pouvons représenter les questions booléennes (c'est-à-dire les requêtes comme « *Est-ce que $a \vee b$ est vrai* », possibles dans les modèles de requêtes fondés sur l'approche logique) en utilisant les opérateurs arithmétiques VDLp :

– Est-ce que que « valeur=2 ou vitesse>3 » :

$$\alpha=\text{Ask}, \tau=\text{is}, \sigma=\emptyset, \nu=\top, \delta=\sqrt{}, \\ \omega=\{ \text{or}[\text{equals}[\text{get}[\text{value}],2]], \text{"greater than"}[\text{get}[\text{speed}],3]] \},$$

Nous pouvons remarquer aussi que toute requête de la forme $\langle Ask, is, \sigma, \omega, \nu, \delta \rangle$ avec $\sigma \neq \emptyset$ peut aussi s'écrire :

$$\langle Ask, is, \emptyset, \{\text{equals}[\text{get}[\sigma], \omega]\}, \nu, \delta \rangle$$

puisque'il s'agit d'un cas particulier de requêtes dans le modèle logique. De cette façon, il est possible de modéliser des requêtes de type *is* faisant intervenir à la fois des valeurs de variables et des relations entre variables.

De manière générale, pour les requêtes statiques (*i.e.* ne faisant pas intervenir d'action), un modèle de requêtes fondé sur l'approche « logique » serait bien mieux adaptée et bien plus efficace que l'approche vecteur proposée ici. Cependant :

1. Le sujet principal de notre étude est le fonctionnement des composants, et non la structure statique ;
2. L'utilisation des opérateurs VDLp permet de simuler l'essentiel des requêtes logiques dont nous avons besoin pour notre étude.

Requêtes d'actions

En utilisant un modèle logique, nous pourrions envisager de définir des requêtes booléennes sur les actions, par exemple pour modéliser la question : « *Est-ce que tu es en train de compter et de dormir ou est ce que tu es en train de manger ?* ». Toutefois, trois problèmes se posent :

1. Ces questions ne sont pas à proprement parler « naturelles » (quel utilisateur poserait une telle question à *Coco* ?).
2. Nous pouvons très bien envisager de concevoir des requêtes faisant intervenir à la fois l'activité et le comportement des vues (par exemple : « *Est-ce que tu es en train de dormir et tu peux manger ?* »).
3. La construction d'une réponse pertinente à ces questions, lorsque la formule n'est pas valide dans la situation courante (par exemple, la vue peut *manger* mais n'est pas en train de *dormir*) consiste à trouver une formule booléenne d'actions décrivant la situation, c'est-à-dire à rechercher la plus petite formule valide plus générale que la question, ce qui est un problème NP-complet.

C'est la raison pour laquelle nous n'avons étudié la possibilité de modéliser ces questions dans notre langage de requêtes. C'est une des extensions possibles de notre modèle.

Travaux futurs

A terme, nous envisageons d'utiliser dans l'objet ω d'une requête un modèle logique, par exemple inspiré de SNePS [133] et utilisant une syntaxe VDL, pour modéliser et raisonner sur des notions naturelles que nous avons simplement contournées ici.

3.3.7.2 Acte *WhatIf*

Les questions de la forme « *Que se passera-t-il si je fais cela ?* » ne sont pas représentables. Elles correspondent à une classe de questions associées à un traitement

particulier. Notre modèle de requêtes peut donc être étendu facilement pour les représenter, en ajoutant l'acte $\alpha = WhatIf$. Le traitement de la requête consisterait alors à rechercher les conséquences d'une action, en utilisant les mécanismes proposés au chapitre suivant. Comme pour le diagnostic, la question qui se pose est de savoir jusqu'où aller dans les ramifications des actions.

Nous prendrons en compte cette extension dans le chapitre suivant.

3.3.7.3 Savoir et pouvoir

Dans notre modèle, les questions de la forme « *Est-ce que tu sais manger ?* » sont représentées à l'aide de requêtes d'acte *Ask* et de type *can*, qui sont ambiguës, comme nous l'avons vu section 3.3.1.3 :

$$\alpha=Ask, \tau=can, \sigma=\{view\}, \omega=\{eat\}, \nu=\top, \delta=\surd$$

Lors du traitement de la requête, le MRF analyse les préconditions des actions pour désambiguïser la requête *can* et déterminer s'il s'agit d'une possibilité morale ou physique. Par conséquent, l'information initiale de possibilité *physique* est perdue lors de la construction de la requête, puis retrouvée par le MRF lors de la construction de la réponse.

Toutefois, cette approche est bien adaptée à notre problématique :

- Lorsque la question de l'utilisateur est ambiguë (utilisation du verbe « pouvoir »), le MRF détermine le cadre de la question ;
- Lorsque la question de l'utilisateur est une question de possibilité physique, le résultat donné par le MRF est correct.

De plus, le MLN peut mettre à profit la connaissance qu'il a de la nature de la question pour produire une réponse adaptée : « *Oui, je sais manger* » ou « *Non, je ne sais pas manger* » plutôt que « *Oui, je peux manger* » ou « *Non, je ne peux pas manger parce que je ne sais pas ce que manger signifie* ».

3.3.7.4 Comparaison entre deux instants

Dans notre modèle, il n'est pas possible de modéliser des questions de la forme : « *Est-ce que tu comptes moins vite qu'il y a 2 minutes ?* ». Idéalement, pour obtenir une réponse appropriée, le MLN devrait produire la requête suivante :

$$\alpha=Ask, \tau=do, \sigma=\{view\}, \omega=\{"slow\ down"\}, \nu=\top, \delta=past$$

Et vérifier que la date de la réponse (si elle est positive) est supérieure à $-\frac{2}{T}$ où T est la période d'un cycle d'exécution. C'est bien évidemment impossible sans des connaissances sur la sémantique opérationnelle de VDLp. En effet, la question (« *compter moins vite* ») elle-même utilise ce type de connaissances.

Pour modéliser cette question sans munir le MLN de connaissances sur la sémantique des actions du composant, il faudrait utiliser un langage beaucoup plus riche, par exemple de type SQL, et demander au MRF de vérifier que la proposition construite est sémantiquement correcte pour le composant. C'est quelque chose que

nous ne souhaitons pas faire dans le cadre de cette thèse, mais que nous envisageons dans des travaux futurs.

3.3.7.5 Manque d’expressivité

Réponses inadaptées

Le manque d’expressivité de notre modèle conduit le MRF à donner des réponses inadaptées dans certaines situations, que le MLN ne peut pas toujours « rattraper ». Par exemple, si l’utilisateur demande « *Est-ce que tu comptes moins vite ?* », le magicien d’Oz répondra : « *Moins vite que quoi ?* » alors que, dans notre système, le MLN construira l’une des deux requêtes suivantes, aboutissant chacune à une réponse inadaptée :

$$\begin{aligned} \alpha &= \text{Ask}, \tau = \text{do}, \sigma = \{\text{view}\}, \omega = \{\text{"slow down"}\}, \nu = \top, \delta = \surd \\ \alpha &= \text{Ask}, \tau = \text{order}, \sigma = \{\text{slower}\}, \omega = \emptyset, \nu = \top, \delta = \surd \end{aligned}$$

Nous verrons au chapitre suivant que le MRF répond alors par une requête modélisant « *Je ne sais pas.* » :

$$\alpha = \text{Unknown}, \tau = \tau_{\text{question}}, \sigma = \sigma_{\text{question}}, \omega = \omega_{\text{question}}, \nu = \top, \delta = \surd$$

Comme pour la comparaison entre deux instants, le MLN ne peut pas construire une requête utilisables par le MLN ni détecter qu’il manque un paramètre, à moins de disposer de connaissances sur la sémantique opérationnelle des vues ou d’utiliser un langage de requêtes plus riche, mais source d’une plus grande complexité syntactico-sémantique et pouvant aboutir à des requêtes inconsistantes pour un composant donné.

Questions sur le fonctionnement non modélisables

Notre modèle permet d’utiliser l’objet ω de la requête pour représenter plusieurs actions, mais toutes les actions utilisées dans une requêtes sont nécessairement des sous-termes du terme référencé par σ . Par conséquent, notre modèle de requête ne permet pas de représenter la question : « *Est-ce que le "chauffer" du four est l’inverse du "refroidir" du frigo ?* ».

Dans le cadre de l’interaction entre un utilisateur et un composant, cette limite peut être contournée simplement en utilisant des noms d’actions différents dans la vue et $\sigma = \{\text{view}\}$ dans les requêtes sur le fonctionnement. Mais dans le cadre d’une interaction entre composants, par exemple dans un SMA, nous ne pouvons pas garantir que deux composants différents n’aient pas la même action. Dans ce cadre, un composant C peut vouloir comparer deux actions de deux agents différents A et B , modélisés dans les connaissances du composant C par deux sous-termes de la vue. Notre modèle de requêtes ne permet alors pas de modéliser la question pour l’agent C .

Toutefois, ce problème est facilement contournable en englobant dans un concept de référence spécifique (par exemple "*ref action*") les actions donnée dans dans l’objet ω . Ainsi, nous pourrions écrire :

```

 $\omega = \{ \text{inverse}["\text{ref action}["\text{oven,heat}],$ 
            $\text{"ref action}["\text{fridge,cool down}"] \}$ 

```

Questions structurelles non modélisables

Notre modèle ne permet pas de représenter les questions de recherche d'une variable vérifiant une relation, alors que ces requêtes sont facilement traitées dans les approches logiques ou en SQL. Par exemple, une vue ne pourra pas répondre à la question : « *Qu'est-ce qui est négatif ?* » (qui est une question assez naturelle) parce qu'elle n'est pas représentable dans notre modèle de requêtes, alors qu'elle est parfaitement traitable par une simple recherche des variables dans la vue, en utilisant un algorithme avec backtracking comme celui mis en œuvre pour la résolution de requêtes en PROLOG.

Dans notre étude, nous nous sommes en effet limité à l'étude du fonctionnement. Les relations statiques et les requêtes faisant intervenir directement les variables ($\tau = is$) ont été abordées par effet de bord, pour faciliter le traitement des requêtes sur le fonctionnement. Comme nous l'avons déjà souligné, les modèles classiques (approche logique, SQL, etc) sont bien mieux adaptées pour la modélisation et le traitement des questions complexes sur la structure. C'est la raison pour laquelle nous envisageons de proposer, dans des travaux futurs, un modèle de requête plus riche, permettant à la fois de modéliser les requêtes « classiques » sur la structure et les requêtes sur le fonctionnement.

3.4 Bilan

Le modèle de requêtes que nous avons présenté ici fait suite à nos travaux sur la construction automatique de réponses aux questions sur le fonctionnement [118, 117, 116]. Il est suffisamment expressif pour modéliser toutes les requêtes de notre corpus de questions sur le fonctionnement pour les composants simples *Coco* et *Jojo*. Ces deux composants utilisant la quasi-totalité des actions exprimables en VDLp, nous pouvons dire que le modèle de requête permet de modéliser une très large classe de questions qu'un utilisateur humain peut poser concernant l'état, l'activité, le comportement et les capacités d'interaction d'un composant actif décrit en VDL.

L'utilisation d'un modèle de type « vecteur » réduit l'expressivité de notre langage mais permet au MLN de construire plus facilement les requêtes (il s'agit de remplir des cases avec des valeurs, et non de passer d'un langage très riche (la langue naturelle) à un autre langage, un peu moins riche (le langage de requête). De plus, la catégorisation des questions correspondant aux différents critères de notre modèle de requêtes permet de rendre plus systématique, et donc plus efficace, le traitement des requêtes au sein du MRF. C'est la raison pour laquelle notre modèle de requêtes est bien adapté à notre problématique, étudiée au sein du projet *InterViews*.

Malgré cela, plusieurs classes de questions ne sont pas représentables avec un modèle aussi simple que celui que nous avons proposé. Nous envisageons plusieurs

pistes pour améliorer ce modèle :

1. Nous voudrions procéder à d'autres expériences de type « magicien d'Oz », sur des composants plus complexes comme nous en proposons dans le chapitre 7. Nous pensons ainsi pouvoir mettre en évidence l'ensemble des questions sur le fonctionnement qui ne sont pas modélisables dans notre modèle. A partir de l'analyse des manques, nous voulons proposer des extensions pour le modèle de requêtes et pour les algorithmes de traitement présentés dans le MRF qui permettraient de prendre en compte une plus large classe de questions.
2. Nous voudrions étudier la problématique de la modélisation des commandes directes sous la forme de requêtes formelles, ainsi que leur traitement par le MRF. Nous pensons qu'un travail conjoint entre le MRF et le MLN est alors nécessaire et nous envisageons d'utiliser un autre modèle de requêtes que celui que nous avons proposé ici, bien adapté pour les questions (de contrôle ou de commande) mais pas pour les commandes.
3. Enfin, pour combler le manque d'expressivité de notre modèle de type « vecteur », nous envisageons de proposer un modèle orthogonal et complet, de type « SQL », pour modéliser une plus large classe de questions. L'utilisation de ce modèle n'est possible que si le MLN est capable de produire des requêtes dans ce langage et si le MRF peut traiter les requêtes construites, même lorsqu'elles sont inconsistantes pour le composant considéré. C'est l'une des raisons pour lesquelles nous nous sommes limité ici à un modèle plus simple, mais plus facilement utilisable.

Dans le chapitre suivant, nous présentons les algorithmes de traitement des requêtes formelles sur le fonctionnement modélisant des questions de contrôle, en nous appuyant sur le modèle que nous avons présenté ici. Nous montrons que l'interaction entre un humain et une machine fait ressortir la notion de « bon sens » et qu'il est nécessaire de pouvoir modéliser des connaissances de bon sens pour traiter les requêtes formelles. Nous présentons un cadre opératoire pour répondre à ce besoin.

Chapitre 4

Traitement des requêtes

Le langage de requêtes que nous avons présenté au chapitre précédent permet de modéliser une large classe de questions qu'un utilisateur humain peut poser concernant l'état, l'activité, le comportement et les facultés d'interaction d'une vue décrite en VDLp. Dans ce chapitre, nous présentons les mécanismes de traitement mis en œuvre dans le MRF pour construire de manière automatique des réponses aux requêtes formelles modélisant des questions sur les actions et le fonctionnement d'un composant actif, produites par le MLN.

Dans la première section, nous étudions la problématique du traitement de ces requêtes formelles. Nous définissons les objectifs du MRF puis nous montrons que, dans le cadre d'une interaction entre un utilisateur ordinaire et une machine, il est nécessaire de prendre en compte des notions de bon sens dans les requêtes. Nous présentons dans la deuxième section un cadre formel qui permet d'une part de modéliser des *connaissances de bon sens* (au sens où l'entend la CHM, *i.e.* non spécifiques à une vue) et d'autre part d'interpréter ces connaissances dans le contexte d'une vue donnée pour pouvoir interagir avec l'*utilisateur ordinaire*. Enfin, dans la troisième section, nous proposons des mécanismes construction de réponses aux requêtes formelles de contrôle, en nous appuyant sur les connaissances de bon sens.

Nous n'abordons pas ici le cas des requêtes faisant intervenir des notions temporelles. Elles sont traitées au chapitre 5.

4.1 Étude de la problématique

4.1.1 Objectifs

Dans cette thèse, nous essayons de définir des composants actifs capables de se représenter leur fonctionnement et de raisonner dessus pour répondre en cours d'exécution à une large classe de questions qu'un utilisateur humain peut poser concernant leurs actions et leur exécution. Le langage VDLp que nous avons proposé au chapitre 2 permet aux composants actifs d'accéder en cours d'exécution à la description de leur propre fonctionnement. Le langage de requêtes formelles que

nous avons proposé au chapitre précédent permet de modéliser une large classe de questions qu'un utilisateur *ordinaire* peut poser concernant les actions et le fonctionnement d'un composant actif.

L'objectif du module de raisonnement sur le fonctionnement (MRF) et de construire des réponses formelles à ces requêtes, en utilisant la description des composants en VDLp.

La modélisation des questions proposées au chapitre précédent et la catégorisation suivant le couple (α, τ) permet de faire ressortir les différents traitements automatiques que doit effectuer le MRF pour construire une réponse. Ces traitements sont regroupés dans le tableau récapitulatif 4.1 page suivante.

Notion de réponse *naturelle et pertinente*

Cette modélisation des questions met aussi en évidence la nécessité pour le MRF de donner de réponses bien adaptées à la classe de question modélisée par le requête. Par exemple, si l'utilisateur demande à *Coco* : « *Tu comptes ou tu es arrêté ?* » le MRF doit répondre « *Je compte.* » ou « *Je suis arrêté.* » selon la situation, mais pas « *Je ne compte pas et je suis arrêté.* » qui n'est pas une réponse « naturelle » et « pertinente » :

Nous appelons réponse *naturelle* une réponse telle que pourrait en donner un humain (par exemple le magicien).

Nous appelons réponse formelle *pertinente* une réponse formelle dans laquelle sont donnés tous les éléments nécessaires à la construction d'une réponse naturelle, et uniquement ceux-ci.

Pour construire des réponses pertinentes, le MRF doit non seulement traiter tous les différentes combinaisons possibles de requêtes mais aussi, pour chaque classe de requête, considérer la sémantique explicative des concepts VDLp intervenant dans la question et dans la réponse, et en particulier la nature des actions, suivant la classification proposée section 2.3.4.2.

Dans ce chapitre, nous présentons le traitement des différentes classes de requêtes, qui s'appuient sur les travaux effectués dans le domaine du raisonnement sur les actions en logique (*cf.* chapitre 1) et, pour chaque classe de requêtes, nous définissons des heuristiques de choix de réponses formelles pertinentes.

4.1.2 Notion de requêtes de bon sens

4.1.2.1 Interaction avec un utilisateur ordinaire

Dans notre étude, nous nous intéressons à l'interaction avec des utilisateurs *ordinaires*. Comme nous l'avons montré au chapitre 1, la prise en compte des utilisateurs ordinaires nécessite de pouvoir manipuler des connaissances « de bon sens » (*common-sense knowledge* en anglais), c'est-à-dire :

	is	do	can	order
What	Recherche de l'état d'une variable	Recherche des processus (in)actifs de la vue	Recherche des actions (im)possibles de la vue	Recherche des événements externes (im)possibles
Ask	Vérification de l'état d'une variable	Recherche d'une action (in)active et analyse de ses préconditions		Recherche d'un (non) envoi d'événement externe
How	Planification	Extraction du corps d'une action	Construction d'un état but et planification	Erreur
Why	Construction d'une explication à base de diagnostic			Extraction d'un comportement
WhatIf	Simulation d'une exécution sur une étape « significative »			
Order	Commande directe		Erreur	Commande indirecte
When	Recherche de la date d'un état	Recherche de la date d'une action (in)active	Recherche de la date d'une précondition (in)valide	Recherche d'un envoi d'événement externe
Assert	Affirmation d'un état	Affirmation de l'(in)activité d'une action		Affirmation d'un (non) envoi d'événement externe
Unknown	Variable inconnue	Action inconnue	Impossible de déterminer l'objet d'une action	Événement externe inconnu
Error	Erreur dans la formulation de la requête ou traitement impossible dans le MRF. Le sujet donne la cause de l'erreur.			

TAB. 4.1 – Mécanismes de traitements des requêtes par le MRF

Dans ce tableau, nous disons par abus de langage qu'une action est « active » pour désigner indifféremment un processus actif ou une réaction possible.

- non spécifiques à un composant mais, au contraire, communes à tous les utilisateurs,
- qui ne sont jamais exprimées explicitement dans la description du composant mais qui vont être utilisées par l'utilisateur pour désigner les éléments du système avec lequel il interagit.

Par exemple, notre corpus de questions pour *Coco* (issus d'une interaction *entre humains*) montre l'utilisation à plusieurs reprises de connaissances de bon sens :

- **Réponse 8** : « *incrémenter la valeur* » : l'action « incrémenter » n'est pas donnée explicitement en VDLp dans la vue.
- **Question 14** : « *compter à l'envers* ». Rien dans la vue ne décrit ce que « à l'envers » peut signifier pour l'action compter ;
- **Réponse 14** : « *vitesse négative* ». Ces relations unaires ne sont pas exprimées directement en VDL dans la vue ;
- **Questions 20** : « *changer la valeur* » : la notion de changement, inhérente au fonctionnement même des vues (l'état change à chaque cycle) n'est pas matérialisée explicitement dans un composant VDL ;
- **Réponse 42** : « *ne rien faire* » : l'inactivité (c'est-à-dire l'absence de processus actif) n'est pas matérialisée explicitement dans un composant VDL ;

4.1.2.2 Traitement de la référence

L'interprétation de ce type de connaissances de bon sens fait appel à des connaissances spécifiques sur la sémantique opérationnelle et explicationnelle des termes VDL de la vue, auxquelles le module de traitement de la référence structurale ne peut pas avoir accès parce que l'utilisation de ces connaissances fait partie des mécanismes de résolution de la requête.

Par exemple, dans un composant comme *Jojo* présenté section 2.4.3.2 page 86, si l'utilisateur demande : « *Pour prendre le cube vert, est-ce que je dois déplacer le rouge ?* », le MLN va relier :

- « le cube vert » avec un terme t_1 de la vue, de concept *figure* et ayant au moins pour attributs *shape[square]* et *color[green]* ;
- « le rouge » avec un terme t_2 , de concept *figure* et d'attributs *shape[square]* et *color[red]* ;
- « prendre » avec l'action a_1 du composant attachée au sens *WordNet* (*take*, 4).

En revanche, aucune action ne peut être reliée à « déplacer » qui fait appel à des connaissances de bon sens. C'est la construction de la réponse formelle, obtenue par des techniques de planification [40, 27], qui permet de déterminer si le cube rouge sera déplacé, c'est-à-dire si sa *position aura changé* au cours de la réalisation du plan.

Dans ce contexte, le MLN ne peut pas identifier les éléments dont parle l'utilisateur en utilisant simplement le traitement de la référence structurale [98]. Il doit donc produire des requêtes formelles dans lesquelles certaines valeurs ne sont pas des éléments de la vue, mais des notions de bon sens que le MRF doit interpréter.

4.1.2.3 Définition

Il existe donc des *requêtes formelles de bon sens* (ou CSR pour *Common-Sense Request*), dans lesquelles les problèmes spécifiques au traitement de la langue naturelle ont été résolus mais faisant intervenir des notions « naturelles » issues directement de la formulation de la question par l'utilisateur et de la différence entre le perçu et l'implémenté.

Pour produire ces requêtes, le MLN doit avoir accès à ces notions de bon sens et à leur sémantique explicationnelle, du point de vue du TALN. En revanche, il n'a pas accès à l'interprétation qu'en fait le MRF.

4.1.2.4 Manipulation des connaissances de bon sens dans le MRF

Considérons les notions de bon sens mises en évidences dans l'extrait de notre corpus donné en annexe page A-2. Nous pouvons remarquer qu'elles apparaissent à la fois dans les questions de l'utilisateur et dans les réponses que devrait fournir le MRF. Du point de vue de l'utilisateur, elles n'apparaissent pas comme des connaissances particulières, mais comme faisant partie des connaissances du composant, sur lesquelles il peut raisonner : il n'existe pas de différence entre une CSR et une requête formelle « ordinaire », ne faisant intervenir que des notions spécifiques au composant.

Nous devons donc, pour chacune de ces connaissances de bon sens qui apparaît dans le corpus, proposer des mécanismes qui permettent de les manipuler dans les algorithmes de traitement de requêtes du MRF, comme s'il s'agissait de termes VDL, afin de pouvoir répondre aux CSR.

Cependant, ces notions de bon sens ne sont munies d'aucune sémantique opérationnelle formelle. Pour pouvoir les manipuler dans les algorithmes de traitement de requêtes, le MRF doit donc associer chaque notion de bon sens à une interprétation en termes VDLp, dont la sémantique opérationnelle a été bien définie. Dans ce chapitre, nous présentons un cadre formel qui permet de modéliser les connaissances de bon sens et cette correspondance avec des termes dont la sémantique est bien définie.

Nous pouvons aussi souligner que les notions de bon sens apparaissent uniquement dans l'objet ω des requêtes. En effet, c'est le seul critère de notre modèle qui peut prendre une valeur libre. Les mécanismes de traitement des CSR porteront donc sur l'objet ω des requêtes : le MRF doit pouvoir traiter des requêtes contenant des notions de bon sens dans ω et, réciproquement, produire des réponses dont l'objet contient des notions de bon sens.

Dans les exemples illustrant l'utilisation des notions de bon sens, nous ne donnerons généralement que l'objet de la CSR.

Remarque

L'utilisation des connaissances de bon sens n'est pas exclusivement du ressort du MRF. En effet, le traitement de la référence spatiale s'appuie aussi sur des connais-

sances de bon sens, comme nous l'avons vu au chapitre précédent (par ex : « *l'objet du haut* »).

Certaines connaissances de bon sens peuvent même relever à la fois du MRF et de l'interface. Par exemple, pour répondre à la question « *C'est quoi le truc qui monte ?* », le module de traitement de la référence spatiale doit utiliser le repère cartésien pour savoir si l'objet monte ou descend dans le repère, puis le MRF doit interpréter en termes d'actions élémentaires l'action de la requête – qui peut donc être « descendre » si l'axe des ordonnées est inversé par rapport à la perception qu'en a l'utilisateur.

4.1.3 Plan du chapitre

La suite de ce chapitre est séparée en deux sections. Dans la première section, nous présentons le cadre formel utilisé pour traiter les requêtes de bon sens, c'est-à-dire pour représenter et manipuler des connaissances de bon sens sur les actions et le fonctionnement dans le MRF.

Nous mettons tout d'abord en évidence l'existence de trois problèmes distincts dans le traitement des CSR sur les actions, correspondant à trois classes de notions de bon sens pouvant intervenir dans ces requêtes. Nous présentons le modèle de représentation des connaissances de bon sens utilisé dans le MRF, appelé *Ground*. Nous proposons ensuite, pour chacune des trois problèmes soulevés, une structure de donnée qui permet de faire le lien entre ces connaissances de bon sens utilisées par des utilisateurs ordinaires et la représentation interne du système, respectant les contraintes et décrit dans le cadre imposé. Nous donnons à chaque fois des algorithmes qui peuvent être utilisés dans le traitement des requêtes pour prendre en comptes ces notions et construire une réponse formelle.

Dans la seconde section, nous présentons les algorithmes de traitement utilisés au sein du MRF pour répondre aux requêtes de contrôle envoyées par le MLN. Nous définissons tout d'abord les algorithmes de recherche d'action et d'extraction de préconditions qui sont à la base de la construction de réponse. Nous donnons ensuite, pour chaque acte α et pour chaque type τ , le mécanisme de traitement effectué par le MRF. Nous discutons à chaque fois le choix d'une réponse pertinente.

Le plan de cette seconde section est très énumératif, et donc moins agréable à lire que s'il était construit autour d'une progression dans les difficultés rencontrées. Il permet en revanche de visualiser de manière beaucoup plus claire la catégorisation des traitements, mise en avant dans le tableau 4.1 page 129. Le lecteur pourra ainsi plus facilement retrouver quel section décrit quel algorithme de traitement en fonction de l'acte et du type de la requête. L'ordre des actes de langage a été choisi de manière à rendre compte de la progression dans les traitements.

Nous n'aborderons pas dans ce chapitre les requêtes faisant intervenir des notions temporelles, c'est-à-dire :

- Les requêtes ayant une date $\delta \neq \surd$;
- Les requêtes ayant pour acte $\alpha = \textit{When}$, correspondant à la recherche d'une date.

Ces deux cas sont étudiés dans le chapitre suivant, dans lequel nous présentons un mécanisme de mémorisation et d'analyse de l'interaction des vues. Nous mettrons aussi en évidence la possibilité d'utiliser une analyse *dynamique* de l'interaction pour répondre aux requêtes d'acte *Why*, modélisant des demandes d'explications.

4.2 Manipulation de connaissances de bon sens

Dans cette section, nous proposons un cadre formel qui permet de représenter et de manipuler des connaissances de bon sens dans les requêtes.

4.2.1 Catégorisation des CSR

4.2.1.1 Analyse du corpus

Considérons les CSR de notre corpus pour *Coco*. Nous voyons apparaître différents types de connaissances de bon sens que le MRF doit pouvoir manipuler :

- **Question 14** : « *compter à l'envers* » : rien dans la vue ne décrit ce que « à l'envers » peut signifier pour l'action compter. Pourtant, un utilisateur humain manipulera cette notion de bon sens sans difficulté. Le MRF doit donc savoir ce que « faire une action à l'envers » signifie, en fonction de la description de cette action en VDL ;
- **Réponse 13 et 14** : « *vitesse négative/nulle* » : les relations « x est négatif » ou « la valeur de x est nulle » ne sont pas exprimables directement en VDL. Elles sont obtenues à partir de l'interprétation des concepts arithmétiques. Pourtant, l'humain manipule ces notions sans difficulté. Le MRF doit donc posséder ce type de connaissances de bon sens ;
- **Réponse 42** : « *ne rien faire* » : le composant « ne fait rien » lorsqu'aucun processus n'est actif. Le MRF doit pouvoir matérialiser cette connaissance de bon sens sur son activité ;
- **Questions 20, 39 et 40** : « *changer la valeur/vitesse* » : la notion de changement est implicite dans la description du composant (elle est inhérente au fonctionnement même des vues, dont l'état change à chaque cycle). Pourtant, dans les requêtes, nous avons besoin de la considérer comme une action effectuée par le composant, alors qu'aucune action « changer » n'est définie dans la vue. Le MRF doit donc pouvoir matérialiser la notion de changement.
- **Réponse 8** : « *incrémenter la valeur.* »

L'utilisateur n'attend pas du système qu'il lui décrive l'action par : « *Je remplace ma valeur par la somme de ma valeur et de ma vitesse (à chaque cycle)* ». Un humain dirait plus naturellement : « *J'ajoute ma vitesse à ma valeur (à chaque cycle)* » ou, de manière plus formelle : « *J'incrmente ma valeur par ma vitesse (à chaque cycle)* ».

Cette action « *d'ajouter y à la variable x* » n'est pas décrite de manière explicite dans la vue. Pourtant, tous les utilisateurs humains peuvent l'utiliser. Le MRF doit donc pouvoir la manipuler.

4.2.1.2 Trois problèmes distincts

Nous pouvons alors mettre en évidence trois classes de notions de bon sens qui peuvent être utilisées dans les CSR :

1. Les relations entre des variables (ou *relations statiques*).

Par exemple, si nous considérons un composant qui permet de visualiser dans une représentation graphique (par exemple dans un histogramme) les valeurs de certains paramètres du système, l'utilisateur peut demander « *pourquoi x est positif* » ou « *est-ce que x divise y* ». En utilisant la syntaxe VDLp, pour le premier exemple, nous aurons dans la requête `positive[get[x]]`. Pour traiter cette requête, le MRF doit alors être capable de remplacer `positive[get[x]]` par `"greater than"[get[x],0]`.

Remarque

Les relations de bon sens entre les variables peuvent apparaître à la fois dans des requêtes portant uniquement sur la structure statique du composant (« *Est-ce que x divise y* ») et dans des requêtes sur le fonctionnement (« *Quelle est l'action qui a rendu x positif ?* »). C'est la raison pour laquelle, même si le traitement de ces relations n'est pas directement du ressort du MRF, mais plutôt d'un sous-module « relations statiques » qui pourrait être situé dans le MLN, nous le prenons en considération dans notre étude.

2. Les actions de bon sens.

Par exemple, l'utilisateur peut constater l'augmentation d'une valeur donnée de l'histogramme et se demander ce que cette valeur représente : « *Qu'est-ce que c'est le truc qui augmente ?* ». Nous obtiendrons alors une requête contenant l'expression `increase[path[x]]`, suivant une syntaxe VDLp, alors qu'aucune action `increase` n'a été définie dans le composant. Pour traiter cette requête, le MRF doit être capable d'associer l'action de bon sens `increase[path[x]]` à l'expression formelle :

```
put[path[x],plus[get[x],get[n]]
```

3. Les relations entre des actions (ou *relations procédurales*).

Par exemple, « *Pourquoi tu comptes à l'envers ?* » ou « *Est-ce que 'start' fait bien le contraire de 'stop' ?* » dans *Coco*. Si l'action `count` est associée au sens *WordNet* (`count,1`), le MLN peut construire une requête contenant l'expression `backwards[count]` dans son objet, comme nous l'avons vu au chapitre précédent. En utilisant la sémantique opérationnelle du langage et le code de l'action `count`, le MRF doit alors être capable de déterminer ce que « faire `count` à l'envers » signifie.

De même, le MRF doit pouvoir traiter une requête contenant dans son objet ω le terme `inverse[start,stop]`.

Nous devons donc munir le système de connaissances de bon sens spécifiques et des mécanismes de traitement des CSR pour chacun de ces trois types de notions.

4.2.2 Notion de *Ground*

Dans cette section, nous proposons :

1. Une structure de donnée permettant de matérialiser les connaissances de bon sens utilisables dans les requêtes par le MLN et leur interprétation en termes VDLp dont la sémantique est définie ;
2. Des algorithmes de traitement pour manipuler ces connaissances lors de la construction de réponses par le MRF.

4.2.2.1 Définition

Nous appelons :

- *Termes de ground*¹ les termes représentant des notions de bon sens qui sont utilisés dans les requêtes, comme *positive[get[x]]*, *increase[path[x]]*, etc.
- *Ground* la structure de donnée qui décrit l'ensemble termes de ground et leur correspondance avec les éléments du langage de programmation (en l'occurrence, VDLp), dont la sémantique est définie de manière rigoureuse et utilisable par les algorithmes du MRF.

Le *Ground* définit donc l'ensemble des connaissances de bon sens que le système (MLN et MRF) est capable de manipuler pour interagir avec un utilisateur ordinaire.

4.2.2.2 Utilisation du *Ground* par le MLN

Pour construire l'objet d'une requête, le MLN s'appuie sur le traitement de la référence structurelle : les variables et les actions référencées dans la question sont identifiées puis matérialisées dans les requêtes. Dans ce cadre, pour pouvoir produire des requêtes de bon sens (CSR), le MLN doit connaître non seulement l'ensemble des identifiants des variables et des actions du programme (reliés à des sens *WordNet*), mais aussi l'ensemble des termes de ground, qu'il peut utiliser dans ces CSR, associés chacun à un sens *WordNet*.

Par exemple, le MLN doit savoir qu'il est possible d'écrire *positive[x]* dans l'objet d'une requête pour modéliser l'expression « *x* est positif » de la question de l'utilisateur.

4.2.2.3 Utilisation du *Ground* dans les réponses

L'utilisateur ordinaire interagissant avec un composant actif ne peut pas utiliser des notions spécifiques au fonctionnement interne du composant. C'est la raison pour laquelle le MLN produit des CSR. Réciproquement, si le MRF envoie au MLN des réponses entièrement définies en terme du langage, le MRF produira une réponse qui fera appel à ces connaissances spécifiques que l'utilisateur n'a pas. Dans ce contexte, l'utilisateur ne pourra pas comprendre la réponse produite. Le MRF doit donc aussi pouvoir utiliser des notions de bon sens dans les réponses. Par exemple, si

¹Par analogie avec le *common ground* de Stalnaker [138].

l'utilisateur demande « qu'est-ce qui augmente », le MRF doit produire une réponse correspondant à « c'est x qui augmente » et non « c'est x qui reçoit $x + 7$ ».

C'est la raison pour laquelle nous devons proposer des algorithmes qui permettent d'utiliser les termes de ground à la fois dans les requêtes et dans les réponses construites par le MRF.

4.2.2.4 Trois *Ground*

Nous avons dégagé trois classes distinctes de notions de bon sens utilisables dans les CSR : les relations statiques, les actions de bon sens et les relations procédurales. Mais nous pouvons remarquer que :

- La définition d'une action de bon sens (2) peut s'appuyer sur des relations statiques de bon sens (1) ; par exemple, *augmenter x* , c'est lui ajouter quelque chose de *positif* ;
- La définition d'une relation procédurale de bon sens (3) peut s'appuyer sur des relations statiques de bon sens (1) et sur des actions de bon sens (2) ; par exemple, *augmenter* est l'*inverse* de *diminuer*.

Ainsi, chaque classe de notion de bon sens doit être définie de manière distincte, afin de pouvoir être utilisée par les « suivantes » : pour pouvoir définir et interpréter au niveau implémentatoire des actions de bon sens (comme *augmenter*), le *Ground* doit d'abord pouvoir traiter des relations statiques (comme *positif*) ; de même, pour pouvoir vérifier des relations de bon sens entre les actions, le *Ground* doit d'abord pouvoir manipuler les relations statiques et les actions de bon sens.

Par conséquent, nous devons définir trois type de *Ground* distincts pour représenter ces trois types de notions de bon sens. Nous notons :

- \mathcal{G}_{rs} le *Ground* des relations statiques.
Son rôle est de définir l'ensemble des relations statiques de bon sens utilisables dans les CSR et d'effectuer la correspondance entre les termes de ground représentant une relation statique de bon sens, comme `positive[x]`, et les relations statiques décrites en termes VDLp ayant une sémantique bien définie, comme `"greater than"[x,0]`.
- \mathcal{G}_p le *Ground* des actions de haut niveau sémantique (ou *Ground procédural*).
Son rôle est de définir l'ensemble des actions de bon sens utilisables dans les CSR et leur sémantique du point de vue du MRF. Il est utilisé par le MRF pour savoir, par exemple, que le terme de ground `increase[path[x]]` correspond à l'action VDLp :

`put[path[x], plus[get[x], n]`

si et seulement si la relation `positive[n]` est vraie (du point de vue de l'interprétation canonique ζ , et en utilisant l'interprétation fournie par \mathcal{G}_{rs}).

- \mathcal{G}_{rp} le *Ground* des relations procédurales.
Son rôle est de définir l'ensemble des relations procédurales de bon sens utilisables dans les CSR. Il est utilisé par le MRF pour savoir, par exemple, que la

relation `inverse[start,stop]` est vraie dans *Coco*, c'est-à-dire que l'action *start* du composant est l'inverse de l'action *stop*.

Le *Ground* des relations procédurales permet aussi au MRF d'interpréter des notions procédurales unaires comme « compter à l'envers », correspondant au terme de `ground backwards[count]` ou des relations d'arité 0 comme « ne rien faire » (`nothing`).

Comme nous pouvons le constater, chaque *Ground* est utilisé de manière différente par le MRF. Dans cette section, nous allons présenter en détail la structure de chacun de ces trois *Ground* et la manière dont ils sont utilisés dans les algorithmes de construction de réponse.

L'utilisation du *Ground* dans le mécanisme de construction de réponses aux questions sur le fonctionnement est illustré sur la figure 4.1. Le type de *Ground*

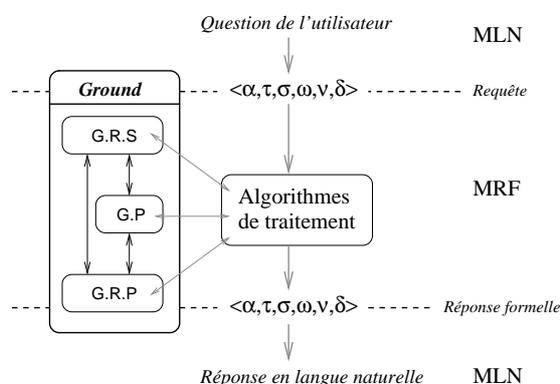


FIG. 4.1 – Le MLN utilise le *Ground* pour construire des CSR. Le MRF utilise les algorithmes de manipulation des notions de bon sens définis dans le *Ground*.

utilisé pour la construction et l'interprétation de la requête (\mathcal{G}_{rs} , \mathcal{G}_p ou \mathcal{G}_{rp}) dépend du type τ de la requête.

4.2.3 Utilisation de *patterns*

4.2.3.1 Présentation du besoin

Le *Ground* doit permettre de décrire des notions de bon sens génériques, c'est-à-dire pouvant s'adapter à tout composant actif décrit dans le langage considéré. Dans ce cadre, nous devons pouvoir décrire dans le *Ground* :

1. Des relations statiques de bon sens génériques, c'est-à-dire :
 - Qui peuvent s'adapter à n'importe quelle valeur ;
 - Qui sont calculables (au sens des machines de *Turing*).

Par exemple, nous voulons pouvoir représenter que « *x est positif* » correspond au fait que la valeur de *x* est plus grande que 0, et ce quel que soit *x* (`greater than[x, 0]` en VDL).

2. Des actions de bon sens génériques, c'est-à-dire dont les paramètres ne sont pas instanciés, pouvant s'appliquer sur n'importe quelle variable(s).

Par exemple, nous voulons pouvoir décrire que « *augmenter x* », c'est remplacer la valeur de x par cette valeur plus quelque chose de positif, et ce quel que soit la valeur de x et y (ce que nous noterons $put[path[x], plus[get[x], y]]$ avec $positive[y]$, en VDL).

3. Des relations procédurales génériques. Compte tenu de la nature des actions, les relations entre des actions ne peuvent pas toujours être calculables. Nous devons alors donner ces relations procédurales en extension, c'est-à-dire par énumération de toutes les actions entrant en relation. En revanche, nous voulons que ces actions puissent être génériques. Par exemple, nous voulons décrire que, quelque soit x , toute action :

$put[path[x], plus[1, get[x]]]$

est l'*inverse* de :

$put[path[x], plus[-1, get[x]]]$

Il est donc nécessaire de pouvoir utiliser des variables muettes dans les termes VDLp, c'est-à-dire des sous-termes « non instanciés ». Mais en VDLp, cette notion de variable muette n'existe pas. C'est la raison pour laquelle nous devons proposer un mécanisme de représentation et de manipulation de ces variables dans le *Ground*.

4.2.3.2 Notion de « *pattern de Ground* »

Par convention, dans le *Ground*, nous utiliserons la notation $\#i$, où $i \in \mathbb{N}$, pour représenter et manipuler des variables non instanciées. Par exemple, nous pourrions écrire :

$"greater\ than"[\#1, 0]$

pour définir l'ensemble des termes de la forme "*greater than*" $[t, 0]$ où t est un terme VDL quelconque.

Nous appelons *pattern* ces termes contenant des nœuds non instanciés.

Nous noterons $p \simeq \#i$ (*i.e.* p est de la forme $\#i$) lorsque p est un terme non instancié, *i.e.* un terme sans attributs et dont le concept est $\#i$, $i \in \mathbb{N}$.

L'utilisation de patterns dans le *Ground* permet de décrire les règles de manière générique, sans préciser certains paramètres.

4.2.3.3 Instanciation des patterns

Lors du traitement des requêtes par le MRF, termes $\#i$ sont instanciés par les termes utilisés dans la requête. Ce sont ensuite ces termes instanciés qui sont utilisés dans les algorithmes d'interprétation du MRF.

Considérons par exemple le pattern suivant :

`increase[path[#1],#2]`

qui modélise la notion de bon sens « *augmenter* ». Supposons que le MLN construise une requête dont l'objet contient le terme `increase[path[value],3]`. Le terme `#1` du pattern sera alors associé à la référence $\{value\}$ et `#2` à la valeur 3.

Cet exemple montre que les termes de la forme `#i` utilisés dans les patterns de ground peuvent être associés soit à des termes VDL, soit à des références, c'est-à-dire à un ensemble de termes, lorsqu'ils modélisent des variables VDL.

Rappel

En VDLp, les références sont englobées dans un terme dont le concept définit l'interprétation du résultat de la référence (*path* pour modifier le terme résultat, *get* pour extraire ses attributs, *etc*). Il est important de noter que nous ne pouvons pas nous affranchir de cette écriture, et ce pour deux raisons :

- D'un point de vue syntaxique, compte tenu du fait qu'une référence peut contenir plusieurs termes VDL, c'est le seul moyen de permettre à un terme de définir deux références, par exemple lorsqu'une action modifie deux variables :

`put[path[a],path[b],...] ≠ put[path[a,b],...]`

- D'un point de vue sémantique, c'est nécessaire pour définir la signification de chaque référence :

`put[path[a],...] ≠ put[a,...]`

Appariement avec des références

Nous pouvons utiliser cette contrainte d'écriture des références dans le *Ground* en englobant dans un terme de concept *path* ou *get* les termes `#i` qui doivent s'apparier avec des références, comme dans `increase[path[#1],#2]`.

Les termes `#i` dans les patterns seront donc instanciés par des *ensembles de termes* contenant un seul terme dans le cas général et n termes (correspondant à une référence) lorsqu'ils sont englobés dans un concept de référence (*path* ou *get*).

4.2.3.4 Algorithme d'appariement

Le *Ground* permet de modéliser non seulement les notions de bon sens utilisables dans les CSR, mais aussi la signification de ces notions pour le MRF, définie par des termes VDLp associés à une sémantique opérationnelle bien définie. Pour traiter les CSR, le MRF doit donc instancier les `#i` des patterns avec des termes VDL précis pour les remplacer dans la description VDLp. Pour ce faire, nous proposons un mécanisme à base de *pattern-matching* qui permet d'apparier un terme avec un pattern.

Nous notons $pm : \Upsilon \times \Upsilon \times (\mathbb{N} \times \mathcal{P}(\Upsilon)) \longrightarrow (\mathbb{N}, \mathcal{P}(\Upsilon)) \times \{\top, \perp\}$ la fonction récursive qui apparie un terme t avec un pattern p et donne les instanciations nécessaires sous

la forme $(i, \{t_j\}_i) \in (\mathbb{N}, \mathcal{P}(\Upsilon))$. Le troisième argument est la liste $l \in \mathbb{N} \times \Upsilon$ des substitutions de chaque $\#i$, passée en argument lors de la récursion (initialement $l = \emptyset$).

La fonction pm est fondée sur l'utilisation d'un algorithme classique de pattern-matching multi-termes. Par exemple, le pattern $p = foo[\#1, bar[\#2]]$ s'apparie avec le terme $t = foo[a, bar[b, c]]$ et nous obtenons la liste :

$$l = \{(1, \{a\}), (2, \{b, c\})\}$$

Algorithme

L'algorithme de pm est défini récursivement de la manière suivante :

1. Si $p \simeq \#i$, alors $l = l \cup \{(i, \{t\})\}$ (appariement avec un terme)
2. Si $concept(t) \neq concept(p)$ et $p \neq \#i$, ou si $|fils(t)| > |fils(p)|$, l'appariement échoue.
3. Si $concept(t) \in \{path, get\}$ et $|fils(p)| = 1$ et $fils(p)_1 \simeq \#i$, alors $l = l \cup \{(i, fils(t))\}$ (appariement avec une référence). Sinon, l'appariement échoue.²
4. Sinon, l'appariement réussit si et seulement si :

$$\forall st_i \in fils(t), \exists ! sp_i \in fils(p) \text{ tq } pm(st_i, sp_i) = (l', \top)$$

dans ce cas, l est remplacée par l' .

5. Si l'appariement réussit, l'algorithme retourne (l, \top) , et (\emptyset, \perp) sinon.³

Cet algorithme est calculable en $\mathcal{O}(|p|)$ où $|p|$ est le nombre de nœuds dans le pattern p .

Notation

Pour tout terme $t \in \Upsilon$ et pour tout ensemble de patterns P , nous noterons

$$\xi(t, P) = \{t' \in P \text{ tq } \exists l \text{ tq } pm(t, t', \emptyset) = (l, \top)\}$$

l'ensemble des patterns instanciables en t .

Cet ensemble est calculable en $\mathcal{O}(|t|^2 \cdot |P|)$ dans le pire des cas.

²Dans les cas 1 et 3, si $\#i$ est déjà apparié avec $\{t_j\}$ dans l et si $\{t_j\} \neq attributs(t)$ (au sens des ensembles classiques), alors l'appariement échoue.

Deux termes sont égaux si et seulement si ils ont le même concept et si leurs ensembles d'attributs sont égaux.

³L'appariement peut réussir et retourner une liste vide.

Exemple

Considérons les deux patterns :

$$\begin{aligned} p_1 &= a[x[\#1], y[\#2]] \\ p_2 &= a[x[\#1], z[\#2]] \end{aligned}$$

Alors $t_1 = a[x[b]]$ s'apparie avec p_1 et p_2 ; $t_2 = a[x[b], y]$ s'apparie avec p_1 seulement ; $t_3 = a[x, y, z]$ ne s'apparie ni avec p_1 , ni avec p_2 .

Remarques

Chaque $\#i$ est toujours instancié par un terme (dans son intégralité) ou une référence (*i.e.* un ensemble de termes). Par conséquent, il n'est pas possible de manière syntaxique d'apparier un $\#i$ en fonction de ses sous-termes (lorsque $\#i$ est apparié avec un terme) ou d'éléments spécifiques de la référence (lorsque c'est une référence).

Cependant, il est toujours possible d'imposer les sous-termes d'un $\#i$ ou certains éléments de la référence en utilisant la sémantique opérationnelle du langage VDLp dans la description du *Ground*. Par exemple, si l'on souhaite que $\#1$ s'instancie uniquement avec les termes ayant un sous-terme *truc*, nous écrivons :

"greater than" ["size ref" [#1, truc], 0]

4.2.4 Le *Ground* des Relations Statiques : \mathcal{G}_{rs}

4.2.4.1 Objectif

L'objectif de \mathcal{G}_{rs} est de permettre au MRF d'interpréter les CSR contenant des relations statiques de bon sens. Ces relations décrivent de manière naturelle des relations entre variables qui sont exprimables en utilisant les concepts arithmétiques VDLp. Pour pouvoir interpréter ces relations statiques, il suffit donc de pouvoir les *convertir* en des relations bien définies dans le langage de programmation. \mathcal{G}_{rs} décrit donc simplement des correspondances entre des relations statiques de bon sens et des relations bien définies dans le langage.

4.2.4.2 Structure

Les éléments de \mathcal{G}_{rs} sont des couples de patterns (s, t) . Nous dirons que s est le pattern de ground et t le pattern associé dans le langage de programmation. Par exemple, nous aurons dans \mathcal{G}_{rs} :

(*positive*[\#1], "greater than"[\#1, 0])

Dans \mathcal{G}_{rs} , un pattern de ground ne peut être associé qu'à un seul pattern dans le langage de programmation, et réciproquement :

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

Il y a donc 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.

Notation

Si $c \in A^*$, nous noterons abusivement $c \in \mathcal{G}_{rs}$ le fait que c est le concept d'un terme du ground des relations statiques, c'est-à-dire que $\exists (s, t) \in \mathcal{G}_{rs}$ tq $\text{concept}(s) = c$.

4.2.4.3 Utilisation dans le MLN

Du point de vue de TALN, toute relation statique de bon sens est caractérisée par :

- Le sens *WordNet* de la relation, qui définit l'interprétation de la relation ;
- Le nombre d'arguments de la relation et, éventuellement, leur rôle dans la relation (par exemple si la relation n'est pas symétrique).

Ces éléments sont utilisés lors du traitement de la référence et de la construction de la requête par le MLN.

L'ensemble de ces informations est représenté dans le concept VDL de la relation, de manière transparente pour le MRF. Nous ne les présenterons donc pas ici. Nous considérerons abusivement que le MLN connaît l'ensemble des patterns possibles en partie gauche de \mathcal{G}_{rs} , pour pouvoir construire des requêtes de bon sens faisant intervenir ces relations.

4.2.4.4 Utilisation dans le MRF

Les éléments de \mathcal{G}_{rs} sont utilisés dans le MRF :

1. Pour convertir une relation statique de bon sens en terme du langage et l'interpréter pour déterminer sa valeur de vérité et pouvoir ainsi construire une réponse pour l'utilisateur.
2. Dans la construction des réponses, pour décrire de manière naturelle (*i.e.* en utilisant le sens commun) les relations qui constituent une explication. Les relations définies dans le langage et utilisées dans les réponses sont converties en relations statiques de bon sens.

Compte tenu de la propriété de bijection entre les relations statiques et leur description dans le langage de programmation, l'utilisation de \mathcal{G}_{rs} par le MRF repose simplement sur un algorithme de conversion bi-directionnel.

Remarque

Les relations statiques sont utilisées uniquement dans les requêtes de type $\tau = is$. Cependant, elles interviennent aussi dans la définition des deux autres *Ground*.

4.2.4.5 Algorithme

Nous notons $\Xi_{rs} : \Upsilon \times \mathbf{B} \longrightarrow \Upsilon$ l'algorithme de conversion d'un terme de ground en terme VDL (et réciproquement) pour \mathcal{G}_{rs} . Le deuxième argument est un booléen permettant de préciser le sens de la conversion ($b = \top$ pour $ground \rightarrow langage$).

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

```

si b, G = {s tq  $\exists t' tq (s, t') \in \mathcal{G}_{rs}$ }
sinon, G = {t' tq  $\exists s tq (s, t') \in \mathcal{G}_{rs}$ }
res =  $\xi(t, G)$ 
si res  $\neq \emptyset$ 
    soit t' = min(res) et (l, b') = pm(t, t',  $\emptyset$ )
    retourner t' instancié suivant l
sinon
    soit c = concept(t) et {st1, ..., stn} = fils(t)
    retourner c[ $\Xi_{rs}(st_1, b), \dots, \Xi_{rs}(st_n, b)$ ]
/* conversion des attributs */

```

L'instance d'un terme t' suivant une liste de substitution l est le terme obtenu en remplaçant dans t' tous les $\#i$ par l'ensemble de termes $\{t_j\}_i$ associé à i dans l , ou en le supprimant s'il n'est pas instancié dans l . Par exemple, l'instance de $positive[\#1]$ suivant la liste $l = \{(1, \{a[b, c]\})\}$ est le terme $positive[a[b, c]]$. L'algorithme d'instanciation est un simple parcours en profondeur du terme avec substitution des $\#i$ par l'ensemble de termes associés.

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, c'est-à-dire qu'un terme donné ne s'apparie qu'avec un seul terme en partie gauche de \mathcal{G}_{rs} . Les cas d'ambiguïté sont traités en amont par le MLN.

Notons enfin que nous avons toujours $b' = \top$ dans l'algorithme puisque :

$$t' \in \xi(t, G) = \{t' \in G \text{ tq } \exists l \text{ tq } pm(t, t', \emptyset) = (l, \top)\}$$

Propriétés

L'algorithme proposé pour Ξ_{rs} est :

- *Robuste* : si $t \in \Upsilon$ est un terme VDL quelconque et $b \in \mathbf{B}$, $\Xi_{rs}(t, b)$ retourne toujours un résultat t' ;
- *Transparent* : $t' = t$ si t n'est pas une relation de bon sens.

Ces propriétés seront utilisées dans les algorithmes du MRF.

Enfin, $\Xi_{rs}(t, b)$ est calculable en $\mathcal{O}(|t|^2 \cdot |\mathcal{G}_{rs}|^{|t|})$ dans le pire des cas (*i.e.* lorsque le terme t n'est pas un terme de ground).

4.2.4.6 Exemple

Considérons un composant actif permettant de manipuler des données financières pour la gestion de comptes, implémenté par une vue v . Supposons que l'utilisateur demande : « *Est-ce que le bilan de ce mois est positif?* ». Le MLN utilisera la liste des patterns de ground, la liste des variables de *Coco* et la classification sémantique *WordNet* [38] pour construire la requête de bon sens suivante :

$$\langle Ask, is, \emptyset, \{positive[get[month, balance]]\}, \top, \sqrt{\} \rangle$$

Nous noterons $t = positive[get[month, balance]]$: l'objet de cette CSR est donc $\omega = \{t\}$.

Pour répondre à cette requête, le MRF doit déterminer la valeur de vérité de cette relation statique de bon sens. Pour ce faire, il faut :

1. Convertir t en un terme VDL $t_{vdl} = \Xi_{rs}(t, \top)$;
 Par construction, $(positive[\#1], greater\ than[\#1, 0]) \in \mathcal{G}_{rs}$. Alors l'ensemble $\xi(t, G)$ contient au moins $positive[\#1]$ (*c'est-à-dire* $\xi(t, G) \supseteq \{positive[\#1]\}$). Supposons pour simplifier que $t' = \min(\xi(t, G)) = positive[\#1]$. En utilisant $pm(t, t', \emptyset)$, nous construisons la liste de substitutions suivante :

$$l = \{(1, \{get[month, balance]\})\}$$

Nous aurons donc :

$$t_{vdl} = \Xi_{rs}(t, \top) = "greater\ than"[get[month, balance], 0]$$

2. Calculer la valeur de vérité de cette relation (*true* ou *false* en VDL) en utilisant l'interprétation canonique sur les termes dans la vue v considérée : $r = \zeta(t_{vdl}, \emptyset)_v$. Le calcul de cette valeur est effectué en rattachant *temporairement* le terme t_{vdl} à la vue v , pour permettre l'interprétation des références (comme $get[month, balance]$). Le terme t_{vdl} est ensuite retiré de la liste des attributs de v .⁴

Ainsi, $r = \zeta("greater\ than"[get[month, balance], 0], \emptyset)$ donc $r = true$ si et seulement si la variable référencée par $\{month, balance\}$ dans v a pour valeur $x > 0$.

⁴Ce mécanisme, dépendant de l'implémentation, n'est pas présenté ici. Nous notons $\zeta(t, \emptyset)_v$ l'interprétation d'un terme t dans une vue v .

La réponse fournie par le MRF est alors :

- $\langle \text{Assert}, is, \emptyset, \{\text{positive}[\text{get}[\text{month}, \text{balance}]]\}, \top, \sqrt{\} \rangle$ si $r = \text{true}$;
- $\langle \text{Assert}, is, \emptyset, \{\text{positive}[\text{get}[\text{month}, \text{balance}]]\}, \perp, \sqrt{\} \rangle$ si $r = \text{false}$.

La construction de cette requête utilise Ξ_{rs} dans l'autre sens ($b = \perp$).⁵ Cette requête formelle sera utilisée par le MLN pour produire la réponse en langue naturelle : « Oui » ou « Non » à la question de l'utilisateur.

La figure 4.2 illustre ce mécanisme de construction d'une réponse pour une relation statique dans une CSR. Nous pouvons remarquer que le traitement des requêtes

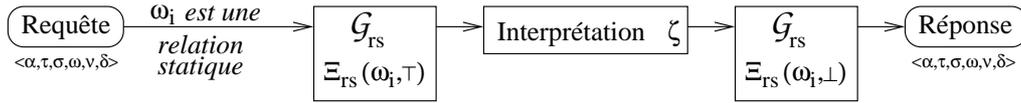


FIG. 4.2 – Traitement des relations statiques

est très simple dans ce cas précis dans lequel n'interviennent que les données et la structure du composant : il s'agit simplement de convertir les termes de ground dans ω .

Autres relations statiques

A titre d'exemple, nous aurons dans \mathcal{G}_{rs} les relations suivantes :

$$\begin{aligned}
 \text{positive}[\#1] &\Leftrightarrow \text{"greater than"}[\#1, 0] \\
 \text{negative}[\#1] &\Leftrightarrow \text{"greater than"}[0, \#1] \\
 \text{null}[\#1] &\Leftrightarrow \text{equals}[\#1, 0] \\
 \text{unknown}[\#1] &\Leftrightarrow \text{equals["size ref"}[\#1], 0] \\
 \text{divides}[D[\#1], d[\#2]] &\Leftrightarrow \text{equals}[\text{times}[\#1, \text{inverse}[\#2]], 0] \\
 &\text{etc.}
 \end{aligned}$$

Signalons que l'utilisation des concepts englobants D et d (pour *Dividende* et *diviseur*) n'est pas nécessaire si le MLN utilise l'ordre des attributs dans les termes VDL. Comme nous l'avons expliqué au chapitre 2, ils permettent de matérialiser la non-commutativité des attributs pour ce terme et la fonction de chaque attribut.

4.2.5 Le *Ground* procédural : \mathcal{G}_p

4.2.5.1 Objectif

L'objectif de \mathcal{G}_p est de permettre au MRF d'interpréter les CSR contenant des actions de bon sens. Les actions de bon sens modélisent de manière naturelle un

⁵En effet, comme nous le verrons dans la section 4.3, chaque relation est analysée et traitée séparément. Le MRF ne peut donc pas simplement recopier la requête initiale en changeant la valeur de ν .

ensemble de modifications dans la vue, qui pourraient être décrites en VDLp puisque le langage est Turing-complet. Pour pouvoir interpréter ces actions de bon sens, il suffit donc pouvoir les *convertir* en des actions du langage de programmation. \mathcal{G}_p décrit donc simplement des correspondances entre des actions de bon sens et des actions bien définies dans le langage. Cependant, nous allons montrer que le mécanisme de conversion est beaucoup plus complexe que dans le cas des relations statiques, compte tenu de la nature même des actions.

4.2.5.2 Structure

Les éléments de \mathcal{G}_p sont des triplets de patterns (s, t, r) tels que :

- s est le pattern de ground procédural, représentant une action de bon sens ;
- t est le pattern associé dans le langage de programmation, représentant une action dont la sémantique opérationnelle es bien définie ;
- r est un pattern de relation booléenne qu'il faut instancier (par pm) et qui doit être vérifiée (par l'interprétation canonique ζ) pour que la correspondance entre s et t soit valide. Cette relation peut utiliser des patterns de ground issus de \mathcal{G}_{rs} ⁶.

Nous dirons que r définit une *contrainte* sur le pattern t .

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

$$\left(\begin{array}{l} increase[path[\#1], \#2] \\ put[path[\#1], plus[get[\#1], \#2]] \\ positive[\#2] \end{array} \right)$$

Un pattern de ground ne peut être associé qu'à un seul pattern du langage et, réciproquement, un pattern du langage contraint par une relation donnée ne peut être associé qu'à un seul pattern de ground :

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

Il y donc une bijection entre l'ensemble des patterns de ground procéduraux et celui des couples (pattern du langage, contrainte) : $s_j \longleftrightarrow (t_j, r_j)$.

Notation

Si $c \in A^*$, nous noterons abusivement $c \in \mathcal{G}_p$ le fait que c est le concept d'un terme du ground procédural, c'est-à-dire que $\exists (s, t, r) \in \mathcal{G}_p tq concept(s) = c$.

4.2.5.3 Utilisation dans le MLN

Du point de vue du TALN, toute action de bon sens est caractérisée par :

- Le sens *WordNet* associé au nom de l'action ;

⁶Dans ce cas, la fonction Ξ_{rs} est utilisée entre l'instanciation (pm) et l'évaluation (ζ) de la relation pour convertir les relations de bon sens en relations VDLp bien définies.

- Le nombre d’arguments de l’action et leur rôle (objet de l’action, argument, *etc.*).

Ces éléments sont utilisés lors du traitement de la référence et de la construction de la requête par le MLN.

L’ensemble de ces informations est représenté dans le concept VDL de la relation, de manière transparente pour le MRF. Nous ne les présenterons donc pas ici. Nous considérerons donc abusivement que le MLN connaît l’ensemble des patterns possibles en partie gauche de \mathcal{G}_p , pour pouvoir construire des requêtes de bon sens faisant intervenir ces relations.

4.2.5.4 Utilisation dans le MRF

Les actions de bon sens apparaissent uniquement dans les requêtes de type $\tau \in \{do, can, order\}$. Comme nous l’avons vu au chapitre précédent, pour traiter ces requêtes, le MRF doit rechercher dans la vue les actions correspondant au type de la requête (éventuellement référencées dans l’objet de la requête lorsque $\alpha \neq What$).

Interaction avec un utilisateur ordinaire

Dans le cas particulier des CSR, pour interpréter une action de bon sens dans un composant donné, le MRF doit vérifier la contrainte associée à cette action. En effet, deux actions de bon sens peuvent avoir la même interprétation en VDLp mais une contrainte différente (par exemple « *augmenter* » et « *diminuer* »).

Mais dans le cadre d’une interaction avec un utilisateur ordinaire, certains paramètres de l’action peuvent être *absents*. Par exemple, si l’utilisateur demande « *Est-ce que x augmente ?* », la valeur de l’incrément n’est pas donnée. Le MLN peut alors seulement construire une requête dans laquelle l’objet contient l’action de bon sens :

```
increase[path[x]]
```

Qui est appariable avec $increase[path[\#1], \#2]$, sans instancier le terme $\#2$.

Lorsque ces termes non instanciés interviennent dans la relation de contrainte r , le MRF ne peut pas évaluer cette relation dans le composant considéré. Dans notre exemple, le MRF ne peut pas évaluer $positive[\#2]$ puisqu’il ne sait pas ce que $\#2$ signifie (*i.e.* quel terme doit y être associé). Il ne peut donc pas déterminer si l’action de bon sens donnée dans la requête correspond ou non à une action VDLp bien définie.

Dans notre exemple, nous pouvons seulement obtenir :

```
t = put[path[x], plus[get[x], #2]]
r = positive[#2]
```

Étude d’une solution

Le MRF ne peut donc pas utiliser \mathcal{G}_p pour convertir directement une action de haut niveau sémantique en une action VDL, comme nous l’avons fait avec Ξ_{rs} . Il

peut en effet être incapable d'évaluer la valeur de la vérité de la relation de contrainte et, par conséquent, de traduire le terme.

Nous ne pouvons pas contourner ce problème en ignorant simplement la contrainte r et les termes $\#i$ non instanciés. En effet, cela pourrait conduire le MRF à trouver dans la vue des termes qui ne correspondent pas à l'action initialement recherchée. Dans notre exemple, le MRF rechercherait indifféremment des incréments et des décréments (lorsque le terme ajouté est négatif).

La solution que nous proposons pour résoudre ce problème consiste à vérifier la contrainte sur les actions trouvées par le MRF. L'interprétation d'une action de bon s'effectue donc en trois étapes :

1. Conversion du terme de ground t_g en un terme VDL t_{vdl} , en ignorant la relation r , à l'aide d'un algorithme Ξ_p similaire à Ξ_{rs} ;
Compte tenu de la propriété de bijection entre les termes de ground et les couples (actions VDL, contrainte), nous obtenons un seul terme VDL par action de bon sens ;
2. Recherche des termes correspondants à t_{vdl} dans la vue, en utilisant l'algorithme de recherche d'action χ_{action} , fondé sur la subsumption et présenté section 4.3.1.1 ;
Nous notons A l'ensemble des termes de la vue correspondant à t_{vdl} ;
3. Pour chaque élément a_i de A , conversion en un terme de ground a'_i , en vérifiant la relation r ;
Compte tenu de la propriété de bijection, nous obtenons un seul terme de ground par action VDL a_i vérifiant la relation.

Le résultat de la conversion est l'ensemble A dans lequel ont été retiré tous les termes a_i tels que $a'_i \not\leq t_g$. Dans notre exemple, $t_g = increase[path[x]]$ est subsumé par $increase[path[x], 3]$ mais pas par $decrease[path[x], -1]$. Ainsi, seule l'action correspondant à $increase$ sera conservée :

```
put [path [x] , plus [get [x] , 3]]
```

4.2.5.5 Algorithme

Principe général

L'interprétation des actions de bon sens s'appuie sur deux fonctions :

- La fonction de conversion $\Xi_p : \Upsilon \times \mathbf{B} \longrightarrow \Upsilon$, dont l'algorithme ressemble à celui de Ξ_{rs} , à ceci près qu'il vérifie la relation r dans le sens $VDL \rightarrow Ground$ ($b = \perp$).
- La fonction de filtrage $\Xi_p^{(filtre)} : \mathcal{P}(\Upsilon) \times \Upsilon \longrightarrow \mathcal{P}(\Upsilon)$ qui retire de l'ensemble A des actions trouvées par le MRF celles qui ne correspondent pas à l'action de ground t initialement considérée et transforme les autres en actions de ground, en utilisant la fonction Ξ_p dans le sens $VDL \rightarrow Ground$:

$$\Xi_p^{(filtre)}(A, t) = \{t' = \Xi_p(a_i, \perp), a_i \in A \text{ tq } t' \preceq t\}$$

Le résultat de ces deux opérations est donc un ensemble de termes. Cela correspond au fait que plusieurs actions correspondant à l'objet de la requête peuvent être trouvées par le MRF dans la vue par la fonction χ_{action} .

La figure 4.3 illustre le mécanisme de construction d'une réponse pour une requête faisant intervenir des actions de bon sens. La recherche des actions dans la vue, qui permet au MRF d'obtenir les éléments nécessaires à la construction d'une réponse, s'effectue entre l'appel à Ξ_p et l'appel à $\Xi_p^{(filtre)}$. Le traitement de la requête proprement dit et la construction de la réponse se fait après l'appel à $\Xi_p^{(filtre)}$.

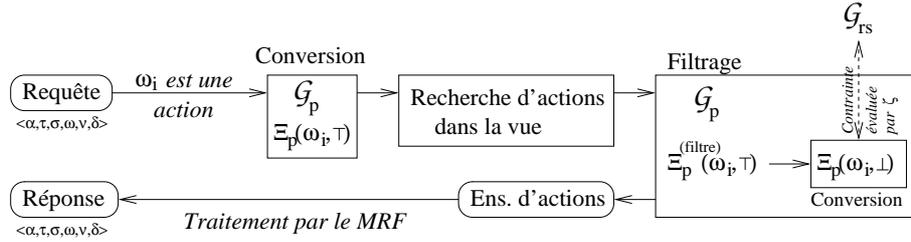


FIG. 4.3 – Traitement des requêtes procédurales

Ce schéma met en évidence que, contrairement au cas simple des relations statiques, le *Ground* procédural ne peut pas être perçu comme une simple couche de traduction entre des requêtes de bon sens et des requêtes VDL. Il doit être intégré dans les algorithmes de construction de réponse du MRF.

Fonction Ξ_p

Nous notons $\Xi_p : \Upsilon \times \mathbf{B} \rightarrow \Upsilon$ l'algorithme de conversion d'un terme de ground en terme VDL (et réciproquement) pour \mathcal{G}_p . Comme Ξ_{rs} , il prend en argument le terme t à convertir et un booléen b indiquant le sens de conversion.

Comme dans Ξ_{rs} , l'utilisation de $\min(res)$ correspond à l'hypothèse que les actions de bon sens construites par le MLN ne sont pas ambiguës : un terme donné ne s'apparie qu'avec un seul terme en partie gauche de \mathcal{G}_p . Les cas d'ambiguïté sont traités en amont par le MLN. Rappelons que dans l'instance d'un terme t' suivant une liste de substitution l , les $\#i$ non instanciés dans l sont supprimés.

Soulignons enfin que Ξ_p utilise la propriété de transparence de Ξ_{rs} : si la contrainte r n'est pas une relation de bon sens, $rinst^t = r$.

Propriétés

L'algorithme proposé pour Ξ_p est *robuste* (il retourne toujours un résultat) et *transparent* (le résultat est identique au terme initial si ce n'est pas une action de bon sens).

$\Xi_p(t, b)$ est calculable en $\mathcal{O}(|t|^2 \cdot |\mathcal{G}_p|^{|t|})$ dans le pire des cas (*i.e.* lorsque l'action t n'est pas un terme de ground).

Algorithm 2 fonction $\Xi_p(t \in \Upsilon, b \in \mathbf{B})$

```
si b, G = {s tq  $\exists(t',r)$  tq  $(s,t',r) \in \mathcal{G}_p$ }
sinon,
  R = {(t',r) tq  $\exists s$  tq  $(t,t',r) \in \mathcal{G}_p$ }
  G = {t' tq  $\exists r$  tq  $(t',r) \in R$ }
res =  $\xi(t,G)$ 
si res  $\neq \emptyset$ 
  soit t' = min(res) et (l,b') = pm(t,t', $\emptyset$ )
  si b
    retourner t' instancié suivant l
  sinon
    soit r la relation tq  $(t',r) \in R$ 
    soit rinst = r instancié suivant l
    soit rinst' =  $\Xi_{rs}(rinst, \top)$ 
    si  $\zeta(rinst')_{vue} = \{\text{true}\}$ 
      retourner t' instancié suivant l
  sinon
    soit c = concept(t) et  $\{st_1, \dots, st_n\} = \text{fils}(t)$ 
    retourner c[ $\Xi_{rs}(st_1, b), \dots, \Xi_{rs}(st_n, b)$ ]
```

4.2.5.6 Exemple

Considérons un composant actif muni d'une interface graphique affichant une représentation graphique en temps réel des paramètres d'un système sous surveillance, par exemple un moteur d'avion. Supposons que, constatant que l'un des paramètres est en augmentation, l'utilisateur demande : « *C'est quoi le truc qui monte ?* ». Le traitement de la référence structurelle et spatiale permet de construire la requête de bon sens suivante :

$$\langle \textit{What}, \textit{do}, \{\textit{view}\}, \{\textit{increase}\}, \top, \sqrt{\} \rangle$$

Pour répondre à cette requête, le MRF doit tout d'abord interpréter l'action de bon sens *increase* utilisée (sans aucun paramètre) dans l'objet de la requête. Pour ce faire, il doit :

1. Convertir $t = \textit{increase}$ en action VDL en utilisant Ξ_p .
Par construction, l'action *increase* est définie dans \mathcal{G}_p :

$$t' = \textit{increase}[\textit{path}[\#1], \#2] \in \xi(\textit{increase}, \mathcal{G}_p)$$

En utilisant $\textit{pm}(t, t', \emptyset)$, nous obtenons une liste de substitution vide. Nous aurons donc :

$$t_{vdl} = \Xi_p(t, \top) = \textit{increase}[\textit{path}, \textit{plus}[\textit{get}]]$$

2. Rechercher les actions correspondant à t_{vdl} dans la vue par χ_{action} .

Nous obtenons un ensemble d'actions A . Supposons, pour les besoins de notre exemple, que nous trouvions deux actions dans la vue :

$$A = \left\{ \begin{array}{l} a_1 = \text{put}[\text{path}[a], \text{plus}[\text{get}[b], -1]] \\ A_2 = \text{put}[\text{path}[a], \text{plus}[\text{get}[a], 1]] \end{array} \right\}$$

3. Filtrer A en utilisant la fonction $\Xi_p^{(\text{filtre})}$.

La première action a_1 ne peut pas être convertie une action de ground de concept *increase*, d'une part parce que $\#1$ ne peut pas être apparié à la fois avec a et b , et d'autre part parce que la relation n'est pas vérifiée pour *increase*. En effet, -1 est négatif donc l'appariement avec *positive* $[\#2]$ pour $l = \{(1, \{a\}), (2, \{-1\})\}$ donne *greater than* $[-1, 0]$ qui est évalué à *false*.

L'action a_2 peut être convertie en *increase* $[\text{path}[a], 1]$.

En raison de la forme de la requête (cf. section 4.3), le MRF produira alors la requête-réponse :

$$\langle \text{Assert}, \text{do}, \{\text{view}\}, \{\text{increase}[\text{path}[a]]\}, \top, \sqrt{\} \rangle$$

qui sera utilisée par le MLN pour produire la réponse en langue naturelle : « *C'est a qui augmente* ».

Autres actions de bon sens :

A titre d'exemple, nous aurons dans \mathcal{G}_p les actions de bon sens suivantes :

$$\begin{aligned} \text{increase}[\text{path}[\#1], \#2] &\Leftrightarrow \text{put}[\text{path}[\#1], \text{plus}[\text{get}[\#1], \#2]] \text{ et } \text{positive}[\#2] \\ \text{decrease}[\text{path}[\#1], \#2] &\Leftrightarrow \text{put}[\text{path}[\#1], \text{plus}[\text{get}[\#1], \#2]] \text{ et } \text{negative}[\#2] \\ \text{modify}[\text{path}[\#1]] &\Leftrightarrow \text{put}[\text{path}[\#1]] \\ \text{remove}[\text{path}[\#1]] &\Leftrightarrow \text{del}[\text{path}[\#1]] \end{aligned}$$

Signalons que l'action de bon sens *decrease* prend en argument $\#2$ une valeur *négative*. Le MLN doit utiliser cette connaissance pour modéliser, par exemple, « *diminuer x de 2* » par *decrease* $[\text{path}[x], 2]$.

4.2.6 Le *Ground* des relations procédurales : \mathcal{G}_{rp}

4.2.6.1 Objectif

L'objectif de \mathcal{G}_{rp} est de permettre au MRF de traiter les CSR contenant des relations de bon sens entre les actions, comme :

- « *f est l'inverse de g* » (relation binaire) ;
- « *compter à l'envers* » (relation unaire) ;
- « *ne rien faire* » (relation d'arité 0).

Il n'est pas possible de modéliser en VDL les relations de bon sens entre les actions. En revanche, nous pouvons donner chaque relation *en extension*, c'est-à-dire en énumérant l'ensemble des tuples d'actions élémentaires qui vérifient chaque relation.

4.2.6.2 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.

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

- T est un ensemble $\{t_1, \dots, t_n\}$ de patterns VDL,
- 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

Nous avons dans \mathcal{G}_{rp} :

$$\left(\begin{array}{c} 1 \\ \text{backwards} \\ \left\{ \left(\begin{array}{c} \{increase[path[\#1], \#2]\} \\ \text{negative}[\#2] \end{array} \right), \dots \right\} \end{array} \right)$$

qui définit la relation unaire « à l'envers » (*i.e.* « effectuer une actions à l'envers ») pour l'action de bon sens *increase* : « *incrémenter à l'envers* » est vrai si et seulement si l'incrément est négatif (*negative*[\#2]).

Notation

Si $c \in A^*$, nous noterons abusivement $c \in \mathcal{G}_{rp}$ le fait que c est le concept d'un terme du ground des relations procédurales, c'est-à-dire que $\exists(n, c, R) \in \mathcal{G}_{rp}$.

4.2.6.3 Utilisation dans le MLN

Du point de vue du MLN, une relation de bon sens entre les actions est caractérisée par son concept c_s et son arité n . Ces éléments sont utilisés pour le traitement de la référence et la construction de la CSR faisant intervenir des relations entre les actions. Le MLN a donc accès à l'ensemble des couples $(c_s, n)_i$ de \mathcal{G}_{rp} .

De plus, le concept c_s peut contenir des informations relatives au rôle de chaque argument de la relation. Nous n'aborderons pas ce point ici.

Définition

Dans l'objet ω d'une telle requête, les relations procédurales sont représentées par un terme de la forme :

$$c_s[t_1, \dots, t_n]$$

Nous dirons que les termes t_1, \dots, t_n sont les *arguments* de la relation procédurale dans la requête.

4.2.6.4 Utilisation dans le MRF

Puisque les relations dans \mathcal{G}_{rp} sont données en extension, le *Ground* des relations procédurales doit être utilisé directement dans les algorithmes de construction de réponse du MRF pour déterminer la valeur de vérité d'une relation donnée dans l'objet ω d'une requête.

Principe général

Le mécanisme de construction d'une réponse pour une requête faisant intervenir des relations procédurales est illustré sur la figure 4.4. Le MRF recherche tout d'abord dans la vue les actions correspondant à chaque argument de la relation. Il évalue alors la valeur de vérité de la relation pour toutes les combinaisons possibles des actions élémentaires trouvées pour chaque argument, afin de pouvoir construire une réponse.

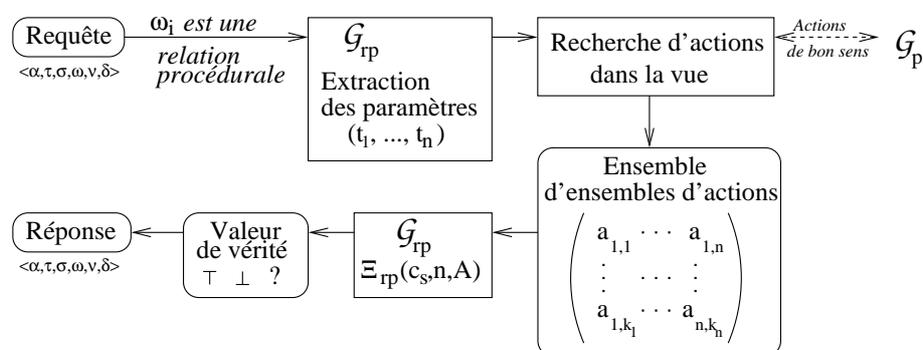


FIG. 4.4 – Traitement des relations procédurales

Comme pour le *Ground* procédural, le *Ground* des relations procédurales ne peut pas être perçu comme une simple couche de traduction entre des requêtes de bon sens et des requêtes VDL. Il doit être intégré dans les algorithmes de construction de réponse du MRF (à un degré même supérieur à \mathcal{G}_p puisqu'il est utilisé directement dans le traitement de la requête).

Utilisation dans les requêtes

Il faut distinguer deux sortes de relations procédurales :

- Les relations exprimant des propriétés *statiques* sur les actions de la vue (par exemple, *être l'inverse de*). Ces relations seront donc utilisées dans des requêtes de type *is*.
- Les relations concernant l'*activité* et le *comportement* de la vue (par exemple, *ne rien faire*). Elles seront utilisées dans des requêtes de type *do* et *can*, puisqu'elles portent sur l'activité ou le comportement de la vue.

L'analyse de notre corpus montre que les relations d'ordre 0 sont généralement des relations concernant l'activité ou le comportement de la vue alors que les relations d'ordre $n \geq 2$ expriment généralement des propriétés entre des actions quelconques. Cependant, la requête suivante est parfaitement correcte :

$$\langle Ask, is, \emptyset, \{nothing\}, \top, \surd \rangle$$

Elle modélise la question : « *Est-ce qu'il n'y a aucune action dans la vue ?* ».

4.2.6.5 Algorithme

Définition

Soit $\{t_1, \dots, t_n\}$ un ensemble de termes VDL représentant des actions. Soit c_s une relation procédurale d'arité n . La relation est **vérifiée** pour les actions $\{t_1, \dots, t_n\}$ ssi il existe un ensemble R tel que $(n, c_s, R) \in \mathcal{G}_{rp}$ et un ensemble de patterns $(\{t'_1, \dots, t'_n\}, r) \in R$ tels que :

- Il existe $l \in \mathbf{N} \times \Upsilon$ telle que $\forall i \in \mathbf{N}, pm(\Xi_p(t_i, \top), t'_i, \emptyset) = (l, \top)$ et, pour les relations procédurales statiques ($\tau = is$), $\forall i \in \mathbf{N}, \Xi_p(t'_i \text{ instancié suivant } l, \perp) = t_i$ (pour vérifier la relation de contrainte associée à l'action de bon sens) ;

Sinon, nous dirons que la valeur de vérité de la relation est **indéterminée** ;

- L'instance de r suivant l est valide, au sens de l'interprétation canonique ζ dans la vue $(\zeta(\Xi_{rs}(r, \top), \emptyset)_{vue} = \{true\})$;

Sinon, nous dirons que la relation **n'est pas vérifiée**.

Par exemple, la relation unaire *backwards* donnée plus haut est :

- vérifiée par *increase[path[toto], -1]* ;
- non vérifiée par *decrease[path[toto], 1]* ;
- indéterminée pour *put[path[toto], titi]*.

Fonction Ξ_{rp}

Nous notons $\Xi_{rp}(c_s, n, \{t_1, \dots, t_n\}) \in \{\top, \perp, ?\}$ la valeur de vérité de la relation caractérisée par le couple (c_s, n) pour un ensemble d'actions $t_1, \dots, t_n : \top$ si la relation est vérifiée, ? si elle est indéterminée et \perp si elle n'est pas vérifiée.

L'algorithme de la fonction Ξ_{rp} est donné page 156. Il utilise le type τ de la requête pour déterminer s'il faut vérifier la contrainte lors de l'utilisation d'actions de bon sens.

Fonction Ξ_{rp}^*

Nous notons $\Xi_{rp}^* : A^* \times \mathbf{N} \times \mathcal{P}(\mathcal{P}(\Upsilon)) \longrightarrow \mathbf{B}$ la fonction récursive qui calcule la valeur de vérité d'une relation, caractérisée par un couple (c_s, n) , pour un ensemble A d'ensembles d'actions A_i trouvées dans la vue. Elle retourne \top si et seulement si :

- Il existe au moins une combinaison $\{k_1, \dots, k_{|A|}\}$ telle que :

$$\Xi_{rp}(c_s, n, \{a_{1,k_1}, \dots, a_{n,k_n}\}) = \top$$

où pour tout $i \in [1, |A|]$, $a_{i,k_i} \in A_i$;

- Toutes les autres combinaisons vérifient $\Xi_{rp}(c_s, n, \{a_{1,k_1}, \dots, a_{n,k_n}\}) = ?$.

Par définition :

$$\Xi_{rp}^*(c_s, n, A) = (f_{rp}^*(c_s, n, A, \emptyset) = \top)$$

où $f_{rp}^* : A^* \times \mathbb{N} \times \mathcal{P}(\mathcal{P}(\Upsilon)) \times \mathcal{P}(\Upsilon) \longrightarrow \{\top, \perp, ?\}$ est la fonction récursive définie par l'algorithme 3. Elle construit dans R une combinaison possible et calcule sa valeur de vérité.

Algorithm 3 fonction $f_{rp}^*(n \in \mathbb{N}, c_s \in A^*, A \in \mathcal{P}(\mathcal{P}(\Upsilon)), R \in \mathcal{P}(\Upsilon))$

```

si A=∅, retourner  $\Xi_{rp}(c_s, n, R)$ 
sinon
  ok =  $\perp$ 
  pour i allant de 1 à  $|A[1]|$ 
    res =  $f_{rp}^*(c_s, n, A \setminus A[1], R \cup A[1][i])$ 
    si res =  $\perp$ , retourner  $\perp$ 
    si res =  $\top$ , ok =  $\top$ 
  si ok, retourner  $\top$ 
  sinon, retourner ?

```

Remarques sur l'algorithme de Ξ_{rp}

1. Dans Ξ_{rp} , nous n'appariions pas chaque $\Xi_p(t_i, \top)$ avec t'_i , mais l'ensemble des $\{\Xi_p(t_i, \top)\}$ avec l'ensemble des $\{t'_i\}$: la liste de substitution l doit être la même à chaque fois.

D'un point de vue algorithmique, il suffit d'apparier chaque $\Xi_p(t_i, \top)$ avec t'_i par pm en utilisant la liste l résultat de l'appariement précédent, puisque chaque $\#i$ ne peut s'apparier que d'une seule façon (à un terme ou à une référence, selon les cas).

2. De même, nous pouvons remarquer qu'un terme t peut éventuellement s'apparier avec plusieurs patterns t'_i d'un ensemble T donné.

Dans les algorithmes, nous ne considérerons que l'ordre des attributs dans la relation est exactement l'ordre des paramètres de la relation.

3. Pour les relations procédurales statiques, la contrainte sur les actions de bon sens est vérifiée. Par conséquent, chaque $\Xi_p(t_i, \top)$ ne peut s'apparier avec un t'_i que dans un seul T de l'ensemble caractéristique R .

En revanche, pour les relations procédurales portant sur l'activité et le comportement, la contrainte n'est pas (et ne doit pas être) vérifiée. En effet, elle

peut être en contradiction avec la contrainte r associée à T (comme *backwards* pour l'action de bon sens *increase*) puisque qu'il s'agit de définir une « variation » sur les actions du composant. L'algorithme prend alors en considération le premier ensemble T pouvant s'apparier avec l'ensemble des $\{\Xi_p(t_i, \top)\}$, s'il y en a plusieurs.

Algorithm 4 fonction $\Xi_{rp}(n \in \mathbb{N}, c_s \in A^*, Arg \in \mathcal{P}(Y))$

```

soit R tq  $(|T|, c_s, R) \in \mathcal{G}_{rp}$ , trouvé =  $\perp$  et  $i = 1$ 
tant que (non trouvé)
  soit  $(T, r) = R[i]$ , succès =  $\top$ ,  $l = \emptyset$ 
  pour j allant de 1 à  $|Arg|$ 
     $t = Arg[j]$ , ok =  $\perp$  et  $k = 1$ 
    tant que (non ok)
      si  $pm(\Xi_p(t, \top), t', l) = (l', \top)$ 
        fini =  $\top$ 
        si  $\tau = is$ ,
          soit  $t'' = t'$  instancié suivant  $l'$ 
          si  $\Xi_p(t'', \perp) = t$ , fini =  $\perp$ 
        si fini,
          ok =  $\top$ ,  $l = l'$  et  $T = T \setminus T[i]$ 
      si (non ok), succès =  $\perp$ 
    si succès, trouvé =  $\top$ 
  si trouvé =  $\perp$ , retourner ?
sinon
  soit  $rinst = r$  instancié suivant  $l'$ 
  si  $\zeta(\Xi_{rs}(rinst, \top), \emptyset)_{vue} = \{true\}$ , retourner  $\top$ 
  sinon, retourner  $\perp$ 

```

Relations d'arité 0

Le calcul des valeurs de vérité se fait dans les algorithmes de traitement suivant un mécanisme légèrement différent. Nous avons les relations d'arité 0 suivantes dans \mathcal{G}_{rp} et nous dirons que :

- *nothing* est vrai ssi aucune action n'est effectuée ;
- *something* est vrai ssi au moins une action est effectuée ;
- *everything* est vrai ssi toutes action sont effectuées.

Ces relations sont donc données en *intention*.

La notion d'action « effectuée » dépend du type de la requête (τ) qui détermine si l'utilisateur parle du comportement du composant ou de son activité. Par exemple, la question « *Pourquoi est-ce que tu ne peux rien faire ?* » :

$$\langle Why, can, \{view\}, \{nothing\}, \top, \sqrt{\} \rangle$$

porte sur toutes les actions (processus et réactions) de la vue, alors que « *Pourquoi est-ce que tu ne fais rien ?* » :

$$\langle \textit{Why}, \textit{do}, \{\textit{view}\}, \{\textit{nothing}\}, \top, \sqrt{\} \rangle$$

porte uniquement sur les processus (activité du composant).

4.2.6.6 Exemple

Nous ne pouvons pas présenter en détail le mécanisme de construction de réponse pour une relation procédurale, puisqu'il est fortement imbriqué avec le traitement proprement dit de la requête dans le MRF, présenté section 4.3. Cependant, considérons les deux exemples suivants dans *Coco* :

Relation portant sur l'activité

Si l'utilisateur demande à *Coco* : « *Est-ce que tu comptes à l'envers ?* », le MLN construit la requête :

$$\langle \textit{Ask}, \textit{do}, \{\textit{view}\}, \{\textit{backwards}[\textit{count}]\}, \top, \sqrt{\} \rangle$$

Pour répondre à cette requête, le MRF doit :

1. Rechercher l'action *count* dans la vue, en utilisant la fonction $\chi_{\textit{action}}$.
Nous notons $A_{\textit{count}} = \{\textit{put}[\textit{path}[\textit{value}], \textit{plus}[\textit{get}[\textit{value}], \textit{get}[\textit{speed}]]]\}$ l'ensemble des actions élémentaires utilisées dans l'action *count* et $A = \{A_{\textit{count}}\}$.
2. Vérifier la relation *backwards* sur les actions élémentaires de *count*. La valeur de vérité de la relation est donnée par $\Xi_{rp}^*(\textit{backwards}, 1, A)$, en utilisant $f_{rp}^*(\textit{backwards}, 1, A, \emptyset)$. Le déroulement de l'algorithme appelle une seule fois la fonction récursive :

$$f_{rp}^*(\textit{backwards}, 1, \emptyset, A_{\textit{count}}) = \Xi_{rp}(\textit{backwards}, 1, A_{\textit{count}})$$

Supposons que la vitesse de *Coco* (donnée par l'attribut de *speed*) est positive. Sans dérouler entièrement l'algorithme Ξ_{rp} , nous pouvons affirmer que $A_{\textit{count}}$ peut s'apparier avec $T = \{\textit{increase}[\textit{path}[\#1], \#2]\}$ (la relation de contrainte est ignorée puisque $\tau = \textit{do}$).

La relation *r* associée à *T* dans *backwards* est alors instanciée en :

$$r = \textit{negative}[\textit{get}[\textit{speed}]]$$

Puisque $\zeta(r)_{\textit{vue}} = \{\textit{false}\}$, la relation n'est pas vérifiée.

Le MRF construit alors la réponse :

$$\langle \textit{Assert}, \textit{do}, \{\textit{view}\}, \{\textit{backwards}[\textit{count}]\}, \perp, \sqrt{\} \rangle$$

qui sera utilisée par le MLN pour produire : « *Non.* ».

Nous verrons dans les algorithmes du MRF que si $\tau = \textit{can}$, la relation *r* n'est pas vérifiée et la réponse sera alors : « *Oui, je peux compter à l'envers.* ».

Relation exprimant une propriété

Si l'utilisateur demande à *Coco* : « *Est-ce que start est l'inverse de stop ?* », le MLN construit la requête :

$$\langle Ask, is, \{view\}, \{inverse[start, stop]\}, \top, \sqrt{\} \rangle$$

Pour répondre à cette requête, le MRF doit :

1. Rechercher les actions *start* et *stop* dans la vue et extraire leurs actions élémentaires. Soit *A* l'ensemble des singletons d'actions élémentaires pour *start* et *stop* :

$$A = \{\{put[path[running], true]\}, \{put[path[running], false]\}\}$$

2. Vérifier la relation *inverse* sur *A*. Le déroulement de l'algorithme appelle une seule fois :

$$\Xi_{rp}(inverse, 2, \{put[path[running], true], put[path[running], false]\})$$

L'appariement des actions élémentaires avec les actions proposées dans \mathcal{G}_{rp} , en vérifiant la contrainte pour les actions de bon sens, permet de déterminer que la relation est vérifiée.

Le MRF construit alors la réponse :

$$\langle Assert, is, \{view\}, \{inverse[start, stop]\}, \top, \sqrt{\} \rangle$$

qui sera utilisée par le MLN pour produire : « *Oui.* ».

Autres relations procédurales

A titre d'exemple, nous aurons les relations suivantes dans \mathcal{G}_{rp} ⁷ :

- Pour les relations d'arité 0, *nothing*, *something* et *everything* ;
- Pour les relations d'arité 1 sur l'activité et le comportement :

$$\left(\begin{array}{c} 1 \\ backwards \\ \left\{ \left(\begin{array}{c} \{increase[path[\#1], \#2]\} \\ negative[\#2] \end{array} \right), \dots \right\} \end{array} \right)$$

- Pour les relations d'arité 1 statiques :

$$\left(\begin{array}{c} 1 \\ idempotent \\ \left\{ \left(\begin{array}{c} \{del[path[\#1], \#2]\} \\ true \end{array} \right), \left(\begin{array}{c} \{put[path[\#1], \#2]\} \\ equals[get[\#1], \#2] \end{array} \right) \right\} \end{array} \right)$$

⁷Nous ne donnons pas ici l'ensemble des actions élémentaires intervenant dans la définition de la relation en extension.

$$\left(\left\{ \left(\begin{array}{c} 1 \\ \text{inverse} \\ \{ \text{increase}[\text{path}[\#1], \#2] \} \\ \text{true} \end{array} \right), \dots, \left(\begin{array}{c} \{ \text{put}[\text{path}[\#1], \text{true}] \} \\ \text{true} \end{array} \right), \dots \right\} \right)$$

– Pour les relations d'arité 2 (statiques) :

$$\left(\left\{ \left(\begin{array}{c} 2 \\ \text{inverse} \\ \{ \text{increase}[\text{path}[\#1], \#2], \text{decrease}[\text{path}[\#1], \#2] \} \\ \text{true} \end{array} \right), \dots \right\} \right)$$

$$\left(\left\{ \left(\begin{array}{c} 2 \\ \text{contains} \\ \{ \#1, \#2 \} \\ \text{equals}[\text{"size ref"}][\text{node}[\text{concept}[\#1]], \#2], 1 \end{array} \right) \right\} \right)$$

Nous pouvons remarquer que certaines relations peuvent être données en *intension* (c'est-à-dire sans énumérer l'ensemble des actions entrant en relations, mais en donnant une relation calculable qui la caractérise) dans \mathcal{G}_{rp} . C'est pas exemple le cas de la relation *contains*. Nous pouvons aussi noter que, dans cette relation, l'ordre des attributs est important, comme dans la relation statique VDL "*greater than*".

4.2.7 Bilan sur les CSR

Le problème de la prise en compte des notions de bon sens dans des requêtes modélisant des questions sur le fonctionnement n'a pas encore été étudié dans la littérature. Les modèles logiques de raisonnement sur les actions, issus des travaux de la communauté du raisonnement de bon sens en IA, proposent des mécanismes de raisonnement génériques. Cependant, ces outils ne peuvent travailler que sur les prédicats définis par l'expert lors de la description du système, parmi lesquelles ne figurent pas de connaissances « de bon sens ».

Pourtant, la prise en compte des utilisateurs ordinaires nécessite d'interpréter des notions de bon sens dans le contexte d'un composant particulier. Nous pensons qu'elle constitue une étape importante vers l'intégration des outils d'interactions naturels (le geste et la parole) avec les formalismes de raisonnement sur la structure et les actions d'un composant.

Contrairement à l'approche « logique », en TALN, les systèmes de dialogue homme-machine pour la réalisation d'une tâche utilisent généralement une approche « tout intégré », comme dans [5, 4], pour prendre en compte des utilisateurs ordinaires. Les connaissances de bon sens ne sont alors pas représentées en tant que telle, mais interprétées dans le contexte de la tâche à réaliser pour munir le système de connaissances *spécifiques*. Le système ne propose donc pas de mécanisme générique de raisonnement.

Le cadre formel que nous avons proposé dans cette section, appelé *Ground*, est une première proposition pour la modélisation formelle et la manipulation des notions de bon sens sur le fonctionnement, dans le cadre d'une interaction homme-machine générique. Ce modèle reste toutefois très limité et nous pensons pouvoir l'améliorer sur plusieurs points :

1. Dans \mathcal{G}_p (et, par conséquent, dans \mathcal{G}_{rp}), chaque action de bon sens ne peut correspondre qu'à une seule action élémentaire. Nous envisageons de proposer un modèle étendu dans lequel une action de bon sens pourrait être définie par un ensemble d'actions élémentaires devant être exécutées simultanément, en un cycle.
2. Le traitement des relations portant sur l'activité et le comportement, comme *backwards*, reste insuffisant pour prendre en compte l'ensemble des notions de bon sens utilisées par un utilisateur ordinaire lors d'une interaction. En particulier, il ne s'appuie que sur analyse statique (à un instant donné) de la vue. Par exemple, la relation proposée dans \mathcal{G}_{rp} pour *backwards* est vraie pour *increase* comme pour *decrease* : le MRF ne peut pas comprendre que « compter à l'envers » n'est vrai que si la vitesse est négative. Seule une analyse dynamique permettrait de détecter le changement de vitesse du positif vers le négatif, permettant d'affirmer que *Coco* compte à l'envers.

Nous proposons au chapitre 5 une approche dynamique de l'analyse du composant. Nous montrerons comment elle peut-être utilisée pour modéliser ce genre de notions de bon sens de manière plus exacte.

Le modèle que nous avons proposé est fortement lié aux algorithmes de traitements du MRF que nous présentons dans la section suivante. Cependant, bien que nous ayons utilisés une syntaxe VDL, il est parfaitement adaptable à d'autres langages de programmation ou formalismes de représentations, comme LISP ou le calcul des situations. Le problème des notions de bon sens sur le fonctionnement dépasse largement le cadre du projet *InterViews* et du langage VDLp que nous avons proposé.

Dans la section suivante, nous présentons, pour chaque acte α et pour chaque type τ , les algorithmes de traitement utilisés au sein du MRF pour répondre aux requêtes envoyées par le MLN. Nous verrons comment le *Ground* est utilisé dans le MRF pour prendre en compte les CSR.

4.3 Traitement des requêtes

Dans cette section, nous présentons les mécanismes utilisés par le MRF pour répondre aux requêtes formelles présentées au chapitre 3 modélisant des questions de contrôle ou de commande. Nous présentons tout d'abord deux algorithmes généraux qui permettent de rechercher des actions spécifiques dans la vue. Nous donnons ensuite, pour chaque acte α et pour chaque type τ , les algorithmes de construction de réponse, en présentant les problèmes soulevés par la classe de requêtes considérée.

Mémorisation des événements

Nous ne traitons pas ici les requêtes temporelles. Cela signifie que nous aurons toujours $\delta = \surd$ dans les requêtes. Cependant, certains algorithmes s'appuieront sur une mémorisation des événements externes envoyés à la vue. Dans ce chapitre, nous considérerons que tous les événements envoyés à la vue sont mémorisés avec le numéro du cycle où il ont été envoyés.

Nous noterons $\Theta \in \mathcal{P}(\Upsilon \times \mathbb{N})$ le tableau dans lequel ces envois sont mémorisés et, à tout instant de l'exécution, nous noterons θ le numéro du cycle d'exécution courant.

Rappel : réponse *pertinente*

L'objectif du MRF n'est pas simplement de construire des réponses formelles à des requêtes formelles. En effet, nous nous plaçons dans le cadre d'une interaction entre un composant actif et un utilisateur ordinaire. Cela signifie que nous souhaitons retourner des réponses aussi naturelles que possible (*i.e.* proches de celles que donnerait un humain) aux questions de l'utilisateur, comme nous l'avons expliqué dans la première section de ce chapitre.

Nous avons vu que cela impliquait de pouvoir prendre en compte des notions de bon sens dans les requêtes formelles. Cela implique aussi de donner une réponse *pertinente* au MLN, c'est-à-dire contenant toutes les informations nécessaires à la construction d'une réponse naturelle, et uniquement celles-ci.

Dans les modèles de requêtes formelles usuels, le système donne généralement une réponse complète (toutes les informations sont présentes), mais non pertinente (il y a trop d'information pour que la réponse soit naturelle). Dans cette section, nous donnons des algorithmes de construction de réponses *pertinentes*.

4.3.1 Algorithmes de recherche d'action

Comme nous l'avons vu dans les exemples précédents, la construction d'une réponse à une requête s'appuie sur la recherche d'actions dans la vue. Plus formellement, pour pouvoir traiter une requête faisant intervenir des actions, le MRF doit pouvoir :

- Rechercher les termes de la vue qui correspondent aux actions données dans l'objet ω de la requête ;
- Pour chaque action de la vue, déterminer s'il s'agit d'une réaction ou d'un processus, et si cette action est active ou non.

Pour ce faire, il doit donc pouvoir extraire les *préconditions* de chaque action. Dans cette sous-section, nous présentons deux algorithmes qui permettent :

1. de rechercher des actions dans la vue à partir d'un ensemble de l'objet d'une requête (c'est-à-dire d'un ensemble de termes VDL) ,
2. de calculer les préconditions de ces actions et leur valeur de vérité.

4.3.1.1 Recherche d'action

Rappel : *nom* d'une action

Par construction (*cf.* section 2.3.4.2 page 63), toute action VDL est munie d'un *nom* qui est le concept englobant des termes procéduraux utilisés pour définir l'action (*guard*, *event*, *add*, *put* ou *del*). Le nom d'une action est utilisé par le MLN pour associer les termes de la question de l'utilisateur avec les actions de la vue, à partir du sens *WordNet* et des informations linguistiques mémorisés dans le concept VDL.

Dans les requêtes formelles, nous pouvons donc avoir :

- Soit des nom d'actions VDL définies dans le composant ;
- Soit des actions de bon sens, définies dans G_p , qui peuvent être interprétées par des actions élémentaires « partielles », c'est-à-dire dans lesquelles certains sous-termes peuvent manquer, parce que les variables $\#i$ n'ont pas été instanciée.

Nous devons donc proposer un algorithme de recherche d'action qui peut trouver indifféremment des actions VDL à partir de leur nom ou des actions élémentaires à partir d'une action élémentaire partielle.

Fonction χ_a

Nous notons $\chi_a : \Upsilon \times \Upsilon \longrightarrow \mathcal{P}(\Upsilon)$ la fonction récursive qui recherche dans un terme t toutes les actions correspondant au second argument a . Elle retourne l'ensemble des actions trouvées. Si $a = \surd$, l'algorithme retourne l'ensemble de toutes les actions dans t .

Algorithm 5 fonction $\chi_a(t \in \Upsilon, a \in \Upsilon)$

```
si fils(t)= $\emptyset$  ou concept(t) $\in$ {guard,event}, retourner  $\emptyset$ 
si concept(t) $\in$ {add,put,del}
  si (a= $\surd$ ) ou (a $\succeq$ t), retourner {t}
  sinon, retourner  $\emptyset$ 
sinon
  soit E = {t' $\in$ fils(t) tq concept(t')=event}
  soit G = {t' $\in$ fils(t) tq concept(t')=guard}
  soit A = {t' $\in$ fils(t) tq concept(t') $\in$ {add,put,del}}
  si (G $\neq$  $\emptyset$  ou A $\neq$  $\emptyset$  ou E $\neq$  $\emptyset$ )
    et ((a= $\surd$ ) ou (concept(t)=concept(a)))
    retourner {t}
  sinon, retourner  $\bigcup_{t' \in \text{fils}(t)} \chi_a(t', a)$ 
```

Étant donné que nous ne travaillons que sur des termes VDL (c'est-à-dire sur des arbres) en utilisant des parcours en profondeur, tous nos algorithmes ont une complexité en $\mathcal{O}(n^2)$ où n est le nombre de nœuds du terme.

Utilisation : fonction χ_{action}

Pour le traitement des requêtes, nous devons pouvoir rechercher des actions dans une action. Le terme t passé initialement ne doit donc pas être pris en compte dans la recherche. C'est la raison pour laquelle, dans les algorithmes du MRF, nous utiliserons la fonction $\chi_{action} : \Upsilon \times \Upsilon \longrightarrow \mathcal{P}(\Upsilon)$ qui appelle χ_a sur chacun de ses attributs :

$$\chi_{action}(t, a) = \bigcup_{t' \in fils(t)} \chi_a(t', a)$$

Remarque

En VDLp, nous utilisons une description hiérarchique des actions : une action a , munie d'un ensemble de préconditions p_a , est composée d'actions a_i , munies chacune de leurs préconditions et sous-actions éventuelles. C'est l'exécution des actions a_i (selon leurs préconditions) qui définit l'action a .

Cette structure est utilisée dans la construction de réponse aux questions de l'utilisateur. Par exemple⁸, si l'on considère un robot qui, pour distribuer le courrier (action a), doit prendre l'ascenseur (action a_i), alors à la question « *Qu'est-ce que tu fais ?* », nous répondrons : « *Je distribue le courrier.* » (action a englobante). Mais à la question « *Comment distribues-tu le courrier.* » ou « *Que fais-tu, là, quand tu distribues le courrier.* », nous répondrons : « *Je prends l'ascenseur.* » (action b , première englobante des sous-actions de a). Et ainsi de suite jusqu'aux actions élémentaires.

La structure arborescente des actions VDL ainsi que le mécanisme de recherche proposée permet donc de construire des réponses pertinentes aux questions de l'utilisateur.

4.3.1.2 Recherche de préconditions

Rappel : classe d'une action

Les actions VDL peuvent être classées en quatre groupes en fonction de leur préconditions : processus actifs, processus inactifs, réactions possibles ou réactions impossibles (cf. section 2.3.4.2 page 63). Comme nous l'avons montré au chapitre 3, le type τ d'une requête définit la classe de fonctionnement qui doit être prise en considération pour la construction de la réponse.

Nous devons donc proposer un algorithme de recherche et d'analyse de préconditions qui met en évidence la classe d'une action donnée.

Fonction χ_p

Nous notons $\chi_p : \Upsilon \times \mathcal{P}'(\Upsilon) \longrightarrow \mathcal{P}(\Upsilon)$ ³ la fonction qui recherche les préconditions d'une action a en remontant dans la vue à partir de l'action considérée pour mettre en évidence la classe de l'action. Afin de pouvoir vérifier les gardes des réactions qui

⁸Merci à S. Shapiro de m'avoir proposé cet exemple.

feraient intervenir les événements en cours de traitement (par *event get*), la fonction χ_p prend en second argument un ensemble d'événements externes E . Elle construit alors trois ensembles de préconditions :

- L'ensemble Π_{\top} des *gardes valides* (cf. section 2.3.4) de l'action a ;
- L'ensemble Π_{\perp} des *gardes invalides* de l'action a ;
- L'ensemble \mathcal{E} des événements externes de l'action a .

Algorithm 6 fonction $\chi_p(t \in \Upsilon, E \in \mathcal{P}(\Upsilon))$

```

si concept(t) ∈ {guard, event}, retourner (∅, ∅, ∅)
 $\mathcal{E} = \Pi_{\top} = \Pi_{\perp} = \emptyset$ 
pour i allant de 1 à |fils(t)|
  t' = fils(t)[i]
  si concept(t')=event,  $\mathcal{E} = \mathcal{E} \cup \text{fils}(t')$ 
  si concept(t')=guard
    f =  $\bigcup_{t_i \in \text{fils}(t')} \zeta(t_i, E)$  et v =  $\top$ 
    si f=∅, v =  $\perp$ 
    sinon pour j allant de 1 à |f|
      si (f[j] ≠ true), v =  $\perp$ 
    si v,  $\Pi_{\top} = \Pi_{\top} \cup \text{fils}(t')$ 
    sinon  $\Pi_{\perp} = \Pi_{\perp} \cup \text{fils}(t')$ 
si pere(t)=√, retourner (∅, ∅, ∅) (on a atteint la racine)
sinon
  ( $\mathcal{E}', \Pi'_{\top}, \Pi'_{\perp}$ ) =  $\chi_p(\text{pere}(t), E)$ 
  retourner ( $\mathcal{E}' \cup \mathcal{E}, \Pi'_{\top} \cup \Pi_{\top}, \Pi'_{\perp} \cup \Pi_{\perp}$ )

```

La valeur de vérité d'une garde est calculée en utilisant l'interprétation canonique ζ et la règle de validité d'une garde VDL $t = \text{guard}[t_1, \dots, t_n]$ à l'instant courant :

$$\forall t' \in \bigcup_{i=1}^n \zeta(t_i, \emptyset), t' = \{true\}$$

L'algorithme est robuste : il peut être appelé sur tout terme t , même si ce n'est pas une action, et même si c'est un terme flottant.

Classe de l'action

La classe de l'action est alors déterminée de la manière suivante :

- Si $\mathcal{E} = \emptyset$ et $\Pi_{\perp} \neq \emptyset$, l'action a est un processus inactif ;
- Si $\mathcal{E} = \emptyset$ et $\Pi_{\perp} = \emptyset$, l'action a est un processus actif⁹ ;
- Si $\mathcal{E} \neq \emptyset$ et $\Pi_{\perp} \neq \emptyset$, l'action a est une réaction impossible ;
- Si $\mathcal{E} \neq \emptyset$ et $\Pi_{\perp} = \emptyset$, l'action a est une réaction possible.

⁹Remarquons que cela inclut le cas $\Pi_{\top} = \emptyset$, conformément à la sémantique opérationnelle de VDLp définie par ϕ_{exec} .

Nous verrons aussi que l'analyse de ces trois ensembles permet de produire des réponses appropriées aux questions de l'utilisateur.

Utilisation : fonction $\chi_{precond}$

Pour matérialiser le fait qu'une action n'ayant aucune garde invalide est toujours active, nous mettrons $\{true\}$ dans Π_{\top} . Les algorithmes du MRF utiliseront donc la fonction $\chi_{precond} : \Upsilon \times \mathcal{P}(\Upsilon) \longrightarrow \mathcal{P}(\Upsilon)^3$ définie par :

$$\chi_{precond}(t, E) = \begin{cases} (\{true\}, \emptyset, \mathcal{E}) & \text{si } \chi_p(t, E) = (\emptyset, \emptyset, \mathcal{E}) \\ \chi_p(t, E) & \text{sinon} \end{cases}$$

Notations

Pour toute action VDL $a \in \Upsilon$, nous notons $(\Pi_{\top}^{(a)}, \Pi_{\perp}^{(a)}, \mathcal{E}^{(a)}) = \chi_{precond}(a, \emptyset)$. Si a est une réaction dans laquelle certaines gardes utilisent l'événement externe pour calculer leur valeur de vérité (elles contiennent un sous-terme de concept "event get"), ces gardes seront alors toujours fausses.

Exemple

Si a est le terme de concept *take* dans le composant *Jojo* présenté page 86 et si la « main » est vide, nous aurons :

$$\begin{aligned} \Pi_{\top}^{(a)} &= \{equals[size[get[taken, cell]], 0]\} \\ \Pi_{\perp}^{(a)} &= \{equals["size ref" ["event get" [take]], 1]\} \\ \mathcal{E} &= \{take[figure]\} \end{aligned}$$

4.3.1.3 Un problème de référence

Lors du traitement des requêtes sur le fonctionnement $(\tau \in \{do, can, order\}$ et $(\tau, \sigma) = (is, \emptyset)$), nous sommes confrontés à un problème de *référence*. En effet, l'objet ω d'une requête peut contenir plusieurs termes, chacun modélisant une action ou une relation différente. Et même lorsque $|\omega| = 1$, ω_1 peut faire référence à plusieurs actions dans la vue ou au contraire à aucune. Par exemple, si $\omega_1 = count$, plusieurs actions de nom *count* peuvent exister dans la vue.

Nous dirons qu'un terme t modélise une action dans une requête si et seulement si il apparaît dans l'objet ω :

- Soit directement $\exists i tq \omega_i = t$;
- Soit comme argument d'une relation procédurale : $\exists i tq t \in fils(\omega_i)$.

Pour tout $t \in \Upsilon$ modélisant une action, nous noterons A_t l'ensemble des actions référencées par t :

$$A_t = \Xi_p^{(filtre)}(\chi_{action}(vue, \Xi_p(t, \top)), t)$$

Lors de la construction des réponses aux requêtes, nous sommes alors confrontés à plusieurs situations qu'il faut traiter au cas par cas :

- Certaines actions (éventuellement toutes) n'ont pas pu être trouvées ($A_i = \emptyset$) : nous ne pouvons pas déterminer si l'action est active ou non et, dans le cas des relations procédurales, nous ne pouvons pas évaluer la relation ;
- Les actions trouvées peuvent être de nature différentes (processus actifs, réactions impossibles, *etc*) : nous ne pouvons pas répondre par une requête multiple donnant la nature de chaque action (ce n'est pas une réponse pertinente) ;
- Le type τ et le booléen ν de la requête peuvent ne pas correspondre aux actions trouvées (par exemple, le MRF recherche des processus actifs et ne trouve que des réactions) : nous devons donner une réponse adaptée, avec éventuellement des explications.

Pour simplifier le traitement de ces différentes situations et construire une réponse pertinente, nous proposons de construire plusieurs ensembles de termes VDL lors du traitement des requêtes, chacun correspondant à une classe particulière (par exemple, U pour les actions inconnues, P_{\top} pour les processus actifs, E pour les réactions, X pour les explications, *etc*). Nous allons montrer dans les sections suivantes comment ces ensembles peuvent être utilisés pour construire une réponse *pertinente* pour chaque acte et chaque type de requête.

4.3.1.4 Difficultés lors de la construction de la réponse

L'algorithme de recherche d'actions construit un ensemble de termes extraits directement de la vue et modélisant des actions. A_i contient donc un ensemble de termes munis d'attributs modélisant des actions VDL et des actions de *ground*. Par exemple :

```
stop[event[stop],put[path[running],false]]
increase[path[value],get[speed]]
etc
```

Pour construire des réponses pertinentes, nous devons tenir compte du fait qu'une action est une action VDL ou une action de *ground* pour déterminer de quelle façon elle doit être modélisée dans la requête. En effet, pour un utilisateur ordinaire, le nom d'une action VDL suffit à la désigner. En revanche, les actions de bon sens doivent être décrites intégralement.

Pour toute action a (et plus généralement, pour tout $a \in \Upsilon$), nous noterons \tilde{a} la modélisation de a dans la réponse, c'est-à-dire :

- $\tilde{a} = a$ si a est une action de *ground* ($\text{concept}(a) \in \mathcal{G}_p$) ;
- $\tilde{a} = \text{node}(\text{concept}(a))$ sinon.

De plus, il est important de savoir si une action dans l'objet ω d'une requête est une action de *ground*, pour déterminer l'acte de la réponse. En effet, si aucune action n'est trouvée dans la vue, le MRF peut répondre par une requête d'acte *Unknown* (« Je ne sais pas ce que x signifie ») lorsque tous les éléments de ω sont

des actions VDL. En revanche, s'il s'agit d'action de ground ou de relations, le MRF doit répondre par une requête d'acte *Assert* avec $\nu = \perp$ (« Je ne fais pas x »).

Enfin, pour permettre au MLN de construire une réponse « *Oui* » ou « *Non* » en utilisant la similitude entre la requête et la réponse construite par le MRF dans le cas de requêtes d'acte *Ask*, comme nous l'avons vu au chapitre précédent, il faut que l'objet d'une requête soit le plus proche possible de l'objet de la question. Nous montrerons dans les sections suivante comment cette contrainte peut être mis en œuvre dans les algorithmes de construction de réponse.

4.3.2 Requêtes d'acte $\alpha = What$

Les requêtes d'acte $\alpha = What$ sont utilisées pour modéliser les questions dans lesquelles l'utilisateur demande au système de rechercher la valeur d'une variable ou la définition d'un terme donnée, l'ensemble des actions du composant ou l'objet d'une action donnée, ou bien l'ensemble des interactions possibles.

Le sujet de la requête ne peut jamais être vide et il doit toujours référencer un seul terme de la vue :

- Si $\sigma = \emptyset$, le MRF retourne la requête :

$$\langle Error, is, \{ "subject empty" \}, \emptyset, \top, \surd \rangle$$

- Si $calcRef(\sigma, vue) = \emptyset$, le MRF retourne la requête :

$$\langle Unknown, is, \sigma, \emptyset, \top, \surd \rangle$$

- Si $calcRef(\sigma, vue) > 1$, le MRF retourne la requête :

$$\langle Error, is, \{ "subject ambiguous" \}, \emptyset, \top, \surd \rangle$$

Sinon, nous notons *sujet* le terme référencé : $\{sujet\} = calcRef(\sigma, vue)$.

4.3.2.1 Requêtes de type $\tau = is$

Les requêtes de la forme $(\alpha, \tau) = (What, is)$ modélisent des demandes de valeur ou de définition du terme *sujet*. Dans ces requêtes, nous aurons toujours :

- $\omega = \emptyset$; sinon, le MRF retourne la requête :

$$\langle Error, is, \{ "object not empty" \}, \emptyset, \top, \surd \rangle$$

- $\nu = \top$; sinon, le MRF retourne la requête :

$$\langle Error, is, \{ "negative request" \}, \emptyset, \top, \surd \rangle$$

C'est le MRF, lors de l'analyse du composant, qui détermine s'il doit donner une définition ou une valeur pour le terme *sujet*.

Notion de *définition* d'un terme

En VDL, chaque terme peut être munis d'une définition :

- *Implicite*, c'est-à-dire liée à la sémantique explicationnelle de son *concept* VDL ;
Par exemple, les concepts définis pour ϕ_{interf} ont tous une définition implicite permettant de matérialiser qu'un *button* est un bouton, qu'une *figure* est une figure, *etc.*

Nous notons $def_i(c) \in A^*$ la définition implicite associée à une concept c .

- *Explicite*, c'est-à-dire attachée par le concepteur lors de la définition du composant.

Nous notons $def_e(t) \in A^*$ la définition explicite d'un terme.

Dans une version intégrée de VDL, la définition explicite est définie par le programmeur dans le concept du terme. Mais dans notre modèle VDLp, les concepts étant réduits à de simples chaînes de caractères, nous représenterons la définition explicite d'un terme dans un attribut de concept *définition*. Par exemple, pour définir que la vue est « *Coco le compteur* », nous écrirons :

```
view[ definition["Coco the counter"], ... ]
```

La définition explicite d'un terme VDLp est alors obtenue en recherchant les termes de concept *définition* dans les attributs de t et en concaténant les représentations textuelles¹⁰ de leurs attributs :

$$def_e(t) = \bigodot_{t'' \in F_d(t)} f_{text}(t'')$$

où $F_d(t) = \bigcup \{t' \in fils(t) \text{ tq } concept(t') = definition\}$.

Réponse

La réponse à une requête d'acte *What* et de type *is* est alors :

$$\langle Assert, is, \sigma, R, \top, \surd \rangle$$

où σ et δ sont les critères de la requête traitée par le MRF et $R \in \mathcal{P}(\mathcal{T})$ est l'ensemble défini de la manière suivante :

- Si $def_e(sujet)$ existe, $R = \{node(def_e(sujet))\}$, c'est-à-dire un singleton contenant le terme atomique de concept $def_e(sujet)$;
- Sinon, si $def_i(concept(sujet))$ existe, $R = \{node(def_i(concept(sujet)))\}$;
- Sinon, $R = fils(sujet)$.

La définition explicite prime donc sur la définition implicite, qui prime sur la valeur du terme, donnée par ses attributs.

¹⁰La représentation textuelle d'un terme VDLp, $f_{text}(t)$, est définie section 2.3.3.9 page 61.

4.3.2.2 Requêtes de type $\tau = do$

Les requêtes de la forme $(\alpha, \tau) = (What, do)$ modélisent des demandes de recherche des actions ($\omega = \emptyset$) ou des termes modifiés par des actions précises ($\omega \neq \emptyset$).¹¹ Pour les recherches d'objets, le critère ν doit toujours être positif. Nous ne voulons pas en effet traiter et modéliser les questions comme : « *Qu'est-ce que tu ne comptes pas ?* ».

Recherche d'action : $\omega = \emptyset$

Lorsque $\nu = \top$, la recherche porte uniquement sur l'activité de la vue (c'est-à-dire sur les processus actifs). Au contraire, lorsque $\nu = \perp$, elle porte sur tout ce que le composant « n'est pas en train de faire », c'est-à-dire les processus inactifs.

Soit $A = \chi_{action}(sujet, \sqrt{\quad})$ l'ensemble des actions de *sujet*. Soit alors A_{\top} l'ensemble des noms des processus actifs dans *sujet* et A_{\perp} celui des noms des processus inactifs :

$$\begin{aligned} A_{\top} &= \left\{ \tilde{a}, a \in Atq \Pi_{\perp}^{(a)} = \mathcal{E}^{(a)} = \emptyset \right\} \\ A_{\perp} &= \left\{ \tilde{a}, a \in Atq \Pi_{\perp}^{(a)} \neq \emptyset \text{ et } \mathcal{E}^{(a)} = \emptyset \right\} \end{aligned}$$

La réponse à une requête d'acte *What*, de type *do* et d'objet $\omega = \emptyset$ est alors la suivante :

$$\langle Assert, do, \sigma, A_{\nu}, \nu, \sqrt{\quad} \rangle$$

où σ , ν et δ sont les critères de la requête traitée.

Notion d'objets d'action

Pour répondre aux requêtes lorsque $\omega \neq \emptyset$, le MRF doit rechercher les *objets* des actions ou des relations procédurales modélisées dans ω . Nous allons tout d'abord définir la notion d'objets d'action pour les différentes classes d'actions.

Pour toute terme t modélisant une action ou une relation procédurale, nous notons $\chi_{object}(t) \in \mathcal{P}(\Upsilon)$ l'ensemble des *objets* de t .

Soit $\chi_{concept} : \Upsilon \times A^* \longrightarrow \mathcal{P}(\Upsilon)$ la fonction qui recherche dans un terme t l'ensemble des sous-termes dont le concept est a :

$$\chi_{concept}(t, a) = \begin{cases} t & \text{si } concept(t) = a \\ \bigcup_{t' \in fils(t)} \chi_{concept}(t') & \text{sinon} \end{cases}$$

Si t est un processus, $\chi_{concept}(t, path)$ donne bien l'ensemble des *objets* de t , c'est-à-dire des termes modifiés par l'action t : $\chi_{object}(t) = \chi_{concept}(t, path)$. Mais si t est

¹¹En effet, comme nous l'avons vu au chapitre précédent, il existe deux actes *What* pour les requêtes portant sur le fonctionnement, que nous avons regroupé en un seul compte tenu de la structure de notre modèle de requêtes.

une réaction à un événement externe, nous ne pouvons plus appliquer cette règle puisque les termes modifiés peuvent dépendre de l'événement externe envoyé¹².

L'objet d'un terme t modélisant une action est alors défini de la manière suivante :

- Si $\exists t' \in \chi_{concept}(t, path) \text{ tq } \chi_{concept}(t', "event\ get") \neq \emptyset$ (certains termes dépendent de l'événement externe) :
 - S'il existe $e \in \mathcal{E}^{(a)}$ tel que $fil_s(e) \neq \emptyset$ (au moins un événement décrit ses attributs), alors $\chi_{object}(t) = fil_s(e)$;
 - Sinon :
 - Si $\forall t' \in \chi_{concept}(t, path), \chi_{concept}(t', "event\ get") \neq \emptyset$, alors $\chi_{object}(t) = \emptyset$;
 - Sinon, $\chi_{object}(t) = \chi_{concept}(t, path) \setminus \{t' \text{ tq } \chi_{concept}(t', "event\ get") \neq \emptyset\}$;
- Sinon $\chi_{object}(t) = \chi_{concept}(t, path)$.

Lorsque t est une relation procédurale ($concept(t) \in \mathcal{G}_{rp}$), l'ensemble des objets de t est alors défini par l'ensemble des objets des actions élémentaires, dans les arguments de t , qui vérifient la relation. Ils sont calculés de la manière suivante :

Soit $\{t_1, \dots, t_n\} = fil_s(t)$ et $R \in \mathcal{P}(\Upsilon^{n+1})$ tel que $(n, concept(t), R) \in \mathcal{G}_{rp}$.
Pour chaque $i \in [1, n]$, soit A_j l'ensemble des actions correspondant au terme t_i dans la vue et A'_j l'ensemble des actions élémentaires de ces actions :

$$A_j = \Xi_p^{(filtre)}(\chi_{action}(vue, \Xi_p(t_j, \top)), t_j)$$

$$A'_j = \bigcup_{a \in A_j, c \in \{add, put, del\}} \chi_{concept}(a, c)$$

Soit A l'ensemble des n -uplets $\{a_1, \dots, a_n\}$ tels que, $\forall j \in [1, n], a_j \in A'_j$ et $\Xi_{rp}(concept(t), n, \{a_1, \dots, a_n\}) = \top$ (les a_i vérifient la relation). L'objet de chaque t_j est défini par :

$$\chi_{object}(t_j) = \bigcup_{T \in A} \chi_{object}(a_j)$$

Si la relation est indéterminée, l'ensemble des objets n'est pas défini.

Recherche d'objet d'action : $\omega \neq \emptyset$

Pour les requêtes de type $\tau = do$, le critère ν doit toujours être positif. Si $\nu = \perp$, le MLN construit comme réponse :

$$\langle Error, is, \{ "negative\ request" \}, \emptyset, \top, \surd \rangle$$

Si $\nu = \top$, le MRF recherche dans la vue les objets de tous les processus actifs et, pour les relations, des relations qui portent uniquement sur des processus actifs. Pour construire la réponse, nous utilisons trois ensembles :

- P pour la description des objets des processus VDL trouvés dans la vue ou des relations vérifiés, portant sur des processus ;

¹²Voir par exemple le composant *Jojo*, présenté section 2.4.3.2 page 86.

- U pour les actions inconnues (qu'il s'agisse directement d'un ω_i ou d'un argument d'une relation modélisée par ω_i);
- N pour les réactions, les processus inactifs et les actions de *ground* qui n'existent pas dans la vue.

Pour tout $\omega_i \in \omega$:

- Si $\text{concept}(\omega_i) \notin \mathcal{G}_{rp}$:
 - Si $A_{\omega_i} = \emptyset$, alors $\omega_i \in N$ ssi $\text{concept}(\omega_i) \in \mathcal{G}_p$ et $\omega_i \in U$ sinon ;
 - Sinon, pour chaque $a \in A_{\omega_i}$, si $\Pi_{\perp}^{(a)} = \mathcal{E}^{(a)} = \emptyset$, soit $Obj = \chi_{object}(a)$. Si $Obj = \emptyset$, alors Obj est remplacé par $\{\text{path}[\text{nothing}]\}$. Alors :

$$\text{concept}(a)[Obj] \in P$$

Soulignons que $\forall a \in A_{\omega_i}, \text{concept}(a) = \text{concept}(\omega_i)$ par construction de A_{ω_i} par χ_{action} .

- Si $\text{concept}(\omega_i) \in \mathcal{G}_{rp}$:
 - Si $\exists t_j \in \text{fils}(\omega_i) \text{ tq } A_j = \emptyset$, alors $\omega_i \in N$ ssi $\text{concept}(\omega_i) \in \mathcal{G}_p$ et $\omega_i \in U$ sinon ;
 - Sinon, pour chaque $t_j \in \text{fils}(\omega_i)$, soit $B_j = \{a_j \in A_j \text{ tq } \Pi_{\perp}^{(a_j)} = \mathcal{E}^{(a_j)} = \emptyset\}$.
 - Si $\exists j \text{ tq } B_j = \emptyset$, alors $\text{concept}(t_j)[\text{path}[\text{nothing}]] \in N$;
 - Sinon, soit $\{a_{j,k}\}$ les actions élémentaires d'un $a_j \in B_j$ vérifiant la relation (c'est-à-dire telles que $\Xi_{rp}(\text{concept}(\omega_i), n, \{a_{1,k_1}, \dots, a_{n,k_n}\}) = \top$, comme nous l'avons expliqué plus haut).
Si la relation est indéterminée ($\{a_{j,k}\} = \emptyset$), $\omega_i \in U$. Sinon, nous notons $Obj_{j,k} = \chi_{object}(a_{j,k})$ les objets de chaque $a_{j,k}$ et $Obj_j = \bigcup_k Obj_{j,k}$ (ou $\{\text{path}[\text{nothing}]\}$ si $\bigcup_k Obj_{j,k} = \emptyset$). Alors pour chaque t_j :

$$\text{concept}(t_j)[Obj_j] \in P$$

La réponse à une requête d'acte *What*, de type $\tau = do$ et d'objet $\omega \neq \emptyset$ est alors :

- $\langle \text{Assert}, \tau, \sigma, P, \top, \sqrt{\ } \rangle$ si $P \neq \emptyset$ (« Je compte ma valeur ») ;
- $\langle \text{Assert}, \tau, \sigma, N, \top, \sqrt{\ } \rangle$ si $P = \emptyset$ et $N \neq \emptyset$ (« Je n'augmente rien ») ;
- $\langle \text{Unknown}, do, \sigma, U, \perp, \sqrt{\ } \rangle$ sinon (« Je ne sais pas ce que manger signifie ») ;

Cet algorithme permet de construire une réponse pertinente, c'est-à-dire contenant toutes les information nécessaires pour produire une réponse naturelle à l'utilisateur, et uniquement ceux-ci.

4.3.2.3 Requêtes de type $\tau = can$

Les requêtes de la forme $(\alpha, \tau) = (\text{What}, can)$ modélisent aussi des demandes de recherche des actions ($\omega = \emptyset$) ou des termes modifiés par des actions précises ($\omega \neq \emptyset$) dans le terme *sujet*, mais elles portent sur toutes les actions du composant (processus ou réactions), actives si $\nu = \top$, inactives sinon.

Ambiguïté de *can*

Nous sommes alors confronté au problème de l'ambiguïté du type *can*, utilisé pour représenter les questions dans lesquelles interviennent la notion de possibilité (cf. section 3.3.1.3 page 111). En effet, « pouvoir » faire quelque chose peut signifier :

- en avoir la possibilité physique : l'action est présente dans la vue ;
- en avoir la possibilité morale : les préconditions de l'action sont vérifiées.

Lorsque l'utilisateur demande « *Qu'est-ce que tu peux faire ?* », modélisé par la requête $\langle \textit{What}, \textit{can}, \{\textit{view}\}, \emptyset, \top, \surd \rangle$, nous ne pouvons pas déterminer s'il s'agit de possibilité physique ou morale.

Dans le MRF, nous mettrons en œuvre l'heuristique suivante :

- S'il existe des actions possibles moralement, la réponse décrit l'ensemble de ces actions ;
- Sinon, la réponse décrit l'ensemble des actions, même impossibles.

Par contre, pour les requêtes négatives ($\nu = \perp$), il n'y a plus d'ambiguïté possible : la question « *Qu'est-ce que tu ne peux pas faire ?* » ne peut pas être interprétée par le composant par « *Qu'est-ce que tu ne **sais** pas faire ?* ». La recherche porte donc uniquement sur les actions impossibles.

Notion d'action *possible*

Nous dirons qu'une action est *possible* si toutes ses préconditions sont vérifiées.

Si $a \in \Upsilon$ est un processus ou une réaction dont aucune garde ne dépend de l'événement externe traité, alors a est possible si et seulement si $\Pi_{\perp}^{(a)} = \emptyset$. En revanche, si a est une réaction dont certaines gardes dépendent de l'événement traité, nous considérerons que a est possible si et seulement si $\Pi_{\perp}^{(a)}$ contient uniquement ce type de gardes :

$$\forall t \in \Pi_{\perp}^{(a)}, \chi_{\textit{concept}}(t, \textit{event get}) \neq \emptyset$$

Pour toute action a , nous noterons $\textit{poss}(a)$ si a est *possible* :

$$\textit{poss}(a) = (\mathcal{E}^{(a)} = \emptyset) \vee \left(\forall t \in \Pi_{\perp}^{(a)}, \chi_{\textit{concept}}(t, \textit{event get}) \neq \emptyset \right)$$

Notion de réaction possible

Nous dirons qu'une réaction est *possible* si tous ses arguments existent dans la vue et s'ils peuvent s'apparier avec l'ensemble caractéristique R de la relation, indépendamment du fait que la relation de contrainte r soit vérifiée ou non, c'est-à-dire si elle n'est pas indéterminée.

Si $t \in \omega_i$ modélise une relation, $\{t_1, \dots, t_n\} = \textit{fils}(t)$ et $A = \{A_{t_i}\}$, alors :

$$\textit{poss}(t) = (f_{rp}^*(c_s, n, A, \emptyset) \in \{\top, \perp\})$$

NB : Pour les requêtes d'acte *What*, nous n'aurons pas de relations d'arité 0, puisqu'il s'agit de rechercher les objets de actions intervenant dans ces relations.

Cependant, nous verrons dans la section suivante que la notion de possibilité d'une relation d'arité 0 soulève de nouvelles difficultés.

Recherche d'action : $\omega = \emptyset$

Pour répondre aux requêtes d'acte *What*, de type *can* et d'objet $\omega = \emptyset$, le MRF recherche les actions dans la vue. Lorsque $\nu = \top$, la recherche porte sur tout ce que le composant « peut » faire, c'est-à-dire sur les actions possibles ou, s'il n'y en a pas, sur toutes les actions du composant. Lorsque $\nu = \perp$, la recherche porte sur les actions impossibles.

Soit $A = \chi_{action}(sujet, \sqrt{ })$, $A_{\top} = \{\tilde{a}, a \in Atq\,poss(a)\}$ l'ensemble des noms des actions possibles et $A_{\perp} = A \setminus A_{top}$ celui des actions impossibles. La réponse à une requête d'acte *What*, de type *can* et d'objet $\omega = \emptyset$ est alors :

- $\langle Assert, can, \sigma, A_{\nu}, \nu, \sqrt{ } \rangle$ si $A_{\nu} \neq \emptyset$ (possibilité ou impossibilité morale) ;
- $\langle Assert, can, \sigma, A, \top, \sqrt{ } \rangle$ si $\nu = \top$, $A_{\top} = \emptyset$ et $A_{\perp} \neq \emptyset$ (possibilité physique) ;
- $\langle Assert, can, \sigma, \{nothing\}, \top, \sqrt{ } \rangle$ si $\nu = \top$ et $A_{\top} = A_{\perp} = \emptyset$ (« Je ne sais rien faire ») ;
- $\langle Assert, can, \sigma, \{everything\}, \top, \sqrt{ } \rangle$ si $\nu = \perp$ et $A_{\perp} = \emptyset$ (« Je peux tout faire ») ;

Soulignons que, pour construire une réponse pertinente, nous ne travaillons pas de manière symétrique par rapport à la valeur de ν .

Recherche d'objet d'action : $\omega \neq \emptyset$

La réponse à une requête d'acte *What*, de type $\tau = can$ et d'objet $\omega \neq \emptyset$ est construite par un algorithme similaire à celui proposé pour $\tau = do$, dans lequel :

- La contrainte $\Pi_{\perp}^{(a)} = \mathcal{E}^{(a)} = \emptyset$ (processus actif) est remplacée par $poss(a) = \nu$ dans les actions et les réactions ;
- Les $\{a_{j,k}\}$ ne vérifient pas nécessairement la relation : il suffit qu'elle soit déterminée : $\Xi_{rp}(concept(\omega_i), n, \{a_{1,k_1}, \dots, a_{n,k_n}\}) \in \{\top, \perp\}$.

De cette façon, l'algorithme recherche les actions et les relations possibles ou impossibles, selon ν , et construit une réponse pertinente donnant leurs objets, en tenant compte des problèmes de références dans ω et de l'ambiguïté de *can*.

4.3.2.4 Requêtes de type $\tau = order$

Les requêtes de la forme $(\alpha, \tau) = (What, order)$ modélisent des demandes de recherche des interactions possibles. Le MRF doit donc déterminer dans la vue quels sont les événements externes traités par une action de la vue, c'est à dire l'ensemble des $\mathcal{E}^{(a)}$, $a \in \chi_{action}(vue, \sqrt{ })$.

Comme pour les requêtes $(What, is)$, nous aurons toujours $\omega = \emptyset$ et $\nu = \top$ (sinon, le MRF retourne les mêmes erreurs que dans le cas $(What, is)$). La réponse

à une requête d'acte *What* et de type *order* est alors :

$$\left\langle \text{Order}, \text{order}, \bigcup_{a \in \mathcal{X}_{\text{action}}(\text{vue}, \nu)} \mathcal{E}^{(a)}, \emptyset, \top, \sqrt{} \right\rangle$$

4.3.3 Requêtes d'acte $\alpha = Ask$

Les requêtes d'acte $\alpha = Ask$ sont utilisées pour modéliser les questions dans lesquelles l'utilisateur demande au système de vérifier l'état, l'activité, le comportement ou les interactions du composant. Elles peuvent faire intervenir des relations de bon sens sur la structure ou sur le fonctionnement.

Dans le cas général, le sujet de la requête ne peut jamais être vide et il doit toujours référencer un seul terme de la vue, sinon le MRF retourne une erreur (comme pour $\alpha = What$). Nous notons alors *sujet* le terme référencé. Cependant, pour $\tau = is$, nous pouvons avoir $\sigma = \emptyset$ lorsque l'objet ω contient uniquement des relations de bon sens.

Enfin, l'objet ω d'une requête d'acte $\alpha = Ask$ ne peut jamais être vide (puisque c'est lui qui donne l'objet du contrôle). Lorsque $\omega = \emptyset$, le MRF retourne la requête :

$$\langle \text{Error}, is, \{ \text{"object empty"} \}, \emptyset, \top, \sqrt{} \rangle$$

4.3.3.1 Requêtes de type $\tau = is$

Les requêtes de la forme $(\alpha, \tau) = (Ask, is)$ modélisent des demandes de vérification structurelles du composant.

Demande de valeur : $\sigma \neq \emptyset$

Lorsque $\sigma \neq \emptyset$, les requêtes (Ask, is) modélisent des demandes de vérification de valeurs de variables. Si $\nu = \top$, le MRF doit vérifier que la variable *sujet* a bien pour valeur l'un des termes de l'objet ω . Si $\nu = \perp$, il doit vérifier que la valeur de *sujet* n'est pas dans l'objet.

La réponse à une requête d'acte *Ask* et de type *is* lorsque $\sigma \neq \emptyset$ est donc :

- $\langle \text{Assert}, is, \sigma, \text{fils}(\text{sujet}), \top, \sqrt{} \rangle$ si $\text{fils}(\text{sujet}) \subseteq \omega$ (« *Ma vitesse est 42* ») ;
- $\langle \text{Assert}, is, \sigma, \omega, \perp, \sqrt{} \rangle$ si $\text{fils}(\text{sujet}) \not\subseteq \omega$ et $\nu = \perp$ (« *Ma vitesse n'est pas 37* ») ;
- $\{ \langle \text{Assert}, is, \sigma, \omega, \perp, \sqrt{} \rangle, \langle \text{Assert}, is, \sigma, \text{fils}(\text{sujet}), \top, \sqrt{} \rangle \}$ si $\text{fils}(\text{sujet}) \not\subseteq \omega$ et $\nu = \top$ (« *Ma vitesse n'est pas 32 : c'est 47* »).

Remarquons que, pour construire une réponse naturelle, l'algorithme de construction de réponse n'est pas symétrique par rapport ν : nous préférons les réponses positives aux réponses négatives.

Relations : $\sigma = \emptyset$

Les requêtes d'acte *Ask* et de type *is* ayant un sujet $\sigma = \emptyset$ sont utilisées pour modéliser des relations statiques ou procédurales. Le MRF doit calculer la valeur de vérité de chaque relation et la comparer avec le critère ν de la requête. Nous construisons alors trois ensembles :

- R_{\top} pour les relations vérifiées;
- R_{\perp} pour les relations non vérifiées.
- U pour les actions inconnues apparaissant dans une relation et les relations indéterminées;

Nous supposons qu'un même concept de relation de bon sens ne peut pas être utilisé dans \mathcal{G}_{rs} et \mathcal{G}_{rp} . Alors, pour tout $i \in [1, |\omega|]$, la valeur de vérité v_i de la relation statique modélisée par ω_i est définie par :

- Si $\text{concept}(\omega_i) \in \mathcal{G}_{rs}$,

$$v_i = (\zeta(\Xi_{rs}(\omega_i, \top), \emptyset) = \{true\})$$

- Si $\text{concept}(\omega_i) \in \mathcal{G}_{rp}$, soit $\{t_1, \dots, t_n\} = \text{fils}(\omega_i)$ et $R \in \mathcal{P}(\Upsilon^{n+1})$ tel que $(n, \text{concept}(\omega_i), R) \in \mathcal{G}_{rp}$.
 - Si $\omega_i = \text{nothing}$, $v_i = (\chi_{action}(vue, \sqrt{\quad}) = \emptyset)$;
 - Si $\omega_i = \text{something}$, $v_i = (\chi_{action}(vue, \sqrt{\quad}) \neq \emptyset)$;
 - Si $\omega_i = \text{nothing}$, $v_i = \perp$;
 - Sinon, pour tout $j \in [1, n]$, soit A_j l'ensemble des actions correspondant au terme t_j dans la vue et A'_j l'ensemble des actions élémentaires de ces actions (cf. section (*What, do*)). S'il existe $j \in [1, n]$ tel que $A_j = \emptyset$, alors $\tilde{t}_j \in U$; sinon :

$$v_i = \Xi_{rp}^*(\text{concept}(\omega_i), n, \{A'_j\}_{j \in [1, n]})$$

Si la relation est indéterminée, alors $\omega_i \in U$.

Alors $\omega_i \in R_{v_i}$.

Réponse

La réponse à une requête d'acte *Ask* et de type *is* lorsque $\sigma = \emptyset$ est alors :

- $\langle \text{Assert}, is, \sigma, R_{\nu}, \nu, \sqrt{\quad} \rangle$ si $R_{\nu} \neq \emptyset$ (« *Oui, ma valeur est positive* »);
- $\langle \text{Assert}, is, \sigma, R_{\neg\nu}, \neg\nu, \sqrt{\quad} \rangle$ si $R_{\nu} = \emptyset$ et $R_{\neg\nu} \neq \emptyset$ (« *Non, ma valeur n'est pas positive* »);
- $\langle \text{Unknown}, is, \sigma, U, \top, \sqrt{\quad} \rangle$ si $R_{\nu} = R_{\neg\nu} = \emptyset$ (« *Je ne sais pas ce que "start est le copain de stop" signifie* »).

4.3.3.2 Requêtes de type $\tau = do$

Les requêtes de la forme $(\alpha, \tau) = (\text{Ask}, do)$ modélisent des demandes de vérification portant sur l'activité du composant.

Cependant, l'utilisateur et le MLN ne connaissent pas la nature des actions. Par conséquent, l'objet ω de la requête peut contenir des terme faisant référence

indifféremment à des processus ou à des réactions. Pour construire une réponse pertinente, le MRF privilégie alors les processus actifs sur les autres actions, et utilise une date $\delta = past$ lorsque les actions retournées sont des réactions.

Les actions de la vue sont alors catégorisées de la manière suivante :

actions ::= ce que fait | ce que ne fait pas
ce que fait ::= *processus actifs*
ce que ne fait pas ::= *processus inactifs* | *réactions*

De plus, l'objet ω peut contenir des relations procédurales portant sur l'activité ou le comportement (par exemple : « *Est-ce que tu comptes à l'envers ?* »). Le MRF doit alors vérifier la relation, indépendamment de la nature des actions.

Pour construire une réponse pertinente, nous utilisons quatre ensembles :

- P_{\top} pour les processus actifs et les relations vérifiées ;
- P_{\perp} pour les processus inactifs et les relations non vérifiées ;
- E pour les réactions ;
- U pour les actions inconnues ou les relations indéterminées.

Notion de conséquence fixe

Soit $a \in \Upsilon$ une action. a est un processus si et seulement si $\mathcal{E}^{(a)} = \emptyset$. Dans ce cas, a est active si et seulement si $\Pi_{\perp} = \emptyset$. En revanche, si a est une réaction ($\mathcal{E}^{(a)} \neq \emptyset$), l'analyse des préconditions de a permet seulement de dire si a pourrait être effectuée si l'événement était envoyé.

Pour répondre à une requête de la forme (*Ask, do*) portant sur une réaction a , le MRF doit déterminer si le composant est dans un état résultat de l'action a , comme le montrent les exemples donnés au chapitre précédent. Nous proposons pour cela d'utiliser les *conséquences fixes* de a pour construire la réponse¹³ :

Nous appelons *conséquences fixes* les résultats d'une action élémentaire ne dépendant pas de l'état courant.

Conséquence fixe d'une action élémentaire

Soit a une actions élémentaire et R_a l'ensemble de ses attributs de concept *path* ne contenant pas de sous-termes utilisant les références :

$$R_a = \{t \in \text{fils}(a) \text{ tq } \text{concept}(t) = \text{path} \text{ et } \bigcup_{c \in \text{Ref}} \chi_{\text{concept}}(t, c) = \emptyset\}$$

où $\text{Ref} = \{get, \text{ "event get"}, \text{ "size ref"}\}$. De plus, si $\text{concept}(a) \in \{add, put\}$, soit A l'ensemble des attributs autres que les *path* et ne contenant pas de sous-termes utilisant les références :

$$A_a = \{t \in \text{fils}(a) \text{ tq } \text{concept}(t) \neq \text{path} \text{ et } \bigcup_{c \in \text{Ref}} \chi_{\text{concept}}(t, c) = \emptyset\}$$

¹³En effet, il est impossible de vérifier les conséquences *non fixes* d'une action.

Pour chaque $r \in R_a$, les conséquences fixes de a associées à r sont vérifiées si et seulement si :

- Lorsque $\text{concept}(a) = \text{del}$, $\text{calcRef}(\text{fils}(r), \text{vue}) = \emptyset$;
Sinon, l'ensemble des conséquences fixes non vérifiées est modélisé par la requête $\rho_a = \langle \text{Unknown}, \text{is}, \sigma \cup \text{fils}(r), \emptyset, \perp, \surd \rangle$ (le terme n'est pas inconnu);
- Lorsque $\text{concept}(a) = \text{add}$, $\forall t \in \text{calcRef}(\text{fils}(r), \text{vue}), A_a \subseteq \text{fils}(t)$;
Sinon, l'ensemble des conséquences fixes non vérifiées est modélisé par la requête $\rho_a = \langle \text{Assert}, \text{is}, \sigma \cup \text{fils}(r), A_a \setminus \text{fils}(t), \perp, \surd \rangle$ (le terme n'a pas x dans ses attributs), où t est l'un des termes tels que $A_a \not\subseteq \text{fils}(t)$;
- Lorsque $\text{concept}(a) = \text{put}$, $\forall t \in \text{calcRef}(\text{fils}(r), \text{vue}), A_a = \text{fils}(t)$;
Sinon, si $\text{fils}(t) \subset A_a$, l'ensemble des conséquences fixes non vérifiées est modélisé par la requête $\rho_a = \langle \text{Assert}, \text{is}, \sigma \cup \text{fils}(r), \text{fils}(t), \top, \surd \rangle$ (le terme vaut x), où t est l'un des termes tels que $A_a \neq \text{fils}(t)$.

Si $a \in \Upsilon$ est une action VDL, nous notons $ok_{csq}(a) = \top$ si et seulement si, pour tout $a' \in \bigcup_{c \in \{\text{add}, \text{put}, \text{del}\}} \chi_{\text{concept}}(a, c)$, l'ensemble des conséquences fixes de a' associées à chaque $r \in R_{a'}$ sont vérifiées. Sinon, nous notons $ko_{csq}(a) = \bigcup_{a'} \rho_{a'}$ l'ensemble des requêtes modélisant des conséquences fixes non vérifiées.

Actions

- Si $\omega_i \in \omega$ modélise une action (c'est-à-dire si $\text{concept}(\omega_i) \notin \mathcal{G}_{rp}$) :
- Si $A_{\omega_i} = \emptyset$, alors ω_i est mémorisé dans U ;
 - Sinon, pour chaque $a \in A_{\omega_i}$, $\tilde{a} \in E$ si $\mathcal{E}^{(a)} \neq \emptyset$, $\tilde{a} \in P_{\top}$ si $\Pi_{\perp}^{(a)} = \mathcal{E}^{(a)} = \emptyset$ et $\tilde{a} \in P_{\perp}$ sinon.

Relations procédurales

Lorsque $\omega_i \in \omega$ est une relation procédurale, le MRF doit évaluer la valeur de vérité de la relation pour construire la réponse. Soit $\{t_1, \dots, t_n\} = \text{fils}(\omega_i)$ et $\forall j \in [1, n]$ soient A_j et A'_j les ensembles usuels. La valeur de vérité v_i de la relation procédurale modélisée par ω_i est alors calculée de la manière suivante :

- Si $\omega_i = \text{nothing}$, $v_i = \top$ ssi $\forall a \in \chi_{\text{action}}(\text{vue}, \surd), (\Pi_{\perp}^{(a)} \neq \emptyset \vee \mathcal{E}^{(a)} \neq \emptyset)$;
- Si $\omega_i = \text{something}$, $v_i = \top$ ssi $\exists a \in \chi_{\text{action}}(\text{vue}, \surd), \Pi_{\perp}^{(a)} = \mathcal{E}^{(a)} = \emptyset$;
- Si $\omega_i = \text{something}$, $v_i = \top$ ssi $\forall a \in \chi_{\text{action}}(\text{vue}, \surd), \Pi_{\perp}^{(a)} = \mathcal{E}^{(a)} = \emptyset$;
- Sinon, pour tout $j \in [1, n]$, soit A_j et A'_j les ensembles d'actions usuels (cf. section (*What, do*)). S'il existe $j \in [1, n]$ tel que $A_j = \emptyset$, alors $\tilde{t}_j \in U$; Si la relation est indéterminée, alors $\omega_i \in U$. Sinon :

$$v_i = \begin{cases} \Xi_{rp}^*(\text{concept}(\omega_i), n, \{A'_j\}_{j \in [1, n]}) \\ \wedge (\forall a \in \bigcup_{j \in [1, n]} A_j, \Pi_{\perp}^{(a)} = \mathcal{E}^{(a)} = \emptyset) \end{cases}$$

C'est-à-dire que la relation est vérifiée si et seulement si toutes les actions sont des processus actifs et si la contrainte est vérifiée.

L'élément ω_i est alors mémorisé dans P_{v_i} .

Réponse

La réponse à une requête d'acte *Ask* et de type *do* est alors construite en utilisant les quatre ensembles d'éléments construits par analyse des $\omega_i \in \omega$:

- Si $P_\nu \neq \emptyset$, le MRF retourne $\langle \text{Assert}, do, \sigma, P_\nu, \nu, \surd \rangle$ (« *Oui, je compte* »);
- Sinon, si $P_{\neg\nu} \neq \emptyset$, le MRF retourne $\langle \text{Assert}, do, \sigma, P_{\neg\nu}, \neg\nu, \surd \rangle$ (« *Si, je compte* »);
- Sinon, si $E = \emptyset$, le MRF retourne $\langle \text{Unknown}, do, \sigma, U, \top, \surd \rangle$ (« *Je ne sais pas ce que compter signifie* »);
- Sinon, si $\forall a \in E, ok_{csq}(a) = \top$, le MRF retourne $\langle \text{Assert}, do, \sigma, E, \top, past \rangle$ (« *Je suis arrêté* »);
- Sinon, si $\nu = \perp$, le MRF retourne $\langle \text{Assert}, do, \sigma, E, \perp, past \rangle$ (« *Je ne suis pas arrêté* »);
- Sinon, il faut construire une explication décrivant les conséquences fixes non vérifiées : soit $R = \bigcup_{a \in E \text{ tq } ok_{csq}(a) \neq \top} ko_{csq}(a)$. La réponse est alors la requête multiple : $\{\langle \text{Assert}, do, \sigma, E, \perp, past \rangle\} \cup R$.

Par exemple, dans *Coco*, si l'utilisateur demande « *Tu es arrêté, non ?* » alors que *running* vaut *true*, le MRF construira la réponse :

$$\{\langle \text{Assert}, do, \{\text{view}\}, \{\text{stop}\}, \perp, past \rangle, \langle \text{Assert}, is, \{\text{running}\}, \{\text{true}\}, \top, \surd \rangle\}$$

ce qui produira la réponse : « *Non, parce que je suis en marche.* ».

Cet algorithme, un peu complexe, permet de construire une réponse pertinente pour un dialogue naturel, dans lequel les actions inconnues sont ignorées et l'activité prime sur le comportement, conformément à la sémantique des requêtes de type $\tau = do$.

4.3.3.3 Requêtes de type $\tau = can$

Les requêtes de la forme $(\alpha, \tau) = (Ask, can)$ modélisent des demandes de vérification portant sur les actions, relation ou interactions « possibles » du composant. En effet, l'objet ω d'une requête (Ask, can) peut contenir :

- Des actions (processus ou réactions) : le MRF doit alors vérifier leurs préconditions booléennes pour déterminer si elles sont possibles;
- Des événements externes : le MRF doit alors vérifier si ces événements sont traitables par une réaction possible de la vue;
- Des relations procédurales : le MRF doit alors vérifier si ces relations sont possibles.

Comme pour les requêtes d'acte *What*, nous sommes alors confrontés au problème de l'ambiguïté des requêtes de type $\tau = can$, liée à la notion de *possibilité*. Pour lever cette ambiguïté, nous proposons :

- Comme dans le cas $\alpha = What$, de considérer d'abord les actions possibles et, en cas d'échec, d'étendre aux actions impossibles avant de considérer qu'une action est inconnue;
- De donner des *explications* dans les cas d'impossibilité morale. En effet, toute action impossible physique l'est parce que ses gardes ne sont pas vérifiées. Nous considérerons que ces gardes décrivent des explications de l'impossibilité.

Pour construire une réponse à une requête d'acte *Ask* et de type *can*, nous utilisons alors quatre ensembles :

- A_{\top} pour les éléments (actions, relations ou événements) possibles ;
- A_{\perp} pour les éléments impossibles ;
- E pour les explications pour les éléments impossibles ;
- U pour les éléments de ω inconnus dans la vue.

Enfin, dans les réponses, nous privilégions les éléments possibles sur les éléments impossibles ou inconnus.

Relations procédurales d'arité 0

Pour les relations d'arité 0, nous ne pouvons pas interpréter « pouvoir » par « savoir » (possibilité physique). En effet, les question comme : « *Est-ce que tu sais tout faire* », « *Est-ce que tu ne sais rien faire* » ou « *Est-ce que tu sais faire quelque chose* » peuvent être traitées en amont, par le MLN, qui connaît les actions dans le composant. Nous traiterons donc ces relations dans les requêtes *can* en utilisant la possibilité physique et en construisant des explications pour lever l'ambiguïté dans la question.

De plus, notre modèle de requête ne permet pas de différencier deux questions de possibilités : « *Est-ce que tu ne peux rien faire ?* » et « *Est-ce que tu peux ne rien faire ?* ». Toutes les deux sont modélisées par la requête :

$$\langle Ask, can, \{view\}, \{nothing\}, \top, \surd \rangle$$

Nous pouvons cependant contourner le problème en utilisant le fait que « *ne rien pouvoir faire* » est le contraire de « *pouvoir faire quelque chose* ». Ainsi, dans notre modèle de requêtes :

- « *Est-ce que tu peux ne rien faire ?* » sera modélisé par :

$$\langle Ask, can, \{view\}, \{nothing\}, \top, \surd \rangle$$

- « *Est-ce que tu ne peux rien faire ?* » sera modélisé par :

$$\langle Ask, can, \{view\}, \{something\}, \perp, \surd \rangle$$

Le MLN doit connaître cette particularité

La valeur de possibilité d'une relation procédurale t d'arité 0 est alors définie par :

- Si $t = nothing$, $poss(t) = \top$ ssi $\forall a \in \chi_{action}(vue, \surd)$, $(\Pi_{\top}^{(a)} \neq \{true\} \vee \mathcal{E}^{(a)} = \emptyset)$;¹⁴
- Si $t = something$, $poss(t) = \top$ ssi $\exists a \in \chi_{action}(vue, \surd)$, $poss(a)$;
- Si $\omega_i = everything$, $\forall a \in \chi_{action}(vue, \surd)$, $poss(a)$.

¹⁴En effet, $\Pi_{\top}^{(a)} = \{true\}$ et $\mathcal{E}^{(a)} = \emptyset$ si et seulement si l'action est un processus non gardé, qui est le seul cas d'actions que la vue ne peut pas s'empêcher de faire.

Relations

Si $\omega_i \in \omega$ est une relation procédurales et $\{t_1, \dots, t_n\}$ ses arguments :

- S'il existe $j \in [1, n]$ tel que $A_{t_j} = \emptyset$, alors $t_j \in U$;
- Sinon, $\omega_i \in A_{\top}$ si $poss(\omega_i)$ et $\omega_i \in A_{\perp}$ sinon.

De plus, pour $\omega_i \in \{something, everything\}$, l'ensemble E des explications utilisé dans l'algorithme de construction de réponse contient l'ensemble des gardes des actions $a \in \chi_{action}(vue, \sqrt{\quad})$ ne contenant pas "event get" (cf. la définition de la notion de possibilité pour les actions, page 172) :

$$E \supseteq \bigcup_{a \in \chi_{action}(vue, \sqrt{\quad})} \left\{ t \in \Pi_{\perp}^{(a)} tq \chi_{concept}(t, "event get") = \emptyset \right\}$$

Actions

Si ω_i est une action, alors :

- Si $A_{\omega_i} = \emptyset$, $\omega_i \in U$;
- Sinon, pour chaque $a \in A_i$, $\tilde{a} \in A_{\top}$ si $poss(a)$ et $\tilde{a} \in A_{\perp}$ sinon.

L'ensemble des explication E contient alors l'ensemble des gardes rendant a impossible :

$$E \supseteq \left\{ t \in \Pi_{\perp}^{(a)} tq \chi_{concept}(t, "event get") = \emptyset \right\}$$

Événements

Lorsque ω_i est un événement externe, le MRF doit rechercher les réactions qui traitent cet événement et déterminer si elles sont possibles, en tenant compte cette fois du fait que l'événement est envoyé à la vue.

Une première approche consiste à rechercher les préconditions de toutes les actions, vérifier que l'événement ω_i en fait partie et calculer les gardes de ces actions. Mais nous pouvons aussi remarquer que toute réaction, définie par une précondition de concept *event*, sera nécessairement englobée dans un concept (le *nom* de l'action). Pour produire une réponse pertinente, il suffit alors de vérifier les préconditions de cette action VDL, que nous appellerons *réaction principale* à ω_i , et non de prendre en compte toutes ses sous-actions.

Notion de *réaction principale* à un événement

Si $e \in \Upsilon$ est un événement externe nous appelons *réactions principales* à e dans une vue v les actions de v dont un attribut est un terme de concept *event* ayant $e' \succeq e$ dans ses attributs.

Nous notons $\chi_{evt} : \Upsilon \times \Upsilon \longrightarrow \mathcal{P}(\Upsilon)$ la fonction récursive qui recherche dans un terme t toutes les réactions principales d'un événement e :

$$\chi_{evt}(t, e) = \begin{cases} t & \text{si } \exists t' \in \text{fils}(t) tq \text{ concept}(t') = \text{event} \\ & \text{et } \exists e' \in \text{fils}(t') tq e' \succeq e \\ \bigcup_{t' \in \text{fils}(t)} \chi_{evt}(t', e) & \text{sinon} \end{cases}$$

Si $\omega_i \in \omega$ est un événement traité par la vue (c'est-à-dire $\chi_{evt}(vue, \omega_i) \neq \emptyset$). Alors :

- S'il existe une réaction principale à ω_i dont les gardes sont vérifiées, $\omega_i \in A_{\top}$:

$$\exists a \in \chi_{evt}(vue, \omega_i), e \succeq \omega_i \text{ et } \Pi_{\top} \neq \emptyset \text{ tq } \chi_{precond}(a, \{\omega_i\}) = (\Pi_{\top}, \emptyset, \{e\})$$

- Sinon, $\omega_i \in A_{\perp}$ et l'ensemble d'explications E contient les gardes non vérifiées :

$$\bigcup_{a \in \chi_{evt}(vue, \omega_i)} \Pi_{\perp}^{(a,e)} \subseteq E$$

où, pour chaque $a \in \chi_{evt}(vue, \omega_i)$, $\Pi_{\perp}^{(a,e)}$ est le second élément de $\chi_{precond}(a, \{e\})$.

Réponse

La réponse à une requête d'acte *Ask* et de type *can* est alors définie de la manière suivante :

- Si $A_{\nu} = A_{\neg\nu} = \emptyset$, le MRF retourne $\langle Unknown, do, \sigma, U, \top, \surd \rangle$ (« *Je ne sais pas ce que compter signifie* »);
- Si ($\nu = \top$ et $A_{\top} \neq \emptyset$) ou ($\nu = \perp$ et $A_{\perp} = \emptyset$), le MRF retourne la requête $\langle Assert, can, \sigma, A_{\nu}, \nu, \surd \rangle$ (« *Je peux compter* »);
- Sinon, le MRF retourne une requête multiple $\{\rho_i\}_{i \in [1, |E|+1]}$ telle que :

$$\begin{aligned} \rho_1 &= \langle Assert, can, \sigma, A_{\neg\nu}, \neg\nu, \surd \rangle \\ \rho_{i \geq 2} &= \langle Assert, is, \emptyset, fils(E_i), \perp, \surd \rangle \end{aligned}$$

Chaque $\rho_{i \geq 2}$ exprime l'invalidité d'une précondition (par exemple, pour modéliser : « *Je ne peux pas compter : je ne suis pas en marche* »).

Les réponses positives ($\nu = \top$) sont donc prioritaires sur les réponses négatives, afin de produire une réponse pertinente.

4.3.3.4 Requêtes de type $\tau = order$

Les requêtes de la forme $(\alpha, \tau) = (Ask, order)$ modélisent des demandes de vérification portant sur les interactions passées du composant. Dans ces requêtes, nous aurons toujours $\nu = \top$, sinon le MRF retourne une erreur « *negative request* ».

Pour construire une réponse à une requête $(Ask, order)$, le MRF doit rechercher dans Θ si un événement parmi ceux proposés dans l'objet ω a été envoyé à la vue au cours de l'exécution passée (cf. page 161) : soit C l'ensemble de ces événements :

$$C = \{(t, d) \in \Theta \text{ tq } \exists \omega_i \in \omega \text{ tq } t = \omega_i\}$$

Si $C = \emptyset$, la réponse donnée par le MRF est :

$$\langle Assert, order, \sigma, \omega, \perp, past \rangle$$

Sinon, soit $d_0 = \max\{d \text{ tq } \exists t \in \Upsilon \text{ tq } (t, d) \in C\}$ la date la plus récente d'envoi d'un de ces événements, $T_0 = \{t \text{ tq } (d_0, t) \in \Theta\}$ l'ensemble des événements qui lui sont associés et $d'_0 = (d_0 - \theta)$ sa valeur (négative) relativement à la date courante. La réponse donnée par le MRF est alors :

$$\langle Assert, order, \sigma, T_0, \top, date[d'_0] \rangle$$

4.3.4 Requêtes d'acte $\alpha = How$

Les requêtes d'acte $\alpha = How$ sont utilisées pour modéliser les questions dans lesquelles l'utilisateur demande au système comment une action fonctionne ($\tau = do$) ou comment faire pour atteindre un état but ($\tau = is$) ou pouvoir exécuter une action ($\tau = can$). Ces deux derniers cas correspondent à des demandes de *plans*.

Le sujet σ de la requête ne peut être vide que pour les requêtes de type $\tau = is$ modélisant des relations. Sinon, il doit toujours référencer un seul terme de la vue (sinon le MRF retourne une erreur); nous notons alors *sujet* le terme référencé. L'objet ω de la requête ne peut pas non plus être vide (puisqu'il décrit l'état à atteindre ou l'action à expliquer).

Enfin, rappelons que les requêtes d'acte *How* ont toutes une date $\delta = \surd$. Sinon, le MRF retourne l'erreur :

$$\langle Error, is, \{ "malformed request" \}, \emptyset, \top, \surd \rangle$$

Nous avons choisi de présenter tout d'abord l'analyse du corps d'une action ($\tau = do$), afin de regrouper les deux parties correspondant à des demandes de plan ($\tau = is$ et $\tau = can$).

4.3.4.1 Requêtes de type $\tau = do$

Les requêtes de la forme $(\alpha, \tau) = (How, do)$ modélisent des demandes d'explications d'actions. Le MRF doit rechercher dans la vue les actions correspondant aux éléments de ω et donner leur corps représenté de manière naturelle, c'est-à-dire en utilisant des actions de bon sens.

Si ω_i est une relation, le MRF doit expliquer les actions intervenant dans cette relation, comme pour le cas *(What, do)* vu précédemment. En effet, la valeur de vérité de la relation procédurale est conditionnée par le corps des actions intervenant dans la relation. Par exemple, pour répondre à la question « *Comment fais-tu pour compter à l'envers ?* », il suffit de répondre à « *Comment fais-tu pour compter ?* ».

Enfin, nous n'accepterons pas les requêtes négatives pour les demandes d'explications d'actions. Si $\nu = \perp$, le MRF retourne une erreur « *negative request* ».

Réponse

Pour chaque $\omega_i \in \omega$:

- Si $concept(\omega_i) \in \mathcal{G}_{rp}$ et $\{t_1, \dots, t_n\} = fils(\omega_i)$, alors $A_i = \bigcup_{j \in [1, n]} A_{t_j}$;
- Sinon, $A_i = A_{\omega_i}$.

Soit alors $A = \bigcup_{i \in [1, |\omega|]} A_i$. Si $A = \emptyset$, le MRF retourne une requête :

$$\langle Unknown, do, \sigma, \omega, \top, \surd \rangle$$

Sinon, la réponse est :

$$\langle Assert, do, \sigma, A, \top, \surd \rangle$$

4.3.4.2 Requêtes de type $\tau = is$

Les requêtes de la forme $(\alpha, \tau) = (How, is)$ modélisent des demandes de plan dans lesquelles l'état but est décrit directement par l'objet ω :

- Si $\sigma \neq \emptyset$ et $\nu = \top$, tout état dans lequel le $\omega \subseteq fils(sujet)$ est un but.
- Si $\sigma \neq \emptyset$ et $\nu = \perp$, tout état dans lequel le $\omega \cap fils(sujet) = \emptyset$ est un but.
- Si $\sigma = \emptyset$ et $\nu = \top$, tout état dans lequel $\forall i \in [1, \omega], v_i = \top$ est un état but, où v_i désigne la valeur de vérité de la relation statique modélisée par ω_i (cf. cas $(Ask, is, \sigma = \emptyset)$:

$$v_i = \begin{cases} (\zeta(\Xi_{rs}(\omega_i, \top), \emptyset) = \{true\}) & \text{si } concept(\omega_i) \in \mathcal{G}_{rs} \\ & \text{si } concept(\omega_i) \in \mathcal{G}_{rp} \end{cases}$$

- Si $\sigma = \emptyset$ et $\nu = \perp$, tout état dans lequel $\forall i \in [1, \omega], v_i = \perp$ est un état but.

Soulignons que si l'état but est décrit pas une conjonction ou une disjonction de valeurs de variables, le MLN doit utiliser une requête relationnelles ($\sigma = \emptyset$) puisque les requêtes telles que $\sigma \neq \emptyset$ décrivent la valeur d'une seule variable à la fois.

Étude du problème

La construction de plan n'étant pas l'objet de notre étude, nous avons choisi de nous appuyer sur les outils de planification existants, fondés sur la logique. Cependant, la modélisation de l'état du composant à un instant donné et des actions possibles dans le langage du constructeur de plans s'est révélée très difficile à automatiser, même en ne prenant pas en compte le fait que les actions peuvent elles-même être modifiées au cours de l'exécution.

Considérons par exemple la vue suivante dans laquelle l'utilisateur peut changer la couleur des figures :

```
view[ figure[shape[square],color[red]],
       figure[shape[circle],color[yellow]],
       figure[shape[square],color[blue]],
       paint[ event[paint[figure,color]],
              guard[equals[size["event get"[figure]],1],
                    put[path["event get"[figure],color],
                        "event get"[color]] ] ] ]
```

Elle se modélise « à la main » en STRIPS par :

```
figure(x1),shape(x1,square),color(x1,red),
figure(x2),shape(x2,circle),color(x2,yellow),
figure(x3),shape(x3,square),color(x2,blue),
op(Action: paint(x,new)
   Precond: figure(x),shape(x,s),color(x,c)
   Effect:  ¬color(x,c),color(x,new) ),
```

Mais cette modélisation ne peut pas se faire de manière automatique sans une information complète sur la structure du composant (quels termes sont des variables,

quel est le type de chaque variable, *etc*). En particulier, en VDLp, les événements externes qu'il faut envoyer à la vue sont généralement partiellement décrits (comme dans l'exemple ci-dessus) et un même concept peut être utilisé par plusieurs termes différents, sans que la sémantique dans chaque cas soit identifiée.

Proposition

Nous ne présenterons pas dans cette thèse un mécanisme de modélisation des vues VDLp et des états buts dans un formalisme logique : ce travail devra faire l'objet d'une étude ultérieure.

Nous considérerons que le système construit un plan permettant d'atteindre l'état but sous la forme d'un ensemble $P \in \mathcal{P}(\Upsilon \times \mathbb{N})$ de couples (e_i, d_i) où e_i est un événement externes devant être envoyé par l'utilisateur à la date d_i . Cette forme constitue en fait un cas particulier des plans construits par l'architecture O-PLAN [25], dans lequel nous ne conservons que les actions externes, c'est-à-dire liée à l'envoi d'un événement¹⁵.

La réponse fournie par le MRF est alors un ensemble de requêtes $\{\rho_i\}_{i \in [1, |P|]}$ telles que :

$$\rho_i = \langle Order, order, \{e_i\}, \emptyset, \top, date[d_i] \rangle$$

4.3.4.3 Requêtes de type $\tau = can$

Les requêtes de la forme $(\alpha, \tau) = (How, is)$ modélisent des demandes de plan dans lesquelles l'état but est décrit par les actions que l'on souhaiterait actives dans cet état.

Pour chaque $\omega_i \in \omega$,

- Si $concept(\omega_i) \in \mathcal{G}_{rp}$ et $\{t_1, \dots, t_n\} = fils(\omega_i)$: s'il existe $j \in [1, n]$ tel que $A_{t_j} = \emptyset$, le MRF retourne une requête :

$$\langle Unknown, do, \sigma, \{t_j\}, \top, \surd \rangle$$

Sinon, soit $T = [t'_1, \dots, t'_n] \in \Upsilon^n$ un ensemble de patterns tel qu'il existe un pattern de relation r tel que $(T, r) \in R$ et il existe $l \in \mathbb{N} \times \Upsilon$ telle que $\forall i \in \mathbb{N}, pm(\Xi_p(t_i, \top), t'_i, \emptyset) = (l, \top)$. Soit r_{inst} l'instanciation de r suivant l et $r' = \Xi_{rs}(r_{inst}, \top)$.

Tout état dans lequel $\zeta(r', \emptyset)_{vue} = \{true\}$ est un état but.

- Sinon, si $A_{\omega_i} = \emptyset$, le MRF retourne une requête :

$$\langle Unknown, do, \sigma, \{\omega_i\}, \top, \surd \rangle$$

Sinon, soit $\Pi = \Pi_{\top}^{(a)} \cup \Pi_{\perp}^{(a)} \setminus \{t tq \chi_{concept}(t, "event get") \neq \emptyset\}$. Tout état dans lequel $\forall t \in \Pi, \zeta(t, \emptyset)_{vue} = \{true\}$ est un état but.

¹⁵Pour la construction de ces plans, l'envoi d'un événement externe est vu comme une action.

Le traitement est ensuite identique au cas (*How, is*) vue précédemment.

Si $|\omega| > 1$, le MRF considère comme état but tout état dans lequel la contrainte modélisée par l'un des ω_i : ω décrit une disjonction de contraintes. De plus, le MRF exige que toutes les actions existent dans la vue. Cette heuristique s'appuie sur l'idée que l'utilisateur ne décrit un but par une action que si l'action existe.

4.3.4.4 Requêtes de type $\tau = order$

Les requêtes $(\alpha, \tau) = (How, order)$ sont de requêtes mal formées : le MRF retourne :

$$\langle Error, is, \{ "malformed request" \}, \emptyset, \top, \surd \rangle$$

4.3.5 Requêtes d'acte $\alpha = Why$

Les requêtes d'acte $\alpha = Why$ sont utilisées pour modéliser les questions de contrôle dans lesquelles l'utilisateur demande des explications au système concernant un état, une action ou une relation, définis dans l'objet de la requête.

Comme pour les requête d'acte *Ask*, l'objet ω ne peut pas être vide et le sujet σ de la requête ne peut être vide que pour les requêtes de type $\tau = is$ modélisant des relations. Lorsque $\tau \in \{do, can, order\}$, le critère σ doit toujours référencer un seul terme de la vue, noté *sujet*.

Comme nous l'avons souligné au premier chapitre, la construction d'une explication *pertinente* s'appuie sur un *modèle causal* (cf. section 1.4.3.2 page 27). Dans cette section, nous proposons une première approche pour la construction d'explication :

- L'explication d'une action est l'ensemble des préconditions valides (resp. invalides lorsque $\nu = \perp$) ;
- L'explication d'un état est l'ensemble des actions effectuées au cycle précédent dont le résultat est (resp. n'est pas) l'état considéré.

Principe général

Dans notre modèle, nous distinguerons trois sortes d'explications :

- Les événements externes (par exemple : « *Pourquoi est-ce que x ? Parce que tu m'as demandé de faire y il y a z minutes.* ») ;
Pour toute requête ρ d'acte *Why*, nous noterons $why_e(\rho) \in \mathcal{P}(\Upsilon \times \mathbb{N})$ l'ensemble des explications événements de ρ ; chaque explication est un couple (e, d) où e désigne un événement envoyé à la vue et d la date de cet envoi ;
- Les relations statiques entre des variables (par exemple : « *Pourquoi est-ce que ta valeur augmente ? Parce que ma vitesse est positive.* ») ;
Pour toute requête ρ d'acte *Why*, nous noterons $why_r(\rho) \in \mathcal{P}(\Upsilon)$ l'ensemble des explications relations de ρ ; chaque explication est l'expression, en VDL et en utilisant éventuellement des termes de ground, d'une relation statique.
- Les actions (par exemple : « *Pourquoi est-ce que tu ne comptes plus ? Parce que suis arrêté.* ») ;

Pour toute requête ρ d'acte *Why*, nous noterons $why_a(\rho) \in \mathcal{P}(\Upsilon \times \mathbf{B})$ l'ensemble des explications relations de ρ ; chaque explication est un couple (a, b) où a modélise une action et b indique s'il s'agit d'une réaction à un événement externe.

La construction d'une réponse à une demande d'explication modélisée par la requête ρ est basée principalement sur la construction des ensembles $why_{x \in \{e, r, a\}}(\rho)$, en fonction du type de ρ .

Réponse

Si $why_e(\rho) \neq \emptyset$, soit $d = \max(\{d_i tq(e_i, d_i) \in why_e(\rho)\})$ et E l'ensemble des événements associés : $E = \{e_i tq(e_i, d) \in why_e(\rho)\}$. Soit $P = \{a tq(a, \perp) \in why_a(\rho)\}$ et $R = \{a tq(a, \top) \in why_a(\rho)\}$.

La réponse produite par le MRF est, dans le cas général, une requête multiple :

$$\left\{ \begin{array}{l} \langle Assert, order, \sigma, E, \top, date[d - \theta] \rangle, \\ \langle Assert, is, \emptyset, why_r(\rho), \top, \surd \rangle, \\ \langle Assert, do, \{view\}, P, \top, \surd \rangle, \\ \langle Assert, do, \{view\}, R, \top, past \rangle \end{array} \right\}$$

Si l'objet ω_x d'une des quatre requêtes est vide, la réponse ne contient pas cette requête. Nous montrerons dans l'algorithme de construction que nous avons toujours au moins une requête, puisqu'il est impossible que $why_e(\rho) = why_r(\rho) = why_a(\rho) = \emptyset$.

4.3.5.1 Requêtes de type $\tau = is$

Les requêtes de la forme $(\alpha, \tau) = (Why, is)$ modélisent des demandes d'explication sur l'état du composant.

Soit $\rho = \langle Why, is, \sigma, \omega, \top, \surd \rangle$ la requête devant être traitée. Le MRF vérifie tout d'abord si le composant est bien dans l'état décrit par ρ (c'est-à-dire si l'utilisateur ne se trompe pas sur l'état de la vue) :

Soit $\bar{\rho}$ la réponse (construite par le MRF) à la requête $\langle Ask, is, \sigma, \omega, \nu, \surd \rangle$.

Si elle est négative ($\bar{\nu} \neq \nu$)¹⁶, la vue n'est pas dans l'état requis par la requête ρ en cours de traitement. Alors la réponse construite par le MRF pour ρ est la requête $\bar{\rho}$.

Sinon, la réponse est construite comme dans le cas général présenté plus haut, à partir des ensembles $why_{x \in \{e, r, a\}}(\rho)$. Nous présentons ci-dessous la construction des $why_x(\rho)$ pour les requêtes de type $\tau = is$, c'est-à-dire pour les variables et les relations statiques.

¹⁶Si $\bar{\rho}$ est une requête multiple, $\bar{\nu}$ est le ν de la première requête, celle qui répond à la question.

Explication des variables

Si $\sigma \neq \emptyset$, soit A l'ensemble des actions qui modifient *sujet*. A est calculé en utilisant les fonctions proposées dans le cas (*What, do, $\omega \neq 0$*) :

$$A = \{a \in \chi_{action}(vue, \surd) \text{ tq } \exists r \in \chi_{object}(a) \text{ tq } calcRef(filts(r), vue) = sujet\}$$

Alors $why_{x \in \{e,r\}}(\rho) = \emptyset$ et $why_a(\rho)$ est construit de la manière suivante : soit P l'ensemble des processus actifs dans A et R l'ensemble des réactions possibles :

$$\begin{aligned} P &= \{a \in A \text{ tq } \mathcal{E}^{(a)} = \Pi_{\perp}^{(a)} = \emptyset\} \\ R &= \{a \in A \text{ tq } \mathcal{E}^{(a)} \neq \emptyset, poss(a)\} \end{aligned}$$

Si $P \neq \emptyset$, $why_a(\rho) = \{(\tilde{p}, \perp), p \in P\}$ est l'ensemble des noms des processus actifs ayant modifié l'action au cours du cycle précédent.

Sinon, pour chaque $r \in R$, soit E_r l'ensemble des couples $(e, d) \in \Theta$ ou e est un événement traité par r :

$$E_r = \{(e, d) \in \Theta \text{ tq } \exists t \in \mathcal{E}^{(r)} \text{ tq } t \succeq e\}$$

Si $\forall r \in R, E_r = \emptyset$, alors $why_a(\rho) = \{(nothing, \perp)\}$. Sinon, Soit d_{max} la date plus récent des événements, pour tout r :

$$d_{max} = \max\left(\bigcup_{r \in R} \{d_i \text{ tq } (e_i, d_i) \in E_r\}\right)$$

Alors $why_a(\rho) = \{(\tilde{r}, \perp), r \in R \text{ tq } \exists e \text{ tq } (e, d_{max}) \in E_r\}$ est l'ensemble des noms des réactions ayant modifié la variable en dernier.

Autrement dit, l'explication de la valeur d'une variable est donnée par la dernière action ayant modifié cette variable.¹⁷

Explication des relations statiques

Si $\sigma = \emptyset$, nous devons expliquer les relations dans ω . Les explications de la requête sont alors données par l'union des explications pour chaque $\omega_i \in \bar{\omega}$ ¹⁸, notées $why_x^{\Upsilon}(\omega_i)$ et construites suivant la méthode présentée ci-après. Ainsi :

$$why_{x \in \{e,r,a\}}(\rho) = \bigcup_{t \in \bar{\omega}} why_x^{\Upsilon}(t)$$

Par construction des $why_x^{\Upsilon}(\omega_i)$, nous n'aurons par $\forall x \in \{r, e, a\}, why_x(\rho) = \emptyset$.

¹⁷L'heuristique utilisée ici est que la valeur de vérité des gardes des actions (processus ou réactions) ne change pas entre le moment où la variable a été modifiée et le moment de l'explication. Nous revenons sur ce problème à la fin de cette section.

¹⁸Nous ne prenons en considération que les éléments de la réponse positive, puisque, comme nous l'avons vu dans le cas (*Ask, is*), les autres sont soit absents de la vue, soit dans un état contraire à celui donné par le critère ν de la requête ρ .

Définition : $why_x^\Upsilon(t)$

Si $t \in \Upsilon$ est un terme VDL, nous noterons $why_e^\Upsilon(t)$ l'ensemble des explications événements de t , $why_r^\Upsilon(t)$ l'ensemble des explications relations de t et $why_a^\Upsilon(t)$ l'ensemble des explications actions de t .

Explications des relations

Nous nous intéressons ici au cas où t modélise une relation statique. $why_{x \in \{e,r\}}^\Upsilon(t)$ est alors calculé de la manière suivante :

- Si $concept(t) \in \mathcal{G}_{rp}$ et $t_1, \dots, t_n = fils(t)$, l'explication est donnée par le corps des actions de la vue intervenant dans la relation, représentées en termes de *ground* :

$$why_{x \in \{e,r\}}(\rho) = \emptyset \quad \text{et} \quad why_a(\rho) = \left\{ (a, \perp), a \in \bigcup_{i \in [1,n]} A_{t_i} \right\}$$

- Sinon, soit $t' = \Xi_{rs}(t, \top)$, $P = \chi_{concept}(t', get)$ l'ensemble de tous les termes utilisés dans la relation. L'explication de t est alors donnée par l'union des explications de ces variables :

$$why_{x \in \{e,r,a\}}(t) = \bigcup_{p \in P} why_x \left(\left\langle Why, is, p, \bigcup_{t' \in calcRef(p, vue)} fils(t'), \top, \sqrt{} \right\rangle \right)$$

L'explication se ramène donc à celle d'une variable, vue précédemment.

Quelques exemples

Voici quelques exemples d'explications produites par le MRF suivant cet algorithme pour le cas $\tau = is$. Ils sont donnés dans l'ordre où apparaissent les différents cas dans cette section :

- *Pourquoi est-ce que ta vitesse vaut 43 ?* Ma vitesse ne vaut pas 43. Elle vaut 37.
- *Pourquoi est-ce que ta valeur vaut 37 ?* Parce que je compte.
- *Pourquoi est-ce que tu es en marche ?* Parce que je n'ai rien fait.
- *Pourquoi est-ce que tu n'es pas en marche ?* Parce que j'ai été arrêté.
- *Pourquoi start est-elle idempotente ?* Parce que *start* change *running* à vrai.
- *Pourquoi est-ce que la valeur est positive ?* Parce que je compte.

4.3.5.2 Requêtes de type $\tau = do$

Les requêtes de la forme $(\alpha, \tau) = (Why, do)$ modélisent des demandes d'explication sur une action. Soit $\rho = \langle Why, do, \sigma, \omega, \top, \sqrt{} \rangle$ la requête devant être traitée. Le MRF vérifie tout d'abord si les actions proposées dans ω sont actives :

Soit $\bar{\rho}$ la réponse à la requête $\langle Ask, do, \sigma, \omega, \nu, \sqrt{\ } \rangle$. Si la réponse est négative ($\bar{\nu} \neq \nu$), la vue n'est pas dans l'état requis par ρ . Alors la réponse fournie par le MRF est $\bar{\rho}$ (avec les explications si c'est un requête multiple).

Sinon, la réponse est construite comme dans le cas général, à partir des ensembles $why_{x \in \{e, r, a\}}(\rho)$, construits par l'union des $why_x^\Upsilon(t)$ de chaque $\omega_i \in \bar{\omega}$:

$$why_{x \in \{e, r, a\}}(\rho) = \bigcup_{t \in \bar{\omega}} why_x^\Upsilon(t)$$

Si $\forall \omega_i \in \omega$ et $\forall x \in \{e, r, a\}$, $why_x^\Upsilon(a) = \emptyset$, alors :

$$why_r(\rho) = why_e(\rho) = \emptyset \quad \text{et} \quad why_a(\rho) = \{(nothing, \perp)\}$$

Explication des actions et des relations procédurales

Si t modélise une relation procédurale sur le comportement ou une action, l'ensemble $why_{x \in \{e, r, a\}}^\Upsilon(t)$ est calculé de la manière suivante :

- Si $concept(t) \in \mathcal{G}_{rp}$, nous devons expliquer la relation procédurale. L'explication est alors donnée par l'instanciation de la relation de contrainte, puisque c'est elle qui caractérise la valeur de vérité de la relation :

Soit $t_1, \dots, t_n = fils(t)$ et $T = [t'_1, \dots, t'_n] \in \Upsilon^n$ tel qu'il existe r tel que $(T, r) \in R$ et $l \in \mathbf{N} \times \Upsilon$ telle que $\forall i \in \mathbf{N}, pm(\Xi_p(t_i, \top), t'_i, \emptyset) = (l, \top)$. Soit r_{inst} l'instanciation de r suivant l (r_{inst} est une relation statique de *ground*). Alors :

$$why_e^\Upsilon(t) = why_a^\Upsilon(t) = \emptyset \quad \text{et} \quad why_r^\Upsilon(t) = \{r_{inst}\}$$

Par exemple, pour répondre à la question « *Pourquoi est-ce que tu comptes à l'envers ?* » (question 14), le MRF recherche la contrainte de la relation *backwards*. C'est cette contrainte (instanciée) qui explique l'activité modélisée dans la questions : « *Parce que ma vitesse est négative* ».

NB : Puisque les relations procédurales portant sur l'activité et le comportement définissent des changements dans l'activité de la vue, la contrainte r associée à une telle relation ne peut pas être *true* (sinon, aucun changement d'activité ou de comportement ne serait représenté).

- Si $concept(t) \notin \mathcal{G}_{rp}$ (t modélise une action), l'explication de t est alors égale à l'union des explications pour chaque $a \in A_t$. Nous sommes alors ramené à l'explication d'une *action de la vue*.

Explication d'une action de la vue

Soit a une *action de la vue* (c'est-à-dire un terme de la vue qui est une action, au sens de la définition donnée section 2.3.4.2 page 63). L'explication de l'action est définie par l'ensemble de ses préconditions :

- Si $\mathcal{E}^{(a)} \neq \emptyset$, $\nu = \top$ et $\Pi_{\perp}^{(a)} = \emptyset$ ¹⁹, soit $C = \{(t, d) \in \Theta \text{ tq } \exists e \in \mathcal{E}^{(a)} \text{ tq } t \preceq e\}$ l'ensemble des événements reçus par la vue et traités par a .

Si $C = \emptyset$, la branche échoue : $\forall x \in \{e, r, a\}, \text{why}_x^{\Upsilon}(a) = \emptyset$.

Sinon, soit $d_0 = \max\{d \text{ tq } \exists t \in \Upsilon \text{ tq } (t, d) \in C\}$ la date la plus récente d'envoi d'un de ces événements, $T_0 = \{t \text{ tq } (d_0, t) \in \Theta\}$ l'ensemble des événements qui lui sont associés et $d'_0 = (d_0 - \theta)$ sa valeur (négative) relativement à la date courante.

Alors $\text{why}_e^{\Upsilon}(t) = \{(e, d'_0)\}$ et $\text{why}_a^{\Upsilon}(a) = \text{why}_r^{\Upsilon}(a) = \emptyset$.

- Sinon, si $(\Pi_{\perp}^{(a)} = \emptyset) \neq \nu$, c'est-à-dire si l'action est un processus inactif (resp. actif) alors que $\nu = \top$ (resp. $\nu = \perp$), alors la branche échoue :

$$\forall x \in \{e, r, a\}, \text{why}_x^{\Upsilon}(a) = \emptyset$$

- Sinon, l'explication de t est donnée par l'ensemble des gardes (vérifiées ou non, selon ν) de l'action a , exprimées en termes de ground :

$$\text{why}_r^{\Upsilon}(t) = \bigcup_{g \in \Pi_{\nu}^{(a)}} \Xi_{rs}(fils(g), \perp) \quad \text{et} \quad \text{why}_e^{\Upsilon}(t) = \text{why}_a^{\Upsilon}(t) = \emptyset$$

Exemples

Voici quelques exemples d'explications produites par le MRF suivant cet algorithme pour le cas $\tau = do$. Ils sont donnés dans l'ordre où apparaissent les différents cas dans cette section :

- *Pourquoi est-ce que tu comptes ?* Je ne compte pas.
- *Pourquoi est-ce que tu comptes à l'envers ?* Parce que ma vitesse est négative.
- *Pourquoi est-ce que tu augmentes ta vitesse ?* Je n'ai rien fait.
- *Pourquoi est-ce que tu augmentes ta vitesse ?* Parce que tu as cliqué sur « + ».
- *Pourquoi est-ce que tu comptes ?* Parce que je suis en marche.

4.3.5.3 Requêtes de type $\tau = can$

Les requêtes de la forme $(\alpha, \tau) = (Why, can)$ modélisent des demandes d'explication sur une action. Le traitement de ces requêtes est identique au cas do , sauf que l'objet ω peut contenir des événements externes.

Explication d'un événement

Si t est un événement externe, soit $A = \chi_{evt}(vue, t)$ l'ensemble des réactions principales à l'événement t . Alors :

$$\text{why}_{x \in \{e, r, a\}}^{\Upsilon}(t) = \bigcup_{a \in A} \text{why}_x^{\Upsilon}(a)$$

¹⁹ $\mathcal{E}^{(a)} \neq \emptyset$, $\nu = \top$ et $\Pi_{\perp}^{(a)} = \emptyset$ correspond à une réaction possible. Cela signifie que l'événement envoyé a été traité par la vue.

L'explication d'un événement est donc définie par l'explication des actions qui le traitent. Nous nous ramenons donc à l'explication d'une action de la vue, présentée dans la section précédente. Soulignons que nécessairement, nous aurons $\mathcal{E}^{(a)} \neq \emptyset$ et, puisque la réponse n'est pas $\bar{\rho}$, l'événement a été envoyé. Par conséquent, l'explication est nécessairement :

- « *Parce que tu me l'as demandé* » si les préconditions de la réaction principale sont vérifiées ;
- « *Parce que x est faux* » si une précondition (x) n'est pas vérifiée.

4.3.5.4 Requêtes de type $\tau = order$

Dans cette thèse, les requêtes de la forme $(\alpha, \tau) = (Why, order)$ sont considérées comme des requêtes mal formées : le MRF retourne une erreur « *malformed request* ».

En fait, ces requêtes sont utilisées dans une extension de VDL pour les SMA, pour modéliser des demandes de reconnaissance d'intention dans les flux d'interaction entre agents. Nous abordons cette problématique succinctement au chapitre 5 mais nous ne parlerons pas des requêtes (*Why, order*), ni de l'utilisation des requêtes formelles en général pour l'interaction entre agents.

4.3.5.5 Analyse de notre approche

Diagnostic et explications

Comme nous l'avons expliqué au premier chapitre, la construction d'explication et l'étude d'un diagnostic en raisonnement sur les actions sont deux problématiques différentes, bien que proches. En effet, la construction d'une explication [76] est fondée sur l'étude de la causalité [99] : il s'agit de trouver la cause la plus pertinente d'un état donné, parmi un ensemble de causes possibles. Au contraire, le diagnostic [9] s'intéresse à l'étude d'un système dont certains composants peuvent être défaillants. L'objectif est de mettre en évidence le composant défaillant, en fonction des valeurs des autres composants, à partir d'un modèle d'exécution et de fautes du système.

Dans notre étude, nous cherchons à construire des explications à un comportement normal du système (les vues ne sont jamais défaillantes), directement à partir de l'étude du code du composant en VDLp. C'est la raison pour laquelle l'algorithme de construction d'explications que nous avons proposé ici n'est pas fondé sur un algorithme de diagnostic.

Le problème des ramifications

Dès que le fonctionnement entre en jeu dans un système, l'analyse des actions de modifications de l'état du composant fait intervenir le problème des *ramifications*, c'est-à-dire de la possibilité pour une action d'avoir des effets indirects (cf. chapitre 1). Dans la littérature, le problème des ramifications est abordé du point de vue de l'analyse statique des actions : à partir d'une situation s , une action a

conduit le système à une situation s' dans laquelle les préconditions d'une action b deviennent vraies, ce qui active l'action b , qui va conduire à un état s'' , etc.

Dans notre modèle de construction de réponses, nous ne « remontons » pas ces ramifications : nous prenons uniquement en considération la première explication possible, c'est-à-dire :

- Pour une action, l'état associé aux préconditions ;
- Pour un état, l'action y conduisant.

Actions, état et événement

Le mécanisme de construction d'explication que nous avons présenté peut toutefois être utilisé pour construire de manière récursive les trois ensembles $why_r(\rho)$, $why_e(\rho)$ et $why_a(\rho)$ donnant l'ensemble des explications « possibles » d'une requête, et non pas seulement la première cause :

- Lorsque $why_r(\rho)$ ou $why_r^\Upsilon(t)$ est construit, nous recherchons les explications des relations donnés dans ces ensembles. Le résultat est alors l'union des explications pour la requête ou le terme considéré (c'est-à-dire celles retournées normalement par l'algorithme de construction de réponse) avec l'union des explications de ces relations.

Par exemple, dans *Coco*, si nous considérons la question : « *Pourquoi est-ce que tu comptes* » modélisée par la requête $\rho = \langle Why, do, \{view\}, \{count\}, \top, \surd \rangle$, nous avons :

$$why_r(\rho) = \{get[running]\}$$

L'explication de $t = get[running]$, construite suivant l'algorithme proposé pour les relations statiques dans le cas $\tau = is$, est donnée par :

$$why_{x \in \{e,r,a\}}^\Upsilon(t) = why_x(\langle Why, is, \{running\}, \{true\}, \top, \surd \rangle)$$

qui retourne :

$$why_a(t) = \{(start, \top)\} \quad et \quad why_{x \in \{e,r\}}(t) = \emptyset$$

L'ensemble des explications $why_{x \in \{e,r,a\}}(\rho)$ est alors donné par l'union de ces ensembles :

$$\begin{aligned} why_r(\rho) &= \{get[running]\} \\ why_a(\rho) &= \{(start, \top)\} \\ why_e(\rho) &= \emptyset \end{aligned}$$

- Lorsque $why_a(\rho)$ ou $why_a^\Upsilon(t)$ est construit, nous recherchons les explications des actions données dans ces ensembles. Le résultat est alors l'union des explications pour la requête ou le terme considéré avec l'union des explications de ces actions.

Ainsi, pour la question : « *Pourquoi est-ce que tu comptes* », nous pouvons à nouveau expliquer l'action $a = start[\dots]$ dont le nom est donné dans l'ensemble $why_a(\rho)$ construit ci-dessus. l'explication de ce terme est définie par

l'algorithme proposé pour les actions dans le cas $\tau = do$. En supposant que l'utilisateur ait envoyé l'événement *start* au cycle 12 de l'exécution de la vue, nous aurons :

$$why_e(a) = \{(start, 12)\} \quad \text{et} \quad why_{x \in \{r,a\}}(a) = \emptyset$$

L'ensemble des explications $why_{x \in \{e,r,a\}}(\rho)$ est alors donné par l'union de tous ces ensembles, ce qui donne finalement :

$$\begin{aligned} why_r(\rho) &= \{get[running]\} \\ why_a(\rho) &= \{(start, \top)\} \\ why_e(\rho) &= \{(start, 12)\} \end{aligned}$$

Les explications-événements, en revanche, ne sont pas expliquées à leur tour (seuls les événements donnés dans l'objet ω de la requête initiale sont expliqués) : cette propriété est nécessaire pour que l'algorithme récursif termine.

Modèle causal et construction d'explication

Dans notre modèle, nous donnons seulement la première explication possible (au sens de l'ordre de construction dans l'algorithme récursif) : nous supposons que si cette explication ne convient pas, l'utilisateur peut demander plus d'explications. Cette méthode présente l'avantage d'être robuste et elle nous permet de nous affranchir de l'utilisation d'un modèle causal (*cf.* section 1.4.3.2 page 27). Cependant, la non-pertinence des réponses formelles engendrées conduit à un dialogue moins naturel :

Q : Pourquoi tu comptes ?
 R : Parce que je suis en marche.
 Q : Oui, mais pourquoi ?
 R : Parce que j'ai été démarré.

Réponse pertinente

De manière générale, les explications fournies par le MRF pour les requêtes *Why* ne sont pas pertinentes en ceci qu'elles donnent l'ensemble des explications possibles, lesquelles peuvent être contradictoires entre elles (comme dans le cas des actions concurrentes).

Pour résoudre ce problème, nous pensons qu'il est nécessaire de s'appuyer sur des techniques de raisonnement comme celles qui sont mises en œuvre dans la construction d'un diagnostic, à partir d'un modèle de relations entre les variables du système. Nous n'avons malheureusement pas pu étudier ce problème plus en détail dans cette thèse.

4.3.6 Requêtes d'acte $\alpha = WhatIf$

Les requêtes d'acte $\alpha = WhatIf$ sont utilisées pour modéliser les questions de contrôle dans lesquelles l'utilisateur demande au système quelles seraient les conséquences d'un état différent, de l'exécution d'une action donnée ou d'un événement externe.

Le sujet σ de la requête ne peut jamais être vide et il doit toujours référencer un seul terme de la vue (sinon le MRF retourne une erreur); nous notons alors *sujet* le terme référencé. L'objet ω de la requête ne peut pas non plus être vide puisque c'est lui qui définit l'hypothèse formulée par l'utilisateur.

Enfin, nous ne prendrons pas en compte les termes de *ground* dans les requêtes d'acte *WhatIf*. Nous considérerons que le MRF retourne une erreur dès lors qu'un élément $\omega_i \in \omega$ est un terme de *ground*. En effet, l'étude des relations procédurales dans les requêtes d'acte *WhatIf* se ramène à l'étude des relations statiques (puisque toute action de *ground* ou relation procédurale est caractérisée par une relation de contrainte). Pour répondre à ces requêtes, il faut alors un modèle de fonctionnement qualitatif, comme dans QPT [44] ou QSIM[74]. Nous ne souhaitons pas aborder cette étude dans notre thèse.

4.3.6.1 Requêtes de type $\tau = is$

Les requêtes de la forme $(\alpha, \tau) = (WhatIf, is)$ modélisent des demandes de prévision à partir d'un état. Le MRF calcule l'ensemble des actions qui ne sont pas possibles mais qui le seraient si *sujet* avait (ou n'avais plus si $\nu = \emptyset$) pour attribut l'un des éléments de ω (ou inversement). La réponse est alors une requête de type $\tau = can$ décrivant l'ensemble des actions « qui deviendraient (im)possibles » dans le composant. Pour construire cette réponse, nous utilisons deux ensembles R_{\top} et R_{\perp} des actions qui changeraient d'état d'activité.

Réponse

Soit A_{\top} l'ensemble des actions possibles dans la vue et A_{\perp} l'ensemble des actions impossibles :

$$\begin{aligned} A_{\top} &= \{t \in \chi_{action}(vue, \surd) tq\,poss(t)\} \\ A_{\perp} &= \{t \in \chi_{action}(vue, \surd) tq\,\neg poss(t)\} \end{aligned}$$

Pour tout $b \in \mathbf{B}$, pour tout $a \in A_b$, pour tout $\omega_i \in \omega$, soit v_i la vue dans laquelle $files(sujet) = \{\omega_i\}$ ²⁰. Soit $a' = correspondent(a, v_i)$ le correspondant de a dans v_i (cf. page 71). Si $poss(a') = \neg b$, alors $\tilde{a} \in R_{\neg b}$.

Alors la réponse à une requête d'acte *WhatIf* et de type *is* est une requête multiple :

$$\{\langle Assert, can, \sigma, R_{\top}, \top, \surd \rangle, \langle Assert, can, \sigma, R_{\perp}, \perp, \surd \rangle\}$$

²⁰La modification temporaire de l'attribut d'un terme, utilisée pour la construction de la réponse, est un mécanisme dépendant de l'implémentation et ne sera par présenté ici.

Si l'un des deux ensembles est vide, la réponse est une requête simple.

Exemple

Considérons un composant similaire à *Coco* mais muni d'une action *eat* conditionnée par le fait que *running* vaut *false*. Alors, si l'utilisateur pose la question : « *Que se passerait-il si tu n'étais plus en marche ?* », modélisée par la requête $\langle \textit{WhatIf}, is, \{running\}, \{true\}, \perp, \surd \rangle$, le MRF répond :

$$\{ \langle \textit{Assert}, can, \{view\}, \{eat\}, \top, \surd \rangle, \langle \textit{Assert}, can, \{view\}, \{count\}, \perp, \surd \rangle \}$$

qui sera utilisé par le MLN pour produire la réponse : « *Je pourrais manger, mais je ne pourrais plus compter.* ».

Pertinence de la réponse

L'algorithme proposé pour les requêtes d'acte *WhatIf* ne permet pas de construire des réponses pertinentes. En effet, si une variable intervient dans la précondition de plusieurs actions, le MRF retourne une réponse décrivant toutes les actions changeant d'état, sans faire ressortir une action principale.

C'est la raison pour laquelle il est important d'utiliser la notion de sous-actions en VDLp : l'algorithme de recherche d'actions retourne en effet uniquement l'action la plus englobante. Ce mécanisme permet de faire ressortir les actions principales par rapport aux sous-actions.

Le même problème se pose pour les requêtes *WhatIf* de type *do*, *can* ou *order* : si une action modifie beaucoup de variables, la réponse construite par le MRF donnera toutes les variables modifiées, sans faire ressortir les variables importantes.

Remarque

Nous pourrions tenir compte de la nature des actions pour construire des réponses plus pertinentes, dans lequel les réactions sont données par des requêtes *can* et les processus par des requêtes *do*. Ainsi, à la question « *Que se passerait-il si tu n'étais plus en marche ?* », en supposant que *eat* soit une réaction, le MRF pourrait répondre : « *Je pourrais manger mais je ne compterais plus* ». Cette extension, relativement simple, n'a pas été présentée formellement ici pour mieux mettre en évidence le principe général de l'algorithme.

4.3.6.2 Requêtes de type $\tau = do$

Les requêtes de la forme $(\alpha, \tau) = (\textit{WhatIf}, do)$ modélisent des demandes de prévision de résultat d'une (ou plusieurs) action(s), processus ou réaction. Le critère ν d'une telle requête ne peut donc jamais être négatif.

Si l'action ω_i est possible, le MRF retourne une requête de type $\tau = is$ décrivant l'ensemble des postconditions de l'action. Sinon, le MRF exprime que l'action n'est

pas possible par une requête de type *can*. Les réponses portant sur des actions possibles sont privilégiées sur celles portant sur les actions impossibles ou inconnues.

Réponse

Pour construire la réponse, nous utilisons trois ensembles : P pour les requêtes modélisant les postconditions des actions possibles, N pour les actions impossibles et U pour les actions inconnues.

Pour chaque $\omega_i \in \omega$, si $A_{\omega_i} = \emptyset$, $\omega_i \in U$. Sinon, pour chaque $a \in A_{\omega_i}$, si $\neg poss(a)$, $\tilde{a} \in N$. Sinon, soit $A_p^{(a)}$ l'ensemble des actions élémentaires dans a :

$$A_p^{(a)} = \bigcup_{c \in \{add, put, del\}} \chi_{concept}(a, c)$$

Pour chaque $t \in A_p^{(a)}$:

- Si $concept(t) = del$, soit $R = \{fils(t) \text{ tq } concept(t) = path\}$ l'ensemble des références aux termes modifiés par l'action élémentaire t . Alors, pour chaque $r \in R$:

$$\langle Unknown, is, fils(r), \emptyset, \top, \surd \rangle \in P$$

Cette réponse modélise le fait que le terme (référéncé par r) ne sera plus présent dans la vue.

- Si $concept(t) = put$, soit $R = \{fils(t) \text{ tq } concept(t) = path\}$ et \bar{R} l'ensemble des interprétations canoniques des attributs ajoutés :

$$\bar{R} = \bigcup_{fils(t) \setminus R} \zeta(t, \emptyset)$$

Alors, pour chaque $r \in R$:

$$\langle Assert, is, fils(r), \bar{R}, \top, \surd \rangle \in P$$

Cette réponse modélise le fait que le terme (référéncé par r) aura pour attribut ceux donnés par l'action a (c'est-à-dire \bar{R}).

- Si $concept(t) = add$, soit R et \bar{R} définis comme ci-dessus. Pour chaque $r \in R$, soit $Res_r = calcRef(fils(r), vue)$ l'ensemble des termes référéncés par r . Alors, pour chaque r et pour chaque $r' \in Res_r$:

$$\langle Assert, is, fils(r), \bar{R} \cup fils(r'), \top, \surd \rangle \in P$$

Cette réponse modélise le fait que le terme (référéncé par r) aura pour attribut ses attributs actuels, plus ceux donnés par l'action a (c'est-à-dire \bar{R}).

La réponse construite par le MRF pour une requête d'acte *WhatIf* et de type *do* est alors :

- P si $P \neq \emptyset$;
- $\langle Assert, can, \sigma, N, \perp, \surd \rangle$ si $U = \emptyset$ et $N \neq \emptyset$;
- $\langle Unknown, do, \sigma, U, \top, \surd \rangle$ sinon.

Exemple

Si l'utilisateur demande à *Coco* : « *Que se passerait-il tu t'arrêtais ?* », le MLN construit la requête $\langle \textit{WhatIf}, do, \{\textit{view}\}, \{\textit{stop}\}, \top, \surd \rangle$ et le MRF répond :

$$\langle \textit{Assert}, is, \{\textit{running}\}, \{\textit{true}\}, \top, \surd \rangle$$

qui sera utilisé par le MLN pour produire la réponse : « *Je serais en marche.* ».

4.3.6.3 Requêtes de type $\tau = can$

Les requêtes de la forme $(\alpha, \tau) = (\textit{WhatIf}, can)$ modélisent des demandes de prévision de résultat d'actions ou d'événements externes, sans tenir compte des préconditions. Comme pour les requêtes de type *do*, le critère ν ne peut jamais être négatif.

Le MRF retourne alors une requête de type $\tau = is$ décrivant l'ensemble des postconditions des actions de ω connues dans la vue. La construction de la réponse par le MRF utilise donc le même que pour les requêtes de type *do*, à deux détails près :

- Nous ne construisons pas d'ensemble N : toutes les actions sont soit inconnues ($\omega_i \in U$), soit utilisées pour construire l'ensemble P ;
- Lorsque ω_i est un événement externe, A_{ω_i} est remplacé par $\chi_{ext}(\omega_i)$.

La réponse construite par le MRF pour une requête d'acte *WhatIf* et de type *can* est alors P si $P \neq \emptyset$, $\langle \textit{Unknown}, do, \sigma, U, \top, \surd \rangle$ sinon.

Exemple

Si l'utilisateur demande à *Coco* : « *Que se passerait-il tu pouvais compter ?* », le MLN construit la requête $\langle \textit{WhatIf}, can, \{\textit{view}\}, \{\textit{count}\}, \top, \surd \rangle$ et le MRF répond²¹ :

$$\langle \textit{Assert}, is, \{\textit{value}\}, \{37\}, \top, \surd \rangle$$

qui sera utilisé par le MLN pour produire la réponse : « *Ma valeur serait 37.* ».

Utilisation d'un critère ν négatif

Nous n'avons pas voulu aborder ici le traitement des requêtes (\textit{WhatIf}, can) avec un critère $\nu = \perp$. Cependant, ce type de requêtes peut être utilisé pour modéliser les questions comme : « *Que se passerait-il si je ne pouvais plus compter ?* ». Pour répondre à ces questions, le MRF doit évaluer les effets directes de l'invalidité des gardes des cations considérées, ce qui ramène le problème au cas (\textit{WhatIf}, is) avec des relations. C'est la raison pour laquelle nous ne l'avons pas étudié ici.

²¹En supposant, par exemple, que la valeur est 34 et la vitesse 3.

4.3.6.4 Requêtes de type $\tau = order$

Les requêtes de la forme $(\alpha, \tau) = (WhatIf, order)$ modélisent des demandes de prévision de résultat d'événements externes, en tenant compte des préconditions. Comme pour les requêtes de type *do* et *can*, le critère ν ne peut jamais être négatif.

Le MRF retourne alors une requête de type $\tau = is$ décrivant l'ensemble des post-conditions des réactions possibles à un événement donné dans ω si ces réactions sont possibles. Sinon, il retourne une requête de type *can* exprimant que ces événements ne peuvent pas être traités.

La construction de la réponse par le MRF utilise donc le même que pour les requêtes de type *do*, à deux détails près :

- A_{ω_i} est remplacé par $\chi_{evt}(\omega_i)$;
- Pour chaque $a \in \chi_{evt}(\omega_i)$, si $\neg poss(a)$, $\omega_i \in N$;

La réponse construite par le MRF pour une requête d'acte *WhatIf* et de type *order* est alors :

- P si $P \neq \emptyset$;
- $\langle Assert, can, \sigma, N, \perp, \surd \rangle$ si $U = \emptyset$ et $N \neq \emptyset$;
- $\langle Unknown, do, \sigma, U, \top, \surd \rangle$ sinon.

4.4 Bilan

Dans ce chapitre, nous avons proposé des algorithmes de construction de réponses aux requêtes construites par le MLN pour modéliser les questions sur le fonctionnement, dans le cadre d'une interaction entre un utilisateur ordinaire et une vue décrite en VDL. Nous avons mis en évidence l'existence de *requêtes de bon sens* (CSR) faisant intervenir des notions de bon sens, partagées par tous les utilisateurs mais non explicitées dans la description du composant. Nous avons présenté un cadre formel, appelé *Ground*, qui permet de modéliser et de manipuler des connaissances de bon sens dans les requêtes.

Le formalisme utilisé pour représenter et manipuler ces connaissances est le langage VDL, mais le problème des CSR n'est pas spécifique au projet *InterViews* : il apparaît de plus en plus crucial dans le cadre de l'interaction homme-machine, indépendamment du langage de requêtes et du modèle de représentations des connaissances dans le composant.

Le *Ground* que nous avons proposé ne prend en considération que les connaissances sur le fonctionnement, indépendamment des autres éléments constitutifs d'un composant. En particulier, nous n'avons pas étudié le lien entre l'interface graphique et la perception des effets des actions par l'utilisateur. Nous avons considéré que cette perception pouvait être modélisée dans le traitement de la référence spatiale et structurelle. Les études récentes [128] montrent que ce n'est pas si simple : seule une analyse couplée du fonctionnement et de la représentation des effets permettrait d'interagir de manière naturelle avec l'utilisateur, en modélisant l'ensemble des connaissances de bon sens liées à la perception de l'utilisateur.

Nous avons ensuite proposé des méthodes formelles de traitement des requêtes construites par le MLN ne faisant pas intervenir de notions temporelles, en nous appuyant sur l'analyse des questions du corpus et de leur représentation dans le langage de requêtes proposé au chapitre précédent. Les algorithmes donnés dans ce chapitre posent les bases d'un mécanisme de construction de réponse aux questions sur le fonctionnement que peut poser un utilisateur humain en interaction avec une vue. Nous n'avons pas traité en détail les requêtes de demande de plan et d'explications ($\alpha \in \{How, WhatIf, Why\}$), déjà largement étudiés dans la littérature, mais nous avons montré que le traitement des requêtes d'acte $\alpha \in \{What, Ask\}$ définissait l'ensemble des mécanismes nécessaires pour la construction des réponses dans le cas général.

Nous n'avons pas abordé le problème de la complexité des algorithmes de construction de réponse présentés dans ce chapitre. Pourtant, c'est un problème primordial du point de vue de l'utilisateur qui interagit avec le composant. En théorie, puisqu'il s'agit uniquement de recherches dans un arbre (la vue), la complexité de tous les algorithmes devrait être égale à $\mathcal{O}(n)$ où n est le nombre de nœuds dans la vue. En pratique, les algorithmes que nous avons écrit utilisent plusieurs recherches, parfois incluses dans une recherche. La complexité de ces algorithmes peut alors rendre impossible l'application des mécanismes de construction de réponses sur des composants complexes, comme nous en présentons au chapitre 7. Cependant, il est toujours possible d'adapter les algorithmes pour ne pas « chercher plusieurs fois la même chose », c'est à dire ne parcourir la vue qu'une fois, ou un nombre constant de fois. En pratique, l'écriture de ces algorithmes dépasse largement le cadre de ce que nous souhaitons étudier dans cette thèse. Par exemple, nous avons écrit une fonction χ_{proc} qui recherche un ensemble d'actions dans la vue. Son algorithme est présenté en annexe 7.2.3.2. Il permet certes de trouver toutes les actions de l'objet ω d'une requête en un seul parcours de la vue. Mais compte tenu de la difficulté de l'écriture d'un tel algorithme, nous n'avons pas pu faire de même pour toutes les fonctions. De plus, cela n'apporte rien du point de vue de l'étude du problème des réponses aux requêtes sur le fonctionnement.

Enfin, le modèle que nous avons proposé est cependant encore limité sur plusieurs points et nous envisageons plusieurs extensions possibles pour pouvoir répondre à une plus large classe de questions sur le fonctionnement :

1. Pour pouvoir modéliser une plus grande classe de question, et en particulier les requêtes *structurelles* « non naturelles » utilisées dans les modèles classiques (PROLOG, SQL, etc), pourrions utiliser les termes non instanciés $\#i$. Par exemple, la question « *Qu'est-ce que est négatif* » pourrait se modéliser par une requête : $\langle What, is, \{\#1\}, \{negative[get[\#1]]\}, \top, \surd \rangle$. Le MRF pourrait alors instancier et vérifier la relation, puis donner l'ensemble des instances de $\#i$, en utilisant un algorithme « à la PROLOG ».
2. Les préconditions booléennes des actions sont utilisées pour construire des explications aux questions sur le fonctionnement. Malheureusement, ces explications ne sont pas toujours pertinentes. Par exemple si une action a a pour

précondition « $a \vee b$ est vrai » ($guard[or[get[a], get[b]]]$ en VDL), l'explication produite sera : « Parce que soit a , soit b est vrai ».

Lors de la construction de l'explication, le MRF devra analyser les opérateurs *or* et *and* des gardes et calculer les valeurs de vérité des variables indépendamment, afin de construire une réponse pertinente (par exemple : « Parce que a est vrai » lorsque b est faux). Cette amélioration, relativement simple à mettre en œuvre (il suffit d'utiliser une fonction intermédiaire entre l'extraction de la garde dans $\Pi_x^{(a)}$ et la production de l'explication) ne sera pas détaillée ici.

3. Le MRF ne peut raisonner que sur des actions de la vue. Il ne peut donc pas répondre à des questions portant sur des actions abstraites non instanciées dans la vue, comme « *Est-ce que grandir est l'inverse de rapetisser ?* » dans *Coco*. Pourtant, lorsqu'il s'agit d'action de *ground*, comme « *Est-ce que retirer est idempotent* », les connaissances données dans le *Ground* devraient permettre de répondre à la question.

Pour pouvoir interagir avec des utilisateurs ordinaires, les vues doivent pouvoir travailler de manière transparente sur les actions de *ground*. C'est la raison pour laquelle nous envisageons dans des travaux futurs de proposer un mécanisme de raisonnement spécifique dans nos algorithmes lorsque χ_{action} échoue et que les actions recherchées sont des actions de *ground*.

4. Les requêtes d'acte *Error* sont munies d'une sémantique spécifique par rapport aux autres requêtes. Pour faciliter l'interaction avec le MLN et permettre au MRF de proposer des réponses naturelles en cas d'erreur, nous pourrions choisir d'utiliser les requêtes *Error* comme des requêtes *Assert*, avec des termes VDL dans l'objet de la requête. Par exemple :

$$\langle Error, is, \emptyset, \{ambiguous[subject]\}, \top, \surd \rangle$$

où $ambiguous[\#1]$ est la relation statique de bon sens associée au terme VDL :

$$"greater\ than"["size\ ref"[\#1], 1]$$

5. Le mécanisme de construction d'explication que nous avons proposé permet d'intégrer au sein d'un seul algorithme l'analyse de l'état et des actions d'un composant pour la recherche des causes, lesquelles peuvent être à la fois des états et des actions. Nous pensons que ce mécanisme peut servir de base à une méthode unifiée de construction d'explications pertinentes. Toutefois, la construction de ces explications ne peut pas se faire sans des informations sur la réponse attendue par l'utilisateur, comme nous l'avons montré au premier chapitre. Ces informations, dépendant du contexte de l'interaction, devraient pouvoir être modélisées par le MLN dans les requêtes.
6. Le mécanisme d'explications que nous avons proposé s'appuie uniquement sur l'état courant de la vue. Par exemple, l'explication de la valeur d'une variable x recherche les actions a qui peuvent modifier cette variable dans l'état courant. Elle n'est donc correcte que si les valeurs des variables y entrant en jeu

dans le calcul des valeurs de vérité des préconditions associées à chaque action a n'a pas été modifiée entre le moment où la variable x a été mise dans l'état courant et la date présente. C'est bien évidemment une hypothèse trop forte pour pouvoir être utilisée dans un système réel. Nous voudrions au contraire pouvoir remonter les ramifications en tenant compte des changements d'état, ce qui nécessite de pouvoir mémoriser l'exécution passée du composant. Nous proposons dans le chapitre suivant un mécanisme de mémorisation de l'exécution passée d'une vue qui permet de retrouver les actions effectuées.

7. Enfin, dans la construction des réponses aux requêtes d'acte *Why* et *WhatIf*, nous ne prenons pas en compte le problème des actions concurrentes étudié au chapitre 2. Si plusieurs actions peuvent avoir modifié une même variable t , elles seront toutes données considérées comme une explication possible pour les requêtes *Why* et leurs effets seront confondus dans la réponse à une requête *WhatIf*.

Nous pourrions envisager d'utiliser le mécanisme de gestion de la concurrence que nous avons présenté au chapitre 2 pour déterminer quelle est l'action qui a (aurait pour les requêtes *WhatIf*) réellement modifié la variable.

Chapitre 5

Extraction de comportement à partir des flux d'interaction

Dans le chapitre précédent, nous avons présenté un cadre formel permettant de construire des réponses aux questions de contrôle qu'un utilisateur humain peut poser concernant l'état, l'activité, le comportement et, de manière générale, le fonctionnement d'un composant actif. Nous avons montré qu'il était nécessaire, pour répondre à des requêtes portant sur l'exécution passée du composant, de mémoriser les actions effectuées.

Dans ce chapitre, nous proposons tout d'abord un modèle de mémorisation de l'exécution des vues fondé sur l'enregistrement des situations dans lesquelles se produisent les actions et les interactions avec l'utilisateur ou d'autres composants. Nous montrons que l'analyse des régularités dans ces flux d'interactions, à partir d'une représentation en *intension*, permet de faire ressortir des *schémas comportementaux* dans un composant. Nous proposons alors d'utiliser des techniques d'apprentissage pour extraire des comportements collectifs dans un SMA. Nous donnons ensuite les algorithmes d'utilisation de cette « mémoire des événements » dans le MRF. Nous présentons les mécanismes formels de construction de réponses aux requêtes portant sur l'exécution passée de la vue, puis nous proposons d'utiliser la mémorisation des actions dans les chroniques comme support à la construction d'explications dans les requêtes d'acte *Why*.

5.1 Mémorisation de l'exécution

5.1.1 Analyse du problème

5.1.1.1 Présentation du besoin

Dans le cadre du projet *InterViews*, nous essayons de définir des composants actifs capables de répondre aux questions qu'un utilisateur humain peut poser concernant leurs actions et *leur exécution*. En particulier, nous voulons pouvoir répondre aux requêtes portant sur le passé de la vue, modélisant des questions comme « *Est-ce*

que tu comptais il y a 5 minutes ? » ou « *Quand ais-je mis ta valeur à 2 ?* ».

Les algorithmes de construction de réponses que nous avons proposé au chapitre précédent ne travaillent que sur la description de la vue à l’instant courant. Pour pouvoir les utiliser dans le traitement des requêtes temporelles, nous devrions connaître l’état de l’ensemble de la vue à chaque instant de l’exécution. C’est malheureusement impossible puisque les capacités de mémorisation des ordinateurs sont limitées.

5.1.1.2 Notion de *chronique*

La notion de *chronique* a été introduite par McDermott [92] pour représenter les occurrences possibles d’événements (au sens de la logique des instants) dans l’exécution à partir d’un état donné.

Une chronique est un ensemble d’événements datés, décrivant partiellement une exécution.

La notion de chronique initiale a été étendue pour permettre de décrire dans une chronique non seulement les événements qui doivent se produire au cours de l’exécution, mais aussi ceux qui ne doivent pas être présents. Les chroniques sont donc plus que de simples « traces d’exécution », puisqu’elles modélisent un *schéma comportemental* du système, défini par l’ensemble des événements qui s’y produisent.

L’une des principales utilisations des chroniques est la reconnaissance de ces schémas dans un flot d’événements. L’objectif est alors de reconnaître une chronique, ou le début d’une chronique, dans un flot d’événements datés, afin de mettre en évidence un comportement spécifique du système. Cette technique peut être utilisée, par exemple, pour la détection de panne. C’est le cas de l’outil IxTeT [33, 32], capable de surveiller *en ligne* un système complexe pour prédire les pannes et expliquer leurs causes.

Dans notre étude, l’idée des chroniques peut être utilisée pour mémoriser occurrences d’événements dans la vue, comme dans QPT [44]. En VDL, les événements sont les actions effectuées par les composants, soit en réponse à une interaction, soit de manière proactive dans les processus. L’enregistrement de ces événements dans des chroniques, datées à la manière du calcul des événements [73], permet de retrouver quelle action a été effectuée à quel moment de l’exécution. Il devient alors possible d’utiliser des outils d’analyse statique de l’exécution, comme ceux proposés par le calcul des situations [91] ou logique dynamique [104], pour raisonner sur l’exécution et les changements dans la vue, en réponse à des demandes d’explications ou de plans.

5.1.1.3 Interaction en environnement ouvert

Mais les composant auxquels nous nous intéressons évoluent dans un contexte *ouvert* [61] : les occurrences événements sont imprédictibles et ils peuvent dépendre des connaissances de la vue, contrairement à l’approche *statique* ou les occurrences d’événements (ou actions) possibles sont figées. Par conséquent, pour répondre aux

demandes de l'utilisateur, nous ne voulons pas simplement de répondre à des questions de la forme « *Est-ce que x était vrai à l'instant t* ». Nous voulons aussi pouvoir expliquer le comportement et l'activité passée du composant comme nous l'avons fait pour le temps présent, par exemple pour répondre à la question : « *Est-ce que tu pouvais compter il y a 10 minutes* ».

McCarthy a montré dans [90] que, pour pouvoir mener à bien ce type de raisonnement sur le fonctionnement dans l'exécution passée dans un composant actif, il était nécessaire de savoir ce qu'il s'était *réellement* passé, ce qui ne peut pas être déduit du seul code (statique) de la vue : il faut aussi prendre en considération les événements externes produits par les utilisateurs (humains ou autres agents) et les connaissances de la vue lors du traitement de ces événements.

C'est la raison pour laquelle nous proposons dans cette section un modèle de chroniques spécifique. Pour chaque événement interne (action) ou externe se produisant dans le composant, nous construisons un modèle de la situation (au sens de Kintsch [71]) dans laquelle il s'est produit et nous enregistrons cette situation. Nous construisons ainsi des chroniques sur lesquelles s'effectuera le travail de raisonnement.

Cette notion de situation permet que les chroniques soient des entités sémantiquement plus riches que de simples traces d'exécution : nous pouvons parler de *traces situées*.

5.1.1.4 Notion d'*intension*

En informatique, une liste (finie) de valeurs peut être représentée :

- En *extension*, par l'énumération de ses valeurs. Par exemple, la liste des dix premiers entiers naturels : $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$.
- En *intension* (avec un s), c'est-à-dire dans une représentation compacte qui peut être utilisée pour reconstituer la représentation en extension, à l'aide d'un algorithme. Par exemple, dans le langage Pascal, nous écrivons $[0..9]$ pour la liste des dix premiers entiers naturels.

Il existe en général une infinité d'intension pour une même extension. Au contraire, une intension donnée ne peut être utilisée que pour calculer une seule extension.

La représentation en intension d'une liste permet :

- De rendre l'espace mémoire utilisé indépendant du nombre d'éléments dans la liste (par exemple, $[0..100]$ ne coûte pas plus en mémoire que $[0..9]$);
- De donner une vision plus globale des données et de faire ressortir les régularités dans la liste.

Nous proposons d'utiliser ces deux avantages pour la mémorisation de l'exécution passée des vues.

5.1.1.5 Intensionnalisation des chroniques

Si nous considérons un composant actif muni de chroniques qui mémorisent les actions et les situations dans lesquelles elles se produisent :

- D'un point de vue opérationnel, l'état s_t du composant croît linéairement avec le nombre de cycles d'exécution, au fur et à mesure que les proactions s'effectuent.

Pour pouvoir mémoriser l'ensemble de l'exécution, nous ne pouvons donc pas conserver une représentation en intension. Il faut donc représenter les chroniques en intension.

- D'un point de vue sémantique, comme nous l'avons souligné, ne nous voulons pas simplement répondre à des requêtes factuelles sur l'exécution passée des vues. Nous voulons au contraire proposer un mécanisme de raisonnement sur le fonctionnement des composants pour répondre à des requêtes en utilisant une analyse globale de l'activité de la vue.

Dans ce cadre, il est nécessaire de représenter les chroniques en intension pour faire ressortir les régularités dans les occurrences des événements passés et extraire ainsi des schémas comportementaux qui peuvent servir de support à un mécanisme explicatif.

Dans notre étude, les chroniques ne sont pas utilisées pour modéliser un comportement spécifique, comme dans [34]. Au contraire, nous voulons extraire de manière automatique des comportements dans les chroniques des flots d'interactions.

Dans ce chapitre, nous proposons un modèle dynamique pour construire les chroniques en intension et nous essayons de montrer comme cela peut être utilisé d'une part pour détecter des comportements dans la vue, et d'autre part pour servir de support à la construction des explications.

5.1.1.6 Exemple : le *frigo*

Dans les sections suivantes, nous utiliserons un exemple volontairement assez trivial pour illustrer le mécanisme d'intensionnalisation des chroniques.

Considérons une vue modélisant un réfrigérateur, interagissant à la fois avec un utilisateur humain et avec l'environnement :

- L'environnement augmente la température du frigo suivant une loi temporelle à chaque cycle d'exécution. Nous supposons que la température de la pièce soit de 20°C. La température $temp$ à l'intérieur du frigo croît alors en fonction du temps suivant la loi :

$$temp = 20(1 - e^{(-\frac{t}{T})})$$

dont la différentielle est :

$$temp_{(t+1)} = e^{-\frac{1}{T}} \times (temp_{(t)} - 20) + 20$$

La valeur de $temp$ au cycle suivant est donc calculée par le terme VDL suivant :

```

plus[ times[ exp[times[-1,inverse[get[T]]]],
          plus[get[temp],-20]],
      20 ]

```

où *exp* est le modifieur arithmétique utilisé pour l'opération unaire « exponentielle », que nous ne détaillerons pas ici.

- L'utilisateur peut ouvrir et fermer la porte du frigo et, lorsque la porte est ouverte, la température augmente plus rapidement.

Du point de vue de la programmation de la vue, cette augmentation de la pente correspond à un facteur de diffusion T plus petit. Nous avons choisi :

- $T = 1500^{\circ}C.min^{-1}$ lorsque la porte est fermé (la température du frigo serait alors de $12,5^{\circ}C$ après 24 heures en partant de $0^{\circ}C$);
- $T = 40^{\circ}C.min^{-1}$ lorsqu'elle est ouverte ($\theta = 8^{\circ}C$ après 20 minutes¹).
- Le frigo est muni d'un système de refroidissement qui se déclenche lorsque $\theta > 4^{\circ}C$. Il a pour effet de diminuer linéairement la température du frigo de $0.2^{\circ}C$ à chaque cycle d'exécution.

Ce mécanisme est modélisé dans la vue par un processus dont le nom est "cool down".

La description en VDL de ce petit monde opératoire est la suivante :

```

view[ temp[3], T[1500],
      "open door"[ event[open], put[path[T], 40] ],
      "close door"[ event[close], put[path[T], 1500] ],
      "heat up"[ put[path[temp],
                    plus[ times[ exp[times[-1,inverse[get[T]]]],
                              plus[get[temp],-20]],
                        20 ]
                ],
      "cool down"[ guard["greater than"[get[temp],4]],
                   put[path[temp],plus[get[temp],-0.2]] ]
]

```

L'utilisateur peut interagir avec le composant en utilisant les événements *open* et *close* pour respectivement ouvrir et fermer la porte du frigo.

5.1.2 Chroniques en extension

Afin de pouvoir extraire des régularités dans les actions effectuées, nous enregistrons l'exécution du composant dans des chroniques qui sont ensuite intensionnalisées. Notre objectif est d'attacher à toute *variable* VDL une chronique intensionnalisée, *i.e.* la représentation sous forme compacte des actions qui l'ont modifié, afin d'étudier leurs régularités et de détecter des comportements.

¹Par conséquent, le cycle d'exécution de la vue a une période d'une minute. Il s'agit ici simplement d'un exemple.

Dans cette section, nous définissons la structure de ces chroniques en extension et la manière dont elles sont construites à l'exécution à partir d'un flot d'événements. Nous présentons leur intensionnalisation dans la section suivante.

5.1.2.1 Notion de *situation*

Rappel

L'ensemble des actions devant être effectuées à chaque cycle de l'exécution est mémorisé dans la liste des modifications μ . Les éléments de μ sont des quadruplets (t, a, R, E) où t est le terme modifié, a le concept de l'action et R les attributs à ajouter, pour $a \in \{add, put\}$.

Pour toute action a de la vue et pour tout terme non flottant t , nous dirons que a est *effectuée* sur t si et seulement si la fonction φ dans ϕ_{exec} atteint l'action a et construit dans μ un quadruplet (t, a, R, E) .

Situation

Soit a une action effectuée sur un terme t . La *situation* dans laquelle cette action est produite (par φ) est modélisée par un quintuplet :

$$sit(a) = \langle who(a), what(a), why(a), when(a), res(a) \rangle$$

dans lequel :

- $who(a) = pere(a)$ est le terme parent de l'opération.
- $what(a) = a$ est l'action élémentaire effectuée.
- $why(a) = \Pi_a$ donne l'ensemble des préconditions de l'action.
Le critère why sera utilisé pour déterminer les causes des actions et, à terme, pour synthétiser le comportement du composant.
- $when(a)$ représente la date à laquelle l'opération a été effectuée.
- $res(a)$ est la nouvelle valeur du terme après cette opération. C'est ce critère qui sera utilisé pour extraire des comportements dans les chroniques.

5.1.2.2 Chroniques extensionnelles

Pour toute variable t de la vue, nous enregistrons les situations de toutes les actions effectuées sur t au cours de l'exécution dans une *chronique extensionnelle* κ_t , au fur et à mesure de l'exécution. Cette chronique qui peut être vue comme un tableau attribut-valeur dont les colonnes correspondent aux cinq critères caractérisant une situation et les lignes sont les situations associées à chaque action effectuée.

Exemple

Considérons la vue *Frigo* définie précédemment et supposons que nous avons l'interaction suivante :

- La température initiale du frigo est de $3^{\circ}C$;
- Au cycle 3, l'utilisateur ouvre la porte ;
- Il la referme au cycle 6 ;
- Il la rouvre au cycle 10.

Le système de refroidissement du frigo (représenté par le terme "*cool down*") régule la température lorsque cela est nécessaire.

La chronique extensionnelle κ_{temp} associée au terme *temp* dans la vue $\phi^{11}(Frigo)$, *i.e.* au cycle d'exécution 11, est la suivante :

who	what	why	when	res	<i>Interactions</i>
✓	✓	✓	0	3	
heat up	put[path[temp], Δ]	\top	1	3.011	
heat up	put[path[temp], Δ]	\top	2	3.023	
heat up	put[path[temp], Δ]	\top	3	3.034	<i>open</i>
heat up	put[path[temp], Δ]	\top	4	3.453	
heat up	put[path[temp], Δ]	\top	5	3.861	
heat up	put[path[temp], Δ]	\top	6	4.260	
cool down	put[path[temp],...]	guard[...]	6	4.060	<i>close</i>
heat up	put[path[temp], Δ]	\top	7	4.070	
cool down	put[path[temp],...]	guard[...]	7	3.870	
heat up	put[path[temp], Δ]	\top	8	3.881	
heat up	put[path[temp], Δ]	\top	9	3.892	
heat up	put[path[temp], Δ]	\top	10	3.903	<i>open</i>
heat up	put[path[temp], Δ]	\top	11	4.300	
cool down	put[path[temp],...]	guard[...]	11	4.100	

Supposons maintenant qu'au cycle 11, l'utilisateur oublie de fermer la porte. La chronique extensionnelle κ_{temp} dans la vue $\phi^{15}(Frigo)$ serait alors :

who	what	why	when	res	<i>Interactions</i>
heat up	put[path[temp], Δ]	\top	12	4.493	
cool down	put[path[temp],...]	guard[...]	12	4.293	
heat up	put[path[temp], Δ]	\top	13	4.681	
cool down	put[path[temp],...]	guard[...]	13	4.481	
heat up	put[path[temp], Δ]	\top	14	4.864	
cool down	put[path[temp],...]	guard[...]	14	4.664	
heat up	put[path[temp], Δ]	\top	15	5.042	
cool down	put[path[temp],...]	guard[...]	15	4.842	

Alors que les changements de valeur de la température dans la première partie de l'exécution sont irréguliers, la seconde partie montre une activité constante. Nous allons montrer dans les sections suivantes comment l'intensionnalisation de chroniques permet de faire ressortir ce schéma comportemental.

5.1.3 Chroniques en intension

Considérons une chronique en extension κ_t . Notre objectif est de la représenter en intension, d'une part pour réduire l'espace mémoire utilisé par la chronique pour mémoriser les actions qui modifient t , et d'autre part pour extraire des comportements dans les modifications de t .

5.1.3.1 Compaction de chroniques extensionnelles

Soulignons que les critères *who*, *what* et *why* d'une situation définissent l'action elle-même alors que *when* et *res* décrivent respectivement la date et l'effet de l'action. Par conséquent, pour une action donnée, *who*, *what* et *why* resteront constant au cours de l'exécution.

Une première possibilité pour compacter κ_t consiste alors à regrouper les situations associées à une même action.

Si deux situations s et s' dans κ_t ont les mêmes critères *who*, *what* et *why*, nous dirons qu'elles sont associées à la *même action*.

Nous pouvons alors construire une chronique compacte κ'_t dans laquelle chaque ligne correspond non pas à une situation, mais à une *action*. La description de l'exécution d'une action sur un terme t dans une telle chronique est donc définie par un quintuplet :

$$\langle who, what, why, W, R \rangle$$

dans lequel W est la liste des dates d'occurrence d'événement ($when(a)$) et R la liste des valeurs dans les nouvelles situations ($res(a)$) telles que $|W| = |R|$ et $\forall i \in [1, |W|]$:

$$\langle who, what, why, W_i, R_i \rangle \in \kappa_t$$

Exemple

Dans notre exemple, pour $\kappa'_{temp}(15)$, nous avons trois actions ;

who	what	why	when	res
✓	✓	✓	0	3
heat up	put[path[temp], Δ]	\top	[1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9 ; 10 ; 11 ; 12 ; 13 ; 14 ; 15]	[3.011 ; 3.023 ; 3.034 ; 3.453 ; 3.861 ; 4.260 ; 4.070 ; 3.881 ; 3.892 ; 3.903 ; 4.300 ; 4.493 ; 4.681 ; 4.864 ; 5.042]
cool down	put[path[temp],...]	guard[...]	[6 ; 7 ; 11 ; 12 ; 13 ; 14 ; 15]	[4.060 ; 3.870 ; 4.100 ; 4.293 ; 4.481 ; 4.664 ; 4.842]

5.1.3.2 Intensionnalisation des suites numériques

Pour construire les chroniques intensionnalisées, il suffit alors de représenter en intension les listes R et W .

Découverte de lois

La construction d'une intension à partir d'une liste donnée en extension est un cas particulier d'apprentissage, situé dans le domaine de la découverte de lois numériques, dont l'objectif est de reconnaître une fonction $f : x \mapsto y$ à partir d'un ensemble de couples (x, y) [36, 97]. Cependant, dans notre approche, nous ne travaillons pas avec des fonctions continues : nous utilisons uniquement des ensembles discrets. Nous proposerons donc des mécanismes de construction d'intension spécifiques.

Notion d'itérateur

Nous proposons de représenter toute suite numérique finie par un ensemble d'itérateurs. Un itérateur est un quadruplet (x_0, Δ, t, op) où :

- x_0 est la valeur du premier élément de la liste ;
- Δ est le pas d'itération itérative, c'est-à-dire, celle la valeur qui modélise « l'écart » entre deux éléments x_i et x_{i+1} ;
- t est la taille de la liste ;
- op est l'opération d'itération.

Par exemple, $(0, 1, 10, +)$ est une représentation en intension de la liste des 10 premiers entiers naturels.

La construction de la représentation en extension L d'un itérateur est obtenue par l'algorithme suivant :

```
val =  $x_0$ 
pour i allant de 1 à t
   $L_i = val$ 
  val = val op  $\Delta$ 
```

Opération d'intensionnalisation ξ

Nous notons ξ l'opérateur d'intensionnalisation qui construit une représentation en intension d'une suite numérique finie L quelconque. Nous noterons $\xi(L) = \surd$ lorsque l'opérateur ξ ne peut pas intensionnaliser L (L n'est pas une suite régulière).

Dans cette thèse, nous nous sommes restreints aux quatre opérations arithmétiques de base $(+, -, \times, \div)$:

- Si $\exists \Delta \in \mathbb{N}tq \forall i \in [2, |L| - 1], L_i - L_{i-1} = \Delta$, alors $\xi(L) = (L_1, \Delta, |L|, +)$;
- Si $\exists \Delta \in \mathbb{N}tq \forall i \in [2, |L| - 1], L_i \div L_{i-1} = \Delta$, alors $\xi(L) = (L_1, \Delta, |L|, \times)$.

Dans des travaux futurs, nous devrions étendre l'opérateur ξ pour traiter d'autres opérations, d'autres types de données (listes, enregistrements, etc) et d'autres suites régulières en utilisant les travaux effectués en découverte de lois [36, 97].

Par exemple, dans le cas de la température du frigo, la suite numérique peut être intensionnalisée en un itérateur dans lequel $op = \ddagger_{(a,b)}$ où $\ddagger_{(a,b)}(x) = a.x + b$. Pour les besoins de notre exemple, nous ajoutons donc la règle suivante dans l'algorithme de ξ :

$$- \exists(a, b) \in \mathbb{N}^2 \text{ tq } \forall i \in [2, |L| - 1], L_i = a \times L_{i-1} + b, \text{ alors } \Delta = (a, b) \text{ et } \xi(L) = (L_1, \Delta, |L|, \ddagger_{\Delta}).$$

Intensionnalisation d'une suite numérique

Soit L une suite numérique finie. Nous appelons *segment* de L toute sous liste L' telle qu'il existe un intervalle $[d, f] \subseteq [1, |L|]$ tel que $\forall i \in [d, f], L'_i = L_i$. Nous noterons alors $L_{[d,f]}$ le segment de L commençant à l'indice d et finissant à l'indice f .

L'intensionnalisation de L se fait alors par segment : soit $x \geq 2$ le plus grand nombre tel que :

$$\xi(L_{[0,x]}) \neq \surd$$

Si x n'existe pas (le début de la liste n'est pas régulier), alors nous notons $i_0 = L_0$ et nous recommençons sur $L_{[1,|L|]}$. Sinon, nous notons i_0 l'itérateur $\xi(L_{[0,x]})$ et nous recommençons la même opération sur la liste $L_{[x+1,|L|]}$. Nous obtenons ainsi une liste d'itérateurs i_0, \dots, i_n qui constitue la représentation en intension de la liste L , dont certains sont réduits à une valeur de L .

Exemple

Soit $L = [1, 2, 3, 4, 6, 8, 16, 32, -1, -2]$. En se limitant aux quatre opérations arithmétiques de base, la liste L s'intensionnalise en une suite de 5 itérateurs :

$$\begin{aligned} i_0 &= (1, 1, 4, +) \\ i_1 &= 6 \\ i_2 &= (8, 2, 3, \times) \\ i_3 &= -1 \\ i_4 &= -2 \end{aligned}$$

Soulignons que ce n'est ni la seule intension par itérateurs possible de L , ni l'intension la plus petite en nombre d'itérateurs. Par exemple :

$$\begin{aligned} i_0 &= (1, 1, 3, +) \\ i_1 &= (4, 2, 3, +) \\ i_2 &= (16, 2, 2, \times) \\ i_3 &= (-1, -1, 2, +) \end{aligned}$$

est une autre intension possible. Notre algorithme recherche l'intension du segment le plus long possible avant de « changer d'itérateur ». Cette heuristique est très

simple à mettre en œuvre et montre de bons résultats sur les suites que nous avons étudiées, mais nous n'avons pas procédé à une évaluation par rapport à d'autres heuristiques possibles.

5.1.3.3 Construction des chroniques intensionnelles

La chronique intensionnelle $\kappa_t^{(int)}$ du terme t est construite à partir de la chronique compacte κ'_t en utilisant l'opérateur ξ sur les listes W et R de chaque action de κ'_t . La construction des itérateurs es effectuée de manière conjointe pour les deux listes, c'est-à-dire que la taille x des chaque itérateur en cours de construction est le plus grand nombre tel que :

$$\xi(W_{[0,x]}) \neq \surd \text{ et } \xi(R_{[0,x]}) \neq \surd$$

Lorsqu'un élément W_i ou R_i de l'une des listes « rompt » la régularité de la liste, l'itérateur courant est fermé pour les deux listes.

Exemple

Dans l'exemple du *Frigo*, en supposant que nous puissions traiter les suites en extension de la forme \ddagger dans ξ , nous obtenons la chronique $\kappa_{temp}^{(int)}(15)$ suivante :

who	what	why	when	res
\surd	\surd	\surd	0	3
heat up	put[...]	\top	(1, 1, 3, +)	(3.011, (0.999, 0.013), 3, \ddagger)
heat up	put[...]	\top	(4, 1, 3, +)	(3.453, (0.975, 0.494), 3, \ddagger)
cool down	put[...]	guard[...]	(6, 1, 2, +)	(4.060, (0.975, 0.294), 2, \ddagger)
heat up	put[...]	\top	7	4.070
heat up	put[...]	\top	(8, 1, 3, +)	(3.881, (0.999, 0.013), 3, \ddagger)
heat up	put[...]	\top	(11, 1, 5, +)	(4.300, (0.975, 0.494), 5, \ddagger)
cool down	put[...]	guard[...]	(11, 1, 5, +)	(4.100, (0.975, 0.294), 5, \ddagger)

Dans cette chronique, les valeurs du pas sont obtenues lors de la construction de la liste, sans connaissance préalable de la loi de température. Mais nous pouvons remarquer que :

$$\begin{cases} e^{-\frac{1}{1500}} \simeq 0.999 & \text{et } 20(1 - e^{-\frac{1}{1500}}) \simeq 0.013 \\ e^{-\frac{1}{40}} \simeq 0.975 & \text{et } 20(1 - e^{-\frac{1}{40}}) \simeq 0.494 \end{cases}$$

Remarque

En supposant que la porte du frigo reste ouverte, $\kappa_{temp}^{(int)}(1000)$ serait exactement la même chronique (ne contenant que 8 lignes), à la seule différence que le nombre 5 dans les itérateurs des deux dernières actions est remplacé par 989. De manière générale, la taille (en nombre de ligne) d'une chronique en intension donne une mesure de la régularité de l'activité du composant.

5.1.4 Extraction de comportement

La construction des chroniques en intension, et en particulier la construction des itérateurs dans ces chroniques, permet de mettre en évidence les changements d'activité au cours de l'exécution de la vue. En effet, pour une action donnée, chaque nouvel itérateur traduit une rupture de la régularité, c'est-à-dire un changement d'activité.

Nous proposons d'utiliser cette propriété pour munir les composants de capacité de raisonnement sur leur activité et transformer un composant simplement *réactif* en un agent *réflexif*, comme dans [53].

Par exemple, en utilisant les régularités qui apparaissent dans les chroniques en intension, le composant *Frigo* peut détecter la présence d'un nouveau *schéma comportemental* dans son environnement, comme nous le montrons sur la figure 5.1.

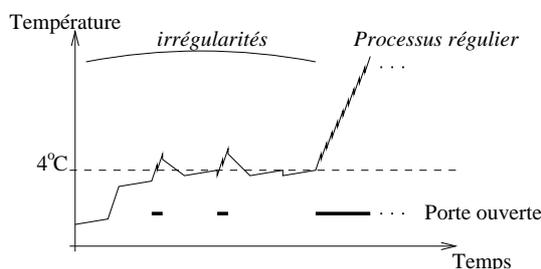


FIG. 5.1 – L'exécution du *Frigo* : faire ressortir les régularités

En supposant que ce processus régulier de réchauffement conduise à un état jugé néfaste (par exemple, une alarme est déclenchée si la température dépasse $8^{\circ}C$), le composant peut alors proposer une réaction à un *niveau sémantique supérieur*, qui consiste à rechercher l'action qui est à l'origine de cette rupture, plutôt que l'action réagir à l'action qui réchauffe. Il pourra ainsi décider de fermer automatiquement la porte. En conséquence, il devient un agent *réflexif*.

Une autre propriété importante de l'intensionnalisation est que le système n'a pas besoin de simuler n cycles d'exécution pour déterminer si une action de rupture donnée conduit au but recherché. La simulation sur 2 cycles de l'action *close* permet de :

- Arrêter le processus de réchauffement : les nouvelles actions effectuées sur *temp* ne seront pas ajoutées dans le schéma comportemental qui lui est associé ;
- Créer une nouvelle activité, qui se traduit par une action en intension dans la chronique :

$\langle \text{"cool down"}, put[...], guard[...], (t_0, 1, n, +), (x, (0.999, -0.987), n, \ddagger) \rangle$

- Résorber le problème de température dans le frigo.

En conséquence, un composant réactif simple peut être transformé en un agent réflexif simplement en ajoutant des chroniques, un mécanisme de détection de processus et une méthode d'inférence d'actions permettant d'utiliser quelques heuristiques

pour répondre à un niveau sémantique élevé à « ce qu'il est en train de se passer » dans le flux interactionnel.

5.1.5 Extraction de comportements collectifs dans un SMA

Les schémas comportementaux mis en évidence dans les chroniques intensionnelles modélisent non pas ce que le composant fait en termes d'actions élémentaires, mais bien ce qu'il est « en train de faire », au sens du présent progressif en anglais. La cause de ce schéma comportemental définit alors une *intention* de l'agent, au sens de [23]. Carle et Benhamou proposent avec CRS [15] d'utiliser la reconnaissance d'intentions dans les chroniques pour l'organisation automatique d'une société d'agents dans un SMA.²

Dans cette section, nous proposons d'étendre notre mécanisme d'intensionnalisation pour découvrir des comportements collectifs dans les SMA et pour caractériser l'organisation du système *a posteriori*, *i.e.* telle qu'elle est en train d'émerger. Dans cette problématique, un agent perçoit un flux d'interaction issus de n autres agents, dont le comportement peut alors être vu comme celui d'un collectif. A l'aide des chroniques intensionnelles, nous pouvons observer et mémoriser les interactions effectuées en situation ouverte dans l'environnement pour construire un modèle de comportement local (comme dans [37]). Notre objectif est alors de mettre en évidence des comportements collectifs à partir de l'intensionnalisation de ces comportements locaux.

5.1.5.1 L'exemple *CoolRoom*

Nous utiliserons le système suivant pour illustrer la synthèse d'organisation par rapport aux comportements collectifs dans les SMA.

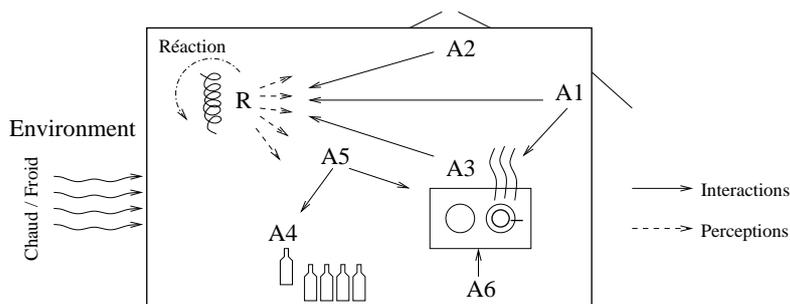


FIG. 5.2 – Interactions dans l'exemple *CoolRoom*.

Considérons le système représenté sur la figure 5.2. Il est composé de $n + 1$ agents : une pièce munie d'un radiateur (l'agent R) et n agents (A_1, \dots, A_n) essayant de survivre au froid dans la pièce en utilisant différents comportements. Lorsqu'il

²L'organisation collective est au cœur de la problématique des SMA (voir [20, 14]).

fait trop froid (par exemple, $\theta < 5^{\circ}C$), nous pouvons trouver les comportements suivants³ :

- L’agent A_1 ferme la porte de la pièce afin de limiter l’interaction avec le reste du bâtiment. Signalons que cette réponse est inadaptée lorsque la température est plus élevée dans le couloir que dans la pièce. L’agent A_1 n’a pas connaissance de ce point.
- L’agent A_2 ferme la fenêtre, pour les mêmes raisons (limiter l’interaction avec l’extérieur). Remarquons que le comportement des agents A_1 et A_2 porte sur la totalité du système : leurs actions peuvent résoudre le problème du froid pour tous les agents.
- L’agent A_3 fait une soupe chaude. Cette action a deux effets : elle réchauffe la pièce et elle offre de nouvelles ressources pour augmenter la résistance au froid des agents. Mais nous pourrions considérer que seuls certains agents peuvent utiliser cette ressource (par exemple, A_1 peut boire la soupe, mais pas A_2). De plus, nous nous trouvons alors dans une situation de compétition entre agents pour cette ressource.
- L’agent A_4 prend une bouteille de vin et la boit. Ce comportement n’est pas coopératif, mais nous pourrions imaginer que certains agents ont la capacité de voler la bouteille à l’agent A_4 .
- *et caetera.*

Nous ne donnons pas ici une description complète en VDL du système *CoolRoom* = $\{R, A_1, \dots, A_n\}$, mais il est facilement modélisable dans une vue où chaque agent correspond à un sous-terme différent. Dans l’optique d’un véritable SMA, nous pourrions envisager $n+1$ vues distinctes communicant par l’envoi d’événements externes. Cette problématique ne sera pas abordée dans cette thèse.

5.1.5.2 Principe général

Dans cet exemple, n agents agissent sur l’agent R afin de modifier sa température. Ils réagissent à l’événement $\theta < 5^{\circ}C$ de manière individuelle et ne coopèrent pas *a priori*⁴. C’est leur co-réaction qui engendre un schéma comportemental \mathcal{C} qui sera détecté comme tel par l’agent R ⁵ et auquel il répondra (par exemple en s’allumant), même si nulle part il n’y a *réification* (représentation *a priori*) de ce comportement (voir [62]).

Dans le modèle d’intensionnalisation de chroniques que nous avons proposé dans la section précédente, seuls les critères *when* et *res* sont intensionnalisés.⁶ Cependant, dans les SMA, un comportement peut émerger de l’interaction entre plu-

³Les agents *ne peuvent pas* ne pas faire ces actions car elles sont inscrites dans leur corps. Il s’agit donc d’agents réactifs.

⁴Ils ne disposent d’aucun modèle des autres agents (*cf.* [13]) : ils ne savent pas que les autres agents sont présents, qu’ils effectuent des actions et ils ne connaissent pas les causes éventuelles de ces actions.

⁵L’agent R n’a pas à postuler s’il s’agit d’une co-action ou d’une coopération : il constate simplement un comportement collectif.

⁶Dans notre exemple, c’est surtout l’intensionnalisation du critère *res* qui guide le processus.

sieurs agents. Par exemple, dans l'exemple *CoolRoom*, plusieurs agents $(A_i)_{i \in [1, n]}$ co-agissent pour résoudre un problème de température qu'ils partagent tous : toutes les actions sont effectuées *pour la même raison*. Nous allons proposer une méthode s'appuyant sur des algorithmes d'apprentissage pour détecter les critères communs des situations associées aux occurrences d'action.

5.1.5.3 Notion d'arbre de décision

Les arbres de décision sont utilisés en apprentissage automatique [72] pour catégoriser les données d'un tableau attribut-valeur : chaque nœud de l'arbre est étiqueté par un attribut et chaque branche par les valeurs pour cet attribut. Les feuilles de l'arbre sont les concepts reconnus par l'arbre de décision. Les arbres sont alors utilisés pour déterminer à quelle *classe* (*i.e.* à quel concept) appartient un *exemple* caractérisé par ses valeurs pour chacun des attributs.

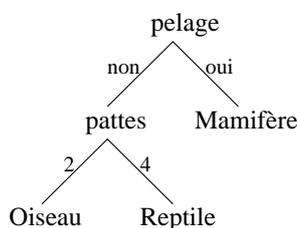


FIG. 5.3 – Un exemple simple d'arbre de décision.

Quinlan propose avec ID3 [106] un algorithme de construction d'arbres de décision minimaux, c'est-à-dire dans lequel chaque nœud est associé à l'attribut le plus discriminant parmi ceux encore non considérés dans le chemin à partir de la racine. L'algorithme ID3 utilise la notion d'entropie pour rechercher l'attribut le plus discriminant, calculée de la manière suivante : soit E l'ensemble des exemples considérés (*i.e.* les éléments du tableau attribut-valeur) et A un attribut ayant n valeurs possibles $v_1 \dots v_n$. Pour la valeur v_i , il y a V_i exemples dont $V_i^{C_j}$ pour le concept C_j . L'entropie résultant de l'utilisation de A comme critère discriminant sur E est alors :

$$Ent_{E,A} = \sum_{i=1}^n \frac{V_i}{|E|} \cdot \left(- \sum_{j=1}^p \frac{V_i^{C_j}}{V_i} \cdot \log_2 \left(\frac{V_i^{C_j}}{V_i} \right) \right)$$

L'algorithme C4.5 [107] est une extension de ID3 qui permet de prendre en compte dans l'apprentissage de l'arbre des exemples auxquels il manque certains attributs.

5.1.5.4 Application aux chroniques intensionnelles

Si nous considérons la chronique intensionnelle comme un tableau attribut-valeur, nous pouvons construire un arbre de décision dont les feuilles sont vides (notre objectif n'étant pas de reconnaître des classes). Cet arbre met en évidence

les critères les plus discriminants, c'est-à-dire ceux qui sont partagés par le plus d'actions.⁷

La figure 5.4 représente l'arbre de décision que nous obtenons pour $\kappa_{temp}^{(int)}(15)$ dans $\phi^{15}(Frigo)$. Remarquons à quel point la régularité des processus (base de la détection de comportement) conduit à un arbre de décision régulier.

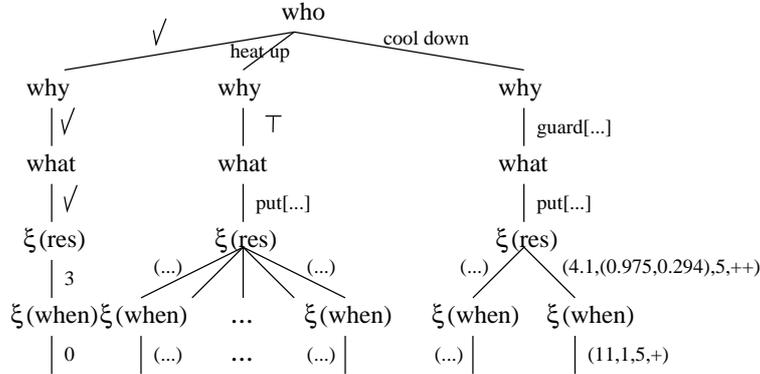


FIG. 5.4 – Arbre de décision pour $\kappa_{temp}^{(int)}(15)$ dans l'exemple *Frigo*.

Si nous considérons maintenant l'exemple *CoolRoom* faisant intervenir plusieurs agents en interactions avec l'agent *R*. La figure 5.5 donne un squelette de l'arbre de décision que nous pourrions obtenir lors de l'exécution de cet exemple. Le critère apparaissant en haut de l'arbre de décision, c'est-à-dire celui qui diminue le plus l'entropie, est le critère *why*. Il correspond à l'élément « le plus commun à plusieurs agents » dans le flot d'interaction concernant *R*. Dans cet exemple, les agents A_i effectuent tous des actions *pour la même raison*. Le mécanisme d'apprentissage permet de faire ressortir cette propriété.

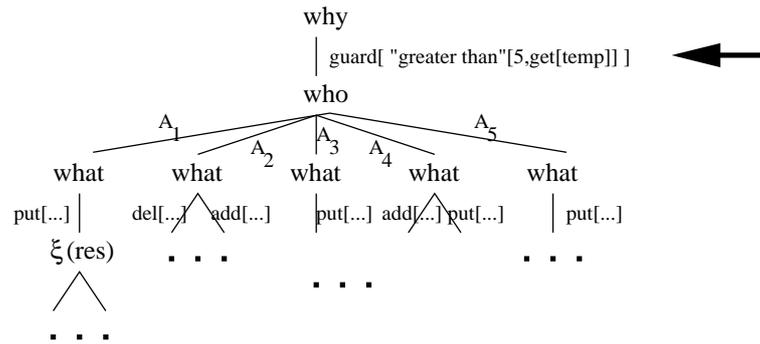


FIG. 5.5 – Squelette de l'arbre de décision pour *CoolRoom*.

⁷En cas d'égalité, nous prendrons l'ordre par défaut *who, why, what, res, when*, correspondant à ce que nous considérons comme étant le plus discriminant.

5.1.5.5 De la co-action à la coopération

Considérons le comportement des A_i du point de vue de la distinction classique entre co-action et coopération dans les SMA [13]. L'agent R détecte un comportement *co-actif*, puisque chaque agent A_i ne connaît pas les réactions des autres agents face au froid. Le comportement collectif des agents en réponse au froid n'a pas été défini *a priori* et par conséquent, il n'est pas possible de détecter un comportement coopératif, au sens de [35].

Cependant, dans notre exemple, l'agent R peut extraire ce que nous pouvons interpréter comme « l'intention commune des autres agents $(A_i)_{i \in [1, n]}$ », correspondant au critère qui est produit au plus haut dans l'arbre de décision. En utilisant alors des mécanismes d'inférence causale comme nous l'avons fait pour les chroniques en intension, l'agent R peut constater qu'il peut répondre à ce comportement de manière *globale*, par exemple en réchauffant la pièce, avec la conséquence que tous les A_i seront individuellement satisfaits.

Mais au-delà, l'agent R peut informer les A_i de ce comportement collectif, les rendant ainsi conscient⁸ du comportement des autres agents. En conséquence, le comportement co-actif devient coopératif.

5.1.5.6 Amélioration : apprentissage incrémental

Fisher et Schlimmer soulignent dans [43] qu'il est rare de disposer de tous les exemples en même temps. Ils introduisent alors la notion d'*apprentissage incrémental* avec ID4, qui modifie un arbre de décision lorsque de nouveaux exemples sont proposés. Utgoff propose avec ID5 [142] un algorithme pour construire de façon incrémentale des arbres de décision *minimaux*, c'est-à-dire, à partir d'un arbre de décision minimal d'un nouvel exemple, modifier l'arbre pour prendre en compte ce nouvel exemple tout en restant minimal. Chaque nœud de l'arbre mémorise seulement les $V_i^{C_j}$ utilisés pour choisir l'attribut discriminant et, dans les feuilles, les v_i des attributs inutilisés dans cette branche (lorsqu'il y en a). La mesure de l'entropie de chaque nœud est alors recalculée à l'aide du nouvel exemple et, si un critère devient plus discriminant que celui de son père, les deux nœuds sont inversés dans l'arbre, comme nous le montrons sur la figure 5.6.

ITI [143] est une amélioration de ID5 qui permet de traiter plus efficacement un plus grand nombre de cas.

Dans l'approche que nous avons présenté ici, l'intensionnalisation des chroniques et la construction des arbres de décision n'est pas incrémentale : le système mémorise la chronique extensionnelle et construit une intension à chaque cycle. Pourtant, l'algorithme de ξ peut facilement être rendu incrémental, ce qui permettrait de ne conserver que les chroniques en intensions pour l'analyse de l'activité des composants. À terme, nous pourrions construire l'arbre de décision en utilisant un algorithme comme ITI et travailler directement sur cet arbre de décision, sans conserver la chronique en intension $\kappa_t^{(int)}$ de chaque variable.

⁸Au sens de la connaissance, c'est-à-dire qu'ils en ont une représentation symbolique.

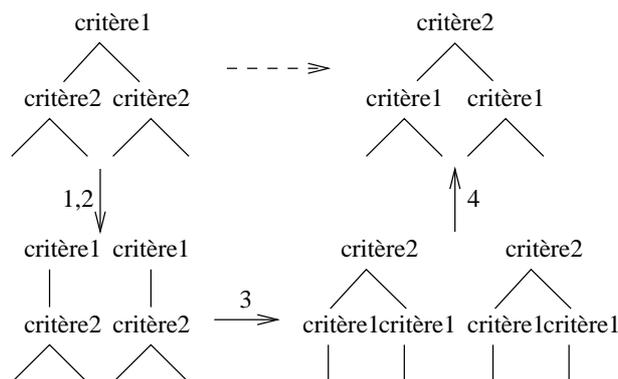


FIG. 5.6 – ID5 : comment produire un arbre optimum.

5.2 Utilisation dans le MRF

Les chroniques que nous avons présenté dans la section précédente permettent de mémoriser les occurrences d’actions et les situations dans lesquelles elles se produisent. Elles peuvent alors être utilisées par le MRF pour construire des réponses aux requêtes temporelles, c’est-à-dire telles que $\delta \neq \surd$ ou $\alpha = \textit{When}$, puisque l’ensemble des informations relatives les actions est mémorisé dans les chroniques.

De plus, nous avons montré que les chroniques en intension permettaient de mettre en évidence les ruptures dans l’activité du composant. Nous proposons dans cette section d’utiliser cette propriété pour répondre plus efficacement aux requêtes d’acte *Why*, modélisant des demandes d’explications.

5.2.1 Requêtes temporelles : $\delta \neq \surd$

Dans cette section, nous montrons comment il est possible d’utiliser les chroniques pour étendre les algorithmes de construction de réponses vus dans le cas $\delta = \surd$ aux requêtes datées ($\delta \neq \surd$). Soulignons que, même si la vue ne conserve que des chroniques intensionnelles, les chroniques en extension peuvent toujours être reconstituées à partir des chroniques en intension, par définition de la notion d’intension. Nous ne donnerons pas l’algorithme d’extensionnalisation, qui est obtenu de manière immédiate à partir de la définition des chroniques en intension.

Nous travaillerons donc indifféremment avec les chronique extensionnelles et les chroniques extensionnelles, selon le contexte.

5.2.1.1 Utilisation de la date

Dates sur le passé et le présent

Nous pouvons distinguer deux classes de requêtes datées, qui doivent être traitées de manière entièrement différente par le MRF :

- Les requêtes sur le passé, c'est-à-dire telles que $\delta = past$ ou δ correspond à une date en cycles depuis le début de l'exécution inférieure à la date courante θ ;
Pour répondre à ces requêtes, le MRF doit simplement rechercher dans les chroniques les informations relatives à l'état de la vue à la date modélisée par δ ;
- Les requêtes sur le futur, c'est-à-dire pour lesquelles le système doit extrapoler l'état du composant dans un nombre donné de cycles.
Nous montrerons que, pour répondre à ces requêtes, le MRF peut utiliser les itérateurs pour extrapoler les valeurs des variables.

Dans cette section, nous présentons à chaque fois un mécanisme d'utilisation des chroniques pour répondre aux requêtes sur le passé et sur le futur.

Deux classes de dates

Dans notre modèle, les dates $\delta \neq \surd$ peuvent prendre plusieurs valeurs (cf. section 3.2.7 page 106) :

- Des valeurs *définies*, comme $global[n]$, $date[z]$ ou $date[z, \rho]$;
Pour répondre à ces requêtes, le MRF doit rechercher dans les chroniques les actions et les situations correspondant à une date précise.
- Des valeurs *indéfinies* : *past* et *future*.
Pour répondre à ces requêtes, le MRF doit rechercher dans un ensemble de dates (fini pour le passé, infini pour le futur) celles qui sont associées à une situation particulière, définie par la requête datée.

La construction de réponse est donc très différente dans ces deux cas. Cependant, nous ramènerons le cas « indéfini » au cas « défini » par un algorithme semi-décidable⁹, en commençant à la date $\theta - 1$ (ou $\theta + 1$ pour le futur) et en parcourant à l'infini jusqu'à trouver une date qui convient.¹⁰ Nous travaillerons donc toujours sur une date donnée dans cette section.

Dans tous les cas, les réponses utiliseront des dates précises, données en valeur relative par rapport à la date courante (c'est-à-dire sous la forme $date[z]$).

Remarque : précision de la date

Notre modèle de date est fondé sur la notion de date exacte en nombre de cycles. Par conséquent, s'il convient pour un traitement au niveau formel et exact des requêtes temporelles, il est mal adapté pour la modélisation des dates *approximatives* en interaction homme-machine. En effet, quel utilisateur demanderait : « *Que faisais-tu il y a 123 cycles ?* ». Même en supposant que les utilisateurs donnent des dates

⁹Rappelons qu'un algorithme est décidable si et seulement s'il est modélisable par une machine de Turing qui termine en temps fini. Un algorithme semi-décidable (ou procédure de semi-décision) est un algorithme qui termine lorsqu'une solution existe, mais qui ne termine pas dans le cas contraire.

¹⁰Pour traiter le cas où la situation décrite dans la requête n'est pas atteignable, nous limiterons la recherche dans le futur à $2.\theta$. Nous n'aborderons pas ce point dans les algorithmes.

exactes, si la période des cycles est suffisamment petite, plusieurs cycles peuvent s'être écoulés simplement pendant le temps nécessaire pour poser la question.

Même si nous faisons abstraction des expressions langagières (comme « dans deux minutes » pour « tout de suite »), nous ne pouvons pas prendre en compte l'aspect approximatif des dates. Lorsque l'utilisateur demande « *Que faisais-tu il y a environ 30 secondes ?* », la date n'est jamais exactement $\delta = 30 \div T$ où T est la période d'un cycle. Ce problème ne peut malheureusement pas être traité en amont pas le MLN, puisqu'une propriété vraie à $30 \div T$ peut très bien être fausse juste avant ou juste après. La même difficulté est soulevée lorsque l'on considère les questions sur le présent (« *Que fais-tu maintenant ?* »), puisque la valeur de « maintenant » change, le temps de la construction de la réponse. Ce problème a été initialement soulevé et étudié par Ismail et Shapiro [65]. Ils montrent que les agents dialogiques doivent le prendre en compte dans la définition des connaissances qu'ils manipulent comme dans les mécanismes de raisonnement mis en œuvre dans le système. Ils proposent un cadre logique pour traiter ce problème, fondé sur la définition d'une fonction NOW pour le présent, le passé et le futur.

Dans cette thèse, nous n'avons pas abordé le problème du NOW et nous utilisons un modèle de recherche qui ne travaille jamais sur des dates approximatives. Nous supposons que lorsque l'utilisateur pose une question, la date formulée est exacte et précise, ce qui est une hypothèse trop forte pour l'interaction avec des utilisateurs ordinaires.

Cette heuristique s'appuie sur deux idées fortes dans le projet *InterViews* :

- Le MLN préférera généralement l'utilisation de *dates indéfinies* (*past* et *future*) dans les requêtes, pour traduire les notions imprécises des utilisateurs, associées à des expressions langagières ;
- L'état perçu d'un composant « reste stable » le temps de la perception, c'est-à-dire suffisamment longtemps pour que l'inexactitude de la date ne soit pas un problème.

Par exemple, dans une version intégrée de *Coco*, nous aurions un compteur interne, incrémenté à chaque cycle, toutes les 10 millisecondes ($T = 0.01 s$) et un compteur « externe », calculé à partir du compteur interne, qui affiche une valeur entière.¹¹ Lorsque l'utilisateur demande « *Quelle était ta valeur il y a 2 minutes ?* », nous pourrions supposer que la valeur entière est resté la même dans un intervalle « suffisamment grand » autour de $\theta - 2 \div T$.

C'est la raison pour laquelle nous travaillerons en date exacte. Dans des travaux futurs, nous envisageons de proposer un mécanisme de construction de réponse qui prend en compte l'aspect « flou » des dates pour rechercher « autour » de la date une valeur correspondant à celle demandée par l'utilisateur, par exemple en nous appuyant sur la fonction NOW définie par Ismail et Shapiro [65].

¹¹De la même manière que, dans une montre, le mouvement de la trotteuse n'est généralement pas continu alors que celui des rouages internes l'est.

5.2.1.2 Analyse du besoin

Les algorithmes du MRF utilisent cinq fonctions principales pour retrouver dans la vue les termes nécessaires à la construction de la réponse :

- La fonction *calcRef* qui recherche dans la vue l'ensemble des termes résultats d'une référence *r* ;

NB : La fonction $\chi_{concept}$ qui recherche dans un terme l'ensemble des sous-termes ayant un concept donné est simplement un sous-cas de la recherche de terme, définie par la fonction *calcRef*.

- L'interprétation canonique ζ utilisée pour calculer la valeur des termes représentant des opérations arithmétiques ;
- La fonction χ_a qui recherche dans un terme l'ensemble des sous-termes correspondant à une action donnée ;
- La fonction χ_p qui recherche l'ensemble des préconditions d'un terme ;
- La fonction χ_{evt} qui recherche les réactions principales à un événement externe donné.

Pour pouvoir répondre aux requêtes sur le passé et le futur, nous devons donc adapter ces fonctions de telle sorte qu'elles utilisent les chroniques en extension pour :

- Rechercher les situations permettent de retrouver les éléments recherchés dans l'exécution passée des vues ;
- Extrapoler des valeurs futures des termes pour répondre à des requêtes sur le futur.

Dans les sections suivantes, nous présentons les extensions de chacune de ses fonctions.

Remarque

Les algorithmes du MRF utilisent d'autres petites fonctions, spécifiques à un acte et à un type donné, dont certaines doivent aussi être adaptées pour le traitement des requêtes temporelles. Cependant, notre objectif n'est pas de présenter en détail des mécanismes de traitement des requêtes, mais d'illustrer une idée nouvelle sur les principaux points clef de la construction d'une réponse. C'est la raison pour laquelle nous étudierons ici uniquement les extensions de ces six fonctions principales.

5.2.1.3 Recherche de termes

Au cours de l'exécution du composant, certains termes peuvent être supprimés ou ajoutés dans la vue. Par exemple, dans *Jojo*, une figure être dans la main à un instant t_1 et dans la grille à un autre instant t_2 . Le terme de concept *figure* qui la modélise sera alors attribut de *taken* à t_1 et de *grid* à t_2 .

Arbre des modifications

Pour rechercher les termes dans la vue, nous proposons de mémoriser l'ensemble des termes faisant l'objet d'une modification dans un arbre de concepts construit de

la manière suivante :

- A l'instant 0 de l'exécution, l'arbre est réduit au terme de concept *view* ;
- Lorsqu'une action élémentaire est effectuée, l'ensemble des chemins des termes modifiés (c'est-à-dire référencés dans un terme de concept *path* parmi les attributs de l'action) est ajouté dans l'arbre. Nous notons $\mathcal{A}_\mu : \mathcal{P}(\Upsilon) \times \Upsilon \longrightarrow \Upsilon$ la fonction récursive qui prend en argument une référence et un arbre de concepts et construit l'arbre dans lequel les éléments de la référence ont été ajoutés.

Algorithm 7 fonction $\mathcal{A}_\mu(r \in \mathcal{P}(\Upsilon), t \in \Upsilon)$

```

si  $r \neq \emptyset$ 
  soit  $t_1, \dots, t_n = r$  et  $dest = \text{calcRef}(\{t_1\}, t)$ 
  si  $dest = \emptyset$ 
     $t_1' = \mathcal{A}_\mu(\{t_2, \dots, t_n\}, t_1)$ 
    ajouter_fils( $t, \{t_1'\}$ )
    retourner  $t$ 
  sinon
    pour  $i$  allant de 1 à  $|dest|$ 
       $t' = \mathcal{A}_\mu(\{t_2, \dots, t_n\}, dest[i])$ 
      remplacer( $dest[i], t'$ )
retourner  $t$ 

```

La fonction $\text{remplacer}(t \in \Upsilon, t' \in \Upsilon)$ remplace dans une vue un terme non flottant t par un autre terme t' . Cette fonction étant fortement dépendante de l'implémentation, elle ne sera pas détaillée ici.

Chaque nœud t de cet arbre est muni d'un sous-ensemble α_t de Act donnant l'ensemble des concepts des actions qui l'ont modifiés (seuls les nœuds terminaux des références ont un ensemble $\alpha_t \neq \emptyset$).

Soulignons que ce n'est pas parce qu'un élément dans l'arbre vérifie $del \in \alpha_t$ que les termes associés avec cette référence sont absent de la vue dans le présent. Ils peuvent en effet avoir été rajoutés après avoir été supprimés.

Enfin, les situations associées aux actions élémentaires (*add*, *put* et *del*) effectuées sur les termes sont mémorisées dans des chroniques κ_t attachées aux feuilles de \mathcal{A}_μ .

Remarque

La construction de l'arbre des modifications s'appuie sur l'hypothèse que les références dans les actions VDL de la vue sont soit toujours des références complètes (tous les nœuds du chemin sont donnés), soit toujours des références minimales (le moins de nœuds possible est donné). En effet, si $[a, c]$ et $[a, b, c]$ référencent le même terme et sont utilisés par deux actions différentes, si $[a, c]$ est inséré dans l'arbre des modifications en premier, la référence $[a, b, c]$ construira une nouvelle branche dans l'arbre.

Toutefois, cette particularité n'est pas un problème dès lors que la fonction calcRef utilise des chemins minimaux. Nous supposons donc dans la recherche

de termes que les références données sont minimales.

Recherche de termes : *calcRef*

Lorsque le MRF recherche le résultat d'une référence r dans une requête sur le passé, il parcourt l'arbre des modifications (en utilisant l'algorithme de *calcRef* pour le présent) à la recherche de cette référence. Nous notons alors α_r l'union des α_t des termes atteints :

$$\alpha_r = \bigcup_{t \in \text{calcRef}(r, \mathcal{A}_\mu)} \alpha_t$$

Si la référence r n'existe pas dans l'arbre, $\alpha_r = \emptyset$. Nous notons κ_r l'union classée par date des κ_t .¹²

Si $\alpha_r = \emptyset$, c'est que le terme lui-même n'a pas été modifié. Dans ce cas, pour construire une réponse, il suffit de travailler sur le temps présent. Par contre, si $\alpha_r \neq \emptyset$, le MRF recherche dans κ_r l'ensemble des actions associées à la plus grande des dates strictement inférieures à la date δ de la requête, ou θ si δ est une date dans le futur. Alors :

- Si la date δ de la requête est dans le passé, le résultat *res* de la dernière de ces actions donne les attributs du terme à la date δ . Le résultat de la recherche de termes est alors vue comme le terme de concept t et d'attributs *res* ;
- Si la date δ de la requête est dans le futur, le MRF utilise les itérateurs $\xi(R)$ et $\xi(W)$ de cette action (donné par la chronique en intension) pour extrapoler les attributs de t à la date δ :
 - Si $\xi(R)$ est réduit à une simple valeur v , le résultat de la recherche de termes est alors vue comme le terme de concept t et d'attributs v ;
 - Si l'extension W de $\xi(W)$ ¹³ donne une date finale $D < \theta$, c'est que la valeur n'a plus changé depuis. Le MRF considère alors que la valeur ne changera pas. Soit v la valeur finale de l'extension R de $\xi(R)$.
Le résultat de la recherche de termes est alors vue comme le terme de concept t et d'attributs v ;
 - Sinon, soit $(x_0, \Delta, n, op) = \xi(R)$ et $m = \delta - \theta$. Soit R' l'extension de $(x_0, \Delta, n + m, op)$ et v' la valeur finale de R' .
Le résultat de la recherche de termes est alors vue comme le terme de concept t et d'attributs v' .

Interprétation de termes : ζ

L'interprétation d'un terme t donné à une date δ ne présente alors plus aucune difficulté puisque l'algorithme de ζ est un simple parcours en profondeur du terme à

¹²C'est-à-dire que, lors de l'union, les situations sont interclassées dans κ_r de telle sorte à obtenir une chronique classée par date.

¹³L'extension W n'a théoriquement pas été conservée. Elle doit être reconstruite à partir de $\xi(W)$. Il en va de même pour l'extension R de $\xi(R)$.

interpréter. Lorsqu'un modifieur de référence (comme *get*) est rencontré, la fonction *calcRef* est utilisé sur les chroniques, comme nous l'avons présenté ci-dessus.

Cependant, pour les requêtes sur le passé, l'ensemble d'événements passés en argument à la fonction d'interprétation est l'ensemble E_δ des événements envoyés à la date δ :

$$E_\delta = \{e \text{ tq } (e, \delta) \in \Theta\}$$

L'algorithme vu pour les cas (*What, do*) et (*Ask, do*) doit alors être corrigé pour prendre en compte les événements envoyés par la vue. Ainsi, lorsque l'utilisateur demande par exemple « *Qu'as-tu fait il y a 2 minutes ?* », le MRF donne l'ensemble des actions (processus ou événements) effectués à la date proposée.

5.2.1.4 Recherche d'actions : χ_a

En VDLp, les composants peuvent modifier leur propre fonctionnement à l'aide d'actions élémentaires, puisque les actions sont aussi des termes. Mais compte tenu de la sémantique opérationnelle de ϕ_{exec} présentée au chapitre 2 et en particulier des ensembles $I_{calcRef}$ et I_ζ :

- Les actions ne peuvent pas être supprimées directement : seul le terme englobant d'une action (celui qui donne le *nom*) peut être supprimé ;
- La description des préconditions d'une action ne peut pas être modifiée directement ;
- Le corps des actions ajoutées est nécessairement décrit dans l'action d'ajout, comme nous l'avons vu dans l'exemple de modification du fonctionnement section 2.3.5.3 page 75.

Nous pourrions donc supposer qu'aucune action n'est jamais intégralement supprimée ou ajoutée dans la vue.¹⁴ Le MRF peut toujours retrouver les actions dans la vue en utilisant une extension l'algorithme χ_a vu dans le cas du présent (*cf.* section 4.3.1.1) dans laquelle la recherche continue lorsque des actions élémentaires sont atteintes :

```

si concept (t) ∈ {add, put, del}
  si (a=√) ou (a ≥ t), retourner {t}
  sinon, retourner  $\bigcup_{t' \in fils(t)} \chi_a(t', a)$ 

```

Cependant, dans le cas d'une action a décrites dans une action d'ajout a' , les préconditions de a' ne doivent pas être prises en considérations lors de la recherche des préconditions de a . Cela peut se faire simplement en utilisant dans les algorithmes

¹⁴Il est pourtant possible d'écrire un composant qui supprime une action puis qui supprime l'action de suppression. Par exemple :

```

view[ "une action"[...],
      détruire[ event[shoot],
                  del[path["une action"], path[détruire]]
            ]

```

L'exécution de cette vue sur, sur réception de l'événement *shoot*, conduit au terme atomique *view* dans lequel toutes les actions ont été supprimées. La fonction χ_a ne pourra alors plus trouver les actions recherchées, mais nous pouvons supposer que c'est conforme à l'intention du programmeur.

de recherche de préconditions un terme flottant copie de a , rattaché temporairement à la vue pour l'évaluation des préconditions, comme nous le montrerons dans la section suivante. Nous considérerons donc que l'algorithme de recherche d'actions χ_a étendu pour les requêtes temporelles retourne un booléen qui indique si l'action trouvée est présente dans la vue en tant que telle ou simplement en tant que « devant être ajoutée ».

Pour la recherche d'actions, le MRF travaille donc toujours sur le présent.

Recherche de réactions : χ_{evt}

Sur le même principe que la recherche d'action (χ_a), la recherche de réaction principale à un événement externe (par la fonction χ_{evt}) peut être directement utilisée pour les requêtes temporelles comme pour les requêtes sur le présent.

5.2.1.5 Recherche de préconditions : χ_p

Comme nous l'avons souligné dans la section précédente, les préconditions d'une action ne sont jamais modifiées directement. Par conséquent, si a est une action, l'ensemble Π_a de ses préconditions est obtenu simplement en remontant jusqu'à la racine du terme. Si l'action a a été obtenue par χ_a en recherchant dans une action élémentaire a_e (c'est-à-dire que l'action n'est pas interprétée dans la vue : c'est une action qui peut être ajoutée), l'algorithme travaille sur une copie flottant de a , c'est-à-dire dont la racine est le terme attribut de l'action a_e .

Par exemple, si nous considérons l'action élémentaire :

```
add[ path[view], count[put[path[x], plus[get[x], 1]]] ]
```

$\chi_a(vue, put[path[x]])$ retourne le terme de concept *put* et définit la racine comme étant le terme de concept *count*. La recherche de préconditions sur l'action *put* « s'arrête » donc aux préconditions décrites dans l'action *count*.

L'objectif de $\chi_{precond}$ est alors de déterminer la valeur de vérité de chaque élément de Π_a pour construire les ensembles Π_{\top} , Π_{\perp} pour un ensemble d'événements externes E envoyés à la vue à la date δ – l'ensemble \mathcal{E} est obtenu directement comme pour le présent :

$$\mathcal{E} = \bigcup_{e \in \Pi_a} \{ fils(e) \text{ tq } concept(e) = event \}$$

Du point de vue théorique, la plus simple consiste à interpréter chaque précondition en utilisant les mécanismes présentés ci-avant. Cette solution, simple à implémenter, est malheureusement trop coûteuse puisqu'il faut rechercher dans la chronique de chaque variable intervenant dans les gardes booléennes. Nous envisageons dans des travaux futurs d'utiliser un mécanisme spécifique de mémorisation des valeurs des préconditions, inspiré des chroniques.

5.2.2 Requêtes d'acte $\alpha = When$

Les requêtes d'acte $\alpha = When$ sont utilisées pour modéliser les questions de contrôle dans lesquelles l'utilisateur au système à quelle date dans le passé ou dans le futur la vue était (sera) dans un état donné, pouvait (pourra) effectuer ou avait (aura) effectué une action ou réagir à un événement externe.

Le sujet σ de la requête ne peut jamais être vide et il doit toujours référencer un seul terme de la vue (sinon le MRF retourne une erreur); nous notons alors *sujet* le terme référencé. L'objet ω de la requête ne peut pas non plus être vide puisque c'est lui qui définit l'hypothèse formulée par l'utilisateur. Enfin, pour les requêtes *When*, la date peut prendre les valeurs *past* ou *future*, pour indiquer au MRF s'il doit rechercher dans l'exécution passée de la vue ou extrapoler une exécution future.

La construction de la réponse est basée sur des mécanismes similaires à ceux que nous venons présenter, fondés sur la recherche dans des chroniques de dates. Le principe général est une recherche en utilisant une procédure de semi-décision commençant à la date présente.

5.2.2.1 Requêtes de type $\tau = is$

Les requêtes de la forme $(\alpha, \tau) = (When, is)$ modélisent des demandes date associée à l'état du composant.

Demande de valeur : $\sigma \neq \emptyset$

Lorsque $\sigma \neq \emptyset$, les requêtes $(When, is)$ modélisent des demandes de recherche de la dernière date (dans le passé) ou de la première date (dans le futur) à laquelle une variable donnée prendra l'une des valeurs proposées dans ω .

Si aucune chronique est associée à *sujet*, deux situations sont possibles :

- Si $fils(sujet) \subseteq \omega$, la réponse est $\langle Assert, is, \sigma, fils(sujet), \top, \delta \rangle$;
- Sinon, la réponse est $\langle Assert, is, \sigma, \omega, \perp, \delta \rangle$.

Si une chronique est attachée à *sujet*, la réponse est construite simplement comme pour la recherche d'attributs d'un terme :

- Si $\delta = past$, soit $R = \{r_1, \dots, r_n\}$ l'extension de $\xi(R)$ et $i \in [1, n]$ le plus grand indice tel que $r_i \in \omega$. Si i est indéfini, la réponse est $\langle Assert, is, \sigma, \omega, \perp, past \rangle$; sinon, la réponse est :

$$\langle Assert, is, \sigma, r_i, \top, w_i \rangle$$

avec $\{w_1, \dots, w_n\}$ l'extension de $\xi(W)$;

- Si $\delta = future$:
 - Si l'extension W de $\xi(W)$ donne une date finale $D < \theta$, (la valeur n'est pas modifiées par un processus en ce moment), la réponse est exactement la même que lorsqu'aucune chronique n'est attachée à *sujet*.
 - Sinon, soit $(x_0, \Delta, n, op) = \xi(R)$. Pour m allant de 1 à ∞ ¹⁵, soit R' l'extension de $(x_0, \Delta, n + m, op)$ et r la valeur finale de R' : si $r \in \omega$, alors

¹⁵Dans l'implémentation du MRF, nous limiterons la recherche à $m = \theta$.

nous notons $(x'_0, \Delta', n, op') = \xi(W)$ et w la valeur finale de l'extension de $(x'_0, \Delta', n + m, op')$. La réponse est alors :

$$\langle Assert, is, \sigma, r, \top, w \rangle$$

Relations : $\sigma = \emptyset$

Lorsque $\sigma = \emptyset$, les requêtes $(When, is)$ modélisent des demandes de recherche de la date à laquelle l'une des relations proposées dans ω était (ou sera) vraie.

- Si il existe $\omega_i \in \omega$ tel que $concept(\omega_i) \in \mathcal{G}_{rp}$, puisque les actions ne peuvent pas être modifiées, la relation ω_i est vraie dans le passé ou dans le futur si et seulement si elle est vrai « au présent ». L'algorithme utilisé est alors celui présenté pour le cas (Ask, is) au chapitre précédent, dans lequel la date de la réponse est remplacée par δ .
- Sinon, il faut évaluer toutes les relations dans le passé ou dans le futur, en utilisant un mécanisme d'extensionnalisation de chroniques similaire à celui vu dans le cas $\sigma \neq \emptyset$, pour toutes les références de $\chi_{concept}(\omega_i, get)$, pour tous les ω_i .

L'algorithme s'arrête dès que la valeur de vérité d'une relation ω_i est égale à ν . Comme pour le cas (Ask, is) , nous privilégions donc les réponses positives sur les réponses négatives. En particulier, si dès le début, une relation ω_i est telle que toutes les références utilisées ($\chi_{concept}(\omega_i, get)$) ne sont pas munies de chroniques et si la valeur de vérité de la relation dans le présent est égale à ν , la réponse (positive) est immédiatement retournée avec cette relation dans l'objet \bar{w} .

Réciproquement, si toutes les références de toutes les relations ne sont pas munies de chroniques et si la valeur de vérité de toutes ces relations est égale à $\neg\nu$, l'algorithme retourne tout de suite une réponse négative.

Nous ne donnerons pas l'algorithme en détail ici : son énoncé n'apporte pas d'information pertinente pour sa compréhension et ne présente aucune originalité par rapport aux fonctions présentées ci-avant.

5.2.2.2 Requêtes de type $\tau = do$

Les requêtes de la forme $(\alpha, \tau) = (When, do)$ modélisent des demandes date associée à l'activité du composant.

Toutes les actions (soit en tant que ω_i , soit en tant qu'attribut d'une relation procédurale, lorsque $concept(\omega_i) \in \mathcal{G}_{rp}$) sont recherchées dans la vue au temps présent, comme nous l'avons expliqué pour χ_a . Les préconditions sont ensuite évaluées comme des relations statiques, suivant le mécanisme vu pour les requêtes de type is , fondé sur utilisant un algorithme de recherche dans les chroniques en extension en partant de la date présente. Pour les relations procédurales, l'interprétation canonique de la relations de contrainte (dans le passé ou le futur, selon δ) est calculée en utilisant le mécanisme proposé ci-avant.

Comme pour les requêtes d'acte *Ask*, nous construisons plusieurs ensembles et nous privilégions les requêtes positives. Dans le cas $\tau = do$, le MRF recherche *exclusivement des processus*, actifs lorsque $\nu = \top$, inactifs sinon.

L'algorithme ne sera pas détaillé ici.

5.2.2.3 Requêtes de type $\tau = can$

Les requêtes de la forme $(\alpha, \tau) = (When, can)$ modélisent des demandes date associée au comportement possible du composant. L'algorithme utilisé est exactement le même que pour les requêtes de type *do* mais le MRF recherche toutes les actions possibles (processus ou réactions), actives lorsque $\nu = \top$, inactives sinon.

5.2.2.4 Requêtes de type $\tau = order$

Les requêtes de la forme $(\alpha, \tau) = (When, order)$ modélisent des demandes date associée à l'envoi d'un événement externe. Si $\delta = future$, la réponse est une requête d'acte *Error* :

$$\langle Error, is, \{ "order in future" \}, \emptyset, \top, \surd \rangle$$

Sinon, la réponse est construite exactement comme dans le cas $(\alpha, \tau) = (Ask, order)$ vu au chapitre précédent.

5.2.3 Requêtes d'acte $\alpha = Why$

5.2.3.1 Un problème de ramification dynamique

Le problème des ramifications est généralement abordé dans la littérature uniquement du point de vue de l'analyse *statique* des actions, c'est-à-dire en considérant les effets d'actions qui changent la valeur de vérité d'une précondition d'une autre action.

L'exemple du *Frijo* que nous avons présenté dans la section précédente (*cf.* figure 5.1 page 214) montre qu'un processus peut émerger de manière dynamique, à partir d'un changement non pas dans les préconditions de l'action mais dans les valeurs intervenant dans le corps d'une action. Dans notre exemple, le processus de réchauffement n'a pas de cause directe en dehors de la valeur trop élevée du facteur *T*. Ce processus n'est même par modélisé *a priori* dans le composant.

Il existe donc des situations où un processus apparaît de manière dynamique, à partir de la modification non pas d'une variable de précondition, mais d'une variable utilisée dans le corps de l'action.

L'utilisateur peut alors désigner ces processus par leur effet, correspondant à des actions dans la vue, alors qu'il n'est pas matérialisé en tant que tel dans le composant. Le MRF ne peut alors pas expliquer ce processus émergent par ses préconditions, puisqu'elles ne sont pas la cause de cet émergence.

Exemple

Pour illustrer ce problème dans la construction d'explications, considérons la question : « *Pourquoi est-ce que la température augmente ?* » qui conduit à l'ensemble d'explications $why_r(\rho) = \{true\}$, puisque l'action de la vue correspondant à $increase[path[temp]]$ n'a aucune précondition (donc $\Pi_{\top}^{(a)} = \{true\}$). Cette relation ne peut pas permettre de construire une explication pertinente. Pourtant, le magicien d'Oz répondrait : « *Parce que la porte est ouverte* ». En effet, la « cause » de l'action *heat up* est l'action *open*, dont un effet (indirect) est non pas d'activer la précondition de l'action *heat up* mais de modifier l'activité du composant de manière significative.

Alors que dans le cas de *Coco*, l'utilisation de la précondition du processus *count* permet de construire simplement une réponse convenable, dans le cas du *Frijo*, la précondition d'un processus similaire (*heat up*) ne donne aucune réponse pertinente, puisque ce n'est pas de la précondition que le problème vient, mais de la modification d'un des éléments de l'action.

Pour répondre à ces questions, le MRF doit donc :

1. Détecter que la question porte sur un processus émergent ;
2. Construire une réponse à partir de l'analyse du corps de l'action.

Nous n'aborderons pas ces deux problèmes dans cette thèse. Nous proposons simplement dans la section suivante une solution pour construire des explications pour des processus non gardés. Dans des travaux futurs, nous envisageons d'utiliser les chroniques en intension et l'extraction de comportements pour détecter les processus émergents, comme nous l'avons proposé dans la section 5.1.4.

5.2.3.2 Utilisation des chroniques

Pour construire des réponses à ces questions, nous proposons de détecter dans les chroniques les changements d'activité de la vue puis d'expliquer ces changements par l'ensemble des actions effectuées au cycle précédent. L'heuristique que nous utilisons alors est que tout changement d'activité (correspondant à l'exécution des processus) est l'effet d'un changement de valeur de variable (qui modifie la précondition ou la variable utilisée dans le corps du processus) en réaction à une action.

Principe général de l'algorithme

Considérons une requête ρ , pour laquelle le MRF a construit trois ensembles $why_{x \in \{e,r,a\}}(\rho)$: si $why_r(\rho) = \{true\}$, la requête ρ ne peut pas être expliquée par cette seule relation. Soulignons que ce cas se produit uniquement lors de l'explication d'un terme t décrivant un *processus non gardés*.

Nous recherchons alors l'ensemble P des références intervenant dans l'action, à l'exception des objets de l'action :

$$R = \chi_{concept}(t, get) \setminus \chi_{concept}(t, path)$$

Pour chaque référence $files(r)$, $r \in R$, nous recherchons dans la chronique associée les itérateurs incluant la date δ de la requête. Si $\delta = \surd$, nous prenons en considération les derniers itérateurs. Soit A l'ensemble des actions associées à ces itérateurs. Alors $why_a^r(t) = A$.

Exemples

Considérons la question « *Pourquoi tu réchauffe ?* » dans le *Frigo*, modélisée par la requête :

$$\langle Why, do, \{view\}, \{ "heat up" \}, \top, \surd \rangle$$

L'explication construite par le MRF suivant l'algorithme proposé au chapitre précédent sera :

$$why_r(\rho) = why_r^r("heat up" [...]) = \{true\}$$

Nous construisons alors l'ensemble R des éléments intervenant dans l'action *heat up* :

$$R = \{get[T]\}$$

Dans la chronique associée à T , le dernier itérateur donne l'action *open* (qui a changé la valeur de T). Nous aurons alors :

$$why_a(\rho) = \{open, \top\}$$

5.3 Bilan

Nous avons présenté dans ce chapitre un mécanisme de mémorisation de l'exécution des vues en intension, fondé sur la modélisation des situations associées aux occurrences d'actions. Notre travail permet de répondre aux requêtes sur le passé de la vue, en utilisant l'essentiel des algorithmes vus dans le cas du présent. Nous n'avons pas présenté en détail les algorithmes de construction de réponses à ces requêtes temporelles, mais nous nous sommes attachés à mettre en évidence le principe général de l'utilisation des chroniques.

Nous avons aussi montré que l'intensionnalisation de chronique peut être utilisée pour extraire des schémas comportementaux de manière automatique à partir des flux d'interactions entre un agent et un utilisateur ou, plus généralement, entre un agent et un collectif de N autres agents co-actifs (mais non coopératifs).

L'algorithme d'intensionnalisation des critères *res* et *when* que nous avons proposé est cependant limitée aux listes numériques simples. Nous voudrions à terme incrémenter le nombre d'opérateurs que la fonction ξ peut traiter ainsi que le degré d'intensionnalisation, afin de détecter des suites plus complexes. Nous voudrions aussi pouvoir utiliser plusieurs itérateurs simultanément pour une même liste pour pouvoir l'intensionnaliser lorsqu'elle est irrégulière mais composée de segments réguliers alternants (ce qui est fréquemment le cas dans les interactions dans un SMA).

Dans notre modèle, la représentation des comportements des agents (correspondant aux régularités dans les chroniques d'exécution de chaque agent) peut être

utilisé par des algorithmes d'apprentissage pour extraire des critères de situation communs et donc, des similitudes de fonctionnalité entre les agents, à l'aide d'algorithmes d'apprentissage d'arbres de décision. Notre objectif est, à terme, d'étendre le mécanisme d'extraction de comportements collectifs pour l'analyse de flux interactionnels entre plusieurs *groupes* d'agents (N vers N) afin de faire émerger des organisations dans les SMA. Chaque permutation de la liste des critères dans l'arbre de décision correspond à une perception possible de l'organisation du SMA.

Nous pensons que le mécanisme de mémorisation sous la forme de chroniques en intension devrait aussi être utilisé dans le MLN pour mémoriser les informations spatiales, nécessaires au traitement de la référence (par exemple pour répondre à la question : « *Où était le cube vert avant ?* »). De manière générale, l'intégration du raisonnement sur le fonctionnement et du raisonnement sur la visualisation des éléments dans l'interface semble une étape importante vers la construction de réponses pertinentes aux questions des utilisateurs.

Enfin, nous envisageons dans des travaux futurs d'utiliser les chroniques en intension pour modéliser les actions et les relations procédurales du *Ground* qui traduisent aussi des schéma comportementaux. Par exemple, *increase* correspond à un itérateur (x_0, Δ, n, op) avec $\Delta > 0$ si $op = +$ et $\Delta > 1$ si $op = \times$. De même, *backwards* correspond à un itérateur (x_0, Δ, n, op) avec $\Delta < 0$ si $op = +$. Dans cette thèse, nous n'avons pas pu pousser cette étude des différentes modélisations possibles de chaque élément du *Ground*, en raison de la complexité des situations possibles et de la trop faible capacité d'intensionnalisation de l'opérateur ξ . Cependant, nous pensons utiliser les travaux en découverte de lois [97, 36] pour étendre cet opérateur et représenter les éléments du *Ground* de manière *dynamique* et non *statique* comme nous l'avons proposé initialement.

Chapitre 6

Application au web sémantique

6.1 Analyse de la problématique

Dans les chapitres précédents, nous avons proposé un modèle de représentation, de requêtes et de raisonnement sur le fonctionnement pour les composants actifs en interaction avec un utilisateur humain. Récemment, Internet est devenu le lieu de l'interaction entre des utilisateurs ordinaires et des composants actifs. Nous proposons dans ce chapitre d'utiliser notre travail dans le cadre nouveau du *Web Sémantique*. Nous montrons qu'il est possible et nécessaire d'adapter le langage VDLp pour décrire en XML des composants intégrant une structure sémantisée riche et la description d'un fonctionnement.

6.1.1 Les composants actifs dans le web sémantique

6.1.1.1 Le web sémantique aujourd'hui

Le web sémantique se veut une extension du web actuel dans laquelle l'information est munie d'une sémantique bien définie [11], afin de permettre aux agents logiciels de procéder à des traitements automatiques portant sur le contenu des pages et sur la *signification* de ce contenu [84]. Ce nouveau web permettrait de proposer aux utilisateurs des services de meilleure qualité autour de la manipulation de l'information (extraction et présentation de l'information, activation automatique de services, *etc*).

Les recherches actuelles sur le web sémantique proposent de s'appuyer sur des techniques de représentation des connaissances [21] pour munir l'information contenues dans les pages web d'une *sémantique*. Grâce à l'émergence de standards comme DAML-OIL comme support de la représentation des connaissances *structurelles* sur l'information contenue dans les pages web, ces travaux permettent déjà de disposer de services plus aboutis dans le domaine de la recherche d'informations : requêtes sémantisées [6, 135], indexation de documents selon la sémantique de l'information contenue [28, 29], guidage de l'utilisateur dans sa recherche [113], *etc*.

6.1.1.2 Composants actifs et web sémantique

Cependant, les utilisateurs ordinaires dans le web n'accèdent pas directement à cette information. Au contraire, ils interagissent avec des composants actifs (services, agents, *etc*) comme ceux que nous essayons de définir dans le projet *InterViews*, c'est-à-dire des composants logiciels qui manipulent cette information en fonction de l'interaction avec l'utilisateur. Ces composants actifs sont caractérisés par un *fonctionnement* [53] et, contrairement au « script CGI » qui ne fait que traiter, manipuler ou produire l'information contenue sur le web, ce fonctionnement fait *partie intégrante* de l'information.

Le web sémantique doit donc permettre de représenter et de manipuler des connaissances non seulement structurelles, mais aussi de nature *fonctionnelle*, c'est-à-dire de représenter le fonctionnement des composants actifs qui y agissent. Plusieurs modèles de description de services web ont été proposés, comme WSDL [22], WSMF [39] ou DAML-S [26], mais dans tous ces modèles, la description du service est séparée de l'information manipulée. Au contraire, dans un composant actif, la structure et le fonctionnement doivent être intégrés au sein d'une même représentation, pour lui permettre d'interagir avec l'utilisateur aussi bien à propos de l'information manipulée que des opérations de manipulation. Il faut donc pouvoir *programmer* directement en XML les composants actifs.

Le langage VDLp que nous avons présenté au chapitre 2 permet d'accéder *en cours d'exécution* à une description *intégrant au sein d'une même représentation* la partie structurelle et la partie opérationnelle (les actions des composants). Compte tenu d'une part de la syntaxe arborescente choisie, qui peut facilement être transposée en XML, et d'autre part de la sémantique opérationnelle adoptée, fondée sur la réécriture d'arbres, ce modèle peut servir de support pour la description de composants actifs dans le web sémantique en XML, et pour munir les pages web statiques sémantisées d'un fonctionnement.

Dans ce chapitre, nous présentons un modèle d'activité et un langage de programmation de composants actifs en XML pour le web sémantique, adapté de VDLp, dans lequel la structure et le fonctionnement sont intégrés au sein d'une même représentation, en vue de pouvoir répondre à la fois à des requêtes sémantisées sur la structure [6] et sur le fonctionnement (comme nous l'avons montré dans les chapitres précédents).

Nous présentons tout d'abord le principe général de fonctionnement des composants dans ce modèle, tant sur le plan structurel que sur le plan fonctionnel, à partir des observateurs. Nous décrivons ensuite la partie structurelle et nous montrons comment elle peut être étendue pour prendre en compte des modèles sémantiques riches étendus, implémentés en XML. Nous décrivons la partie fonctionnelle des composants actifs, en donnant la syntaxe du langage, la sémantique opérationnelle. Nous illustrons les possibilités de notre modèle sur un exemple simple mais complet. Enfin, nous situons nos travaux par rapport aux approches similaires proposées dans la littérature.

6.1.2 Notion d'observateur en XML

6.1.2.1 Principe général

Le modèle d'exécution que nous proposons pour les composants actifs décrits en XML s'appuie sur la notion d'*observateur* présentée au chapitre 2. Les observateurs recherchent dans un document XML des éléments¹ spécifiques, conformes à un schéma de description ou à une DTD, et les interprète pour effectuer des actions particulières. Chaque observateur est donc caractérisé par :

- Une DTD, méta-DTD ou un schéma auquel le document doit se conformer pour pouvoir être interprété par cet observateur. Elle décrit l'ensemble des éléments qui sont connus de cet observateur (ceux qu'il va interpréter) et la *syntaxe* à respecter ;
- Une fonction d'interprétation ϕ pour les éléments définis dans ce schéma, implémentée dans un langage de programmation utilisant l'API « *DOM* » [148]. Elle définit les actions effectuées sur les éléments reconnus par l'observateur, c'est-à-dire la *sémantique* de ces éléments.

Nous pouvons alors séparer les observateurs en deux classes :

1. **Les observateurs statiques**, qui ne modifient pas le document de manière proactive. Par exemple :
 - l'observateur structurel qui répond aux requêtes sémantisées portant sur les informations contenues dans le composant ;
 - l'observateur d'interface qui gère l'interface graphique utilisateur du composant (modifiée par effet de bord).
2. **L'observateur dynamique** qui considère le document passée en paramètre comme la description du composant à l'instant t et qui construit sa description à l'instant suivant $t + 1$ en fonction des éléments spécifiques décrivant le fonctionnement du composant actif.

L'exécution d'un composant actif dans ce schéma consiste donc en la réécriture du document XML. La description du composant en XML est sérialisée à chaque cycle d'exécution et chaque observateur (l'ensemble des observateurs statiques puis à nouveau l'observateur dynamique) interprète le nouveau document, comme le montre la figure 6.1.

6.1.2.2 Modèle de description d'un observateur

En VDLp, les observateurs sont robustes : si un élément recherché n'est pas trouvé dans un document ou si le document n'est pas conforme à la syntaxe de l'observateur, alors aucune action n'est effectuée par cet observateur. Par conséquent, la description d'un composant actif en XML doit être prévue *ab initio* pour être interprétée par plusieurs observateurs différents : au moins l'observateur statique

¹Dans ce chapitre, le terme « élément » est employé uniquement pour désigner les éléments XML, correspondant aux « termes » du langage VDLp.

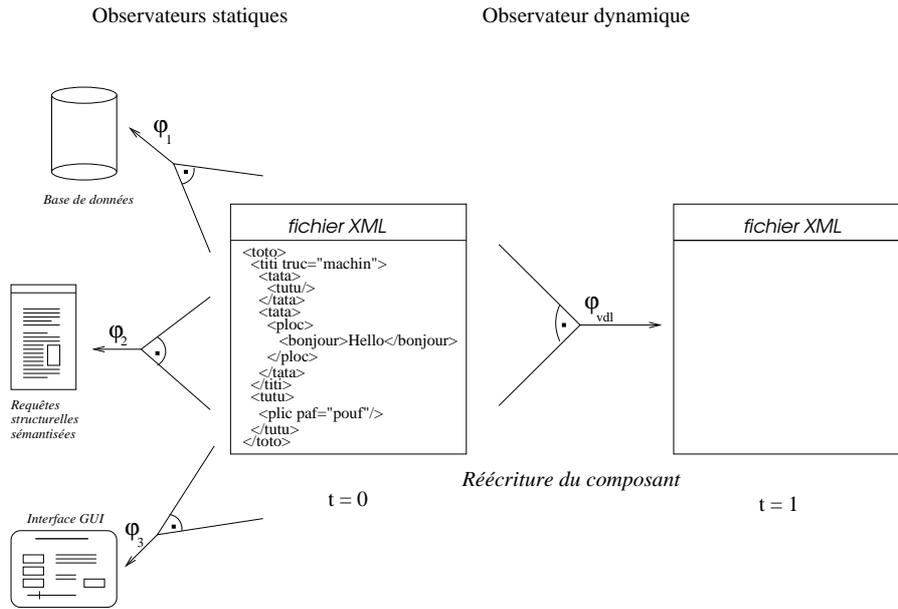


FIG. 6.1 – Modèle d’activité des composants actifs en XML.

d’interface (correspondant à l’interprétation du document dans un cadre statique) et l’observateur d’exécution. De plus le document devrait pouvoir être interprété par des observateurs non prévus initialement.

Dans ce cadre, les documents dynamiques ne peuvent pas se conformer à une seule DTD (ou alors c’est une DTD qui englobe tous les observateurs, ce qui est contraire à l’idée d’extensibilité pour de nouveaux observateurs). Ils ne peuvent donc être valides que pour une méta-DTD ou un schéma.²

6.1.2.3 Observateur *exec*

Dans ce chapitre, nous présentons la syntaxe de l’observateur *exec* dans le modèle XML. Nous avons choisi de la définir par un schéma plutôt que par une méta-DTD parce que cela nous permet de préciser le type des contenus. De plus, la partie arithmétique de notre schéma pourrait être étendue pour tenir compte des types primitifs (ce qui n’est pas le cas dans la version présentée ici).

Dans les sections suivantes, nous décrivons en détail la syntaxe (définie par le schéma) et la sémantique (définie par les algorithmes d’interprétation) de cet observateur. Le schéma complet (*vdl.xsd*) est donné en annexe page A-6.

²Toutefois, il est possible de rendre dynamique un document statique existant, défini par rapport à une DTD, en déclarant cette DTD comme une meta-DTD dans le document :

```
<DOCTYPE toto[<?IS10744:arch name="monNom" dtd-system-id="maDTD.dtd"
doc-elem-form="maDTD" form-att="maDTD"?>>
```

6.2 Présentation du modèle

6.2.1 Partie structurelle

6.2.1.1 Principe général

Un composant actif est décrit dans notre modèle par un arbre XML de racine *view*, conformément au schéma *vdl.xsd* :

```
<xsd:element name="view">
  <xsd:complexType content="elementOnly"/>
</xsd:element>
```

Le contenu d'un élément *view* n'est pas précisé, pour permettre l'utilisation d'autres éléments que ceux définis dans le schéma de l'observateur d'exécution, mais c'est nécessairement un arbre.

6.2.1.2 Notion de valeur

Les « valeurs » des variables modifiées sont, par définition, les données contenues dans les éléments (c'est-à-dire ce qui est entre les balises), conformément aux recommandations du W3C. Au contraire, les noms de balises, les attributs et les valeurs d'attributs sont du ressort de la définition de la structure et ne peuvent pas être modifiées (de même qu'un programme ne change pas le type d'une variable en C). Les actions élémentaires de modification du document XML ne travailleront donc qu'au niveau des contenus.

Dans notre modèle, pour des raisons de simplification, nous ne prenons pas en compte le type des contenus manipulés. Pour donner des valeurs dans la description du fonctionnement, nous les englobons dans des éléments **value** :

```
<xsd:element name="value" type="xsd:anyType"/>
```

Le contenu d'un élément *value* n'est donc pas précisé, pour permettre l'utilisation soit directement de valeurs (sérialisées par un contenu de type *string*), soit d'éléments structurés définis dans un schéma spécifique au composant. Notre langage de programmation est donc *non typé*. Comme nous le verrons, il peut être étendu pour prendre en compte les types primitifs des schémas et même les types plus riches définis dans les schémas structurels spécifiques à chaque composant actif.

6.2.1.3 Intégration de la structure et du fonctionnement

Les éléments *view* et *value* peuvent contenir des éléments quelconques, en particulier des éléments qui ne sont pas définis dans le schéma *vdl.xsd*. Cela permet d'inclure des éléments munis d'une structure riche, définie dans un autre schéma, et de les manipuler en utilisant les actions élémentaires VDL.

Dans le web sémantique, les composants manipulent l'information structurée et sémantisée, utilisant des représentations complexes [29]. Notre modèle permet alors d'intégrer la description du fonctionnement au sein de n'importe lequel de ces documents structuré pour en faire un composant actif, d'une part en englobant l'ensemble du document dans l'élément *view* et d'autre part en englobant les éléments modifiables (ceux manipulés par le composant) dans ces éléments *value*.

De plus, le fonctionnement étant décrit dans la même représentation que la structure, il est lui-même manipulable, comme dans les langages de programmation orientée Intelligence Artificielle comme LISP ou Prolog.

6.2.2 Partie fonctionnelle

6.2.2.1 Références

Pour faire référence aux éléments afin de les modifier ou d'en extraire leur valeur, nous proposons d'utiliser la syntaxe *XPath* [146] définie dans la norme XML. Comme en VDL, les éléments permettant de faire des références dans notre langage sont **path** et **get**. Ils contiendront donc uniquement des valeurs *XPath* :

```
<xsd:element name="get" type="xpath"/>
<xsd:element name="path" type="xpath"/>
```

Toutefois, nous n'avons pas défini le type primitif *xpath* dans notre schéma : nous supposons qu'il est défini par ailleurs (par le W3C par exemple).

La sémantique est la même que celle définie dans le chapitre précédent, tenant compte de la nouvelle syntaxe des références : le calcul du résultat de la référence par la fonction *calcRef* se fait alors suivant l'algorithme donné dans les recommandations du W3C pour *XPath* [146].

6.2.2.2 Groupe *operation*

Dans le modèle XML de VDLp, les éléments correspondant aux opérations arithmétiques (*plus*, *and*, *equals*, *etc*), aux références (*get*, *event get* et *size ref*) ainsi qu'à tous les autres concepts utilisés par la fonction d'interprétation canonique ζ (*size*, *node*, *etc*) sont regroupées dans le groupe *operation*, avec l'élément de valeur terminale quelconque *value* :

```
<xsd:group name="operation">
  <xsd:choice>
    <xsd:element name="equals" type="vdl:binaryOpType"/>
    <xsd:element name="plus" type="vdl:binaryOpType"/>
    ...
    <xsd:element name="value" type="xsd:anyType"/>
  </xsd:choice>
</xsd:group>
```

Ce groupe est donné en détail dans le schéma en annexe page A-6. Soulignons que nous utilisons des types spécifiques, définis dans le schéma, pour les opérations unaires, binaires et « binaires ou plus ».

La sémantique de ces éléments est définie par l'interprétation canonique ζ , implémentée en utilisant l'API « *DOM* ». L'interprétation canonique d'une valeur (*value*) est l'interprétation canonique de son contenu (par exemple, 2 est interprété comme le nombre entier deux).

Exemple

L'opération $1 + \frac{1}{2}$ est représentée dans notre langage par l'élément :

```
<plus>
  <value>1</value>
  <inverse><value>2</value></inverse>
</plus>
```

Cet élément sera évalué par ζ pour construire l'élément flottant³ :

```
<value>1.5</value>
```

6.2.2.3 Actions élémentaires

Les actions élémentaires de modifications de l'arbre XML, correspondant chacune à une balise XML, sont les mêmes que celles proposées pour VDLp :

- **add** pour ajouter des fils à un élément,
- **put** pour remplacer tout le contenu (tous les fils) d'un élément,
- **del** pour supprimer un élément et tous ses fils.

Ces trois opérations forment une base des modifications sur le document XML.

Le schéma *vdl.xsd* définit que chaque action élémentaire doivent nécessairement contenir au moins un élément *path* et que les éléments *add* et *put* doivent contenir au moins un élément du groupe *operation* donnant les attributs à ajouter :

```
<xsd:complexType name="basicActionType">
  <xsd:element name="path" type="xpath" minOccurs="1"/>
</xsd:complexType>
<xsd:complexType name="addActionType" base="vdl:basicActionType"
  derivedBy="extension">
  <xsd:group ref="vdl:operation" minOccurs="1"/>
</xsd:complexType>
```

La sémantique opérationnelle de ces trois éléments est celle définie par les fonction φ et ψ du chapitre 2, conformément au schéma illustré figure 2.5 page 67.

Nous donnons en annexe page A-10 un exemple d'action élémentaire en XML utilisée dans un contexte structurel riche.

³Comme en VDLp, les éléments construits sont flottants. Ils sont rattachés au document XML lors de sa réécriture par la fonction ψ .

6.2.2.4 Événements externes

Dans ce modèle XML, nous conservons l'idée de modéliser l'interaction d'un composant avec un utilisateur humain ou avec d'autres composants à l'aides des *événements externes* : ces événements sont des éléments XML qui peut être envoyé au composant au cours de son exécution. Nous utilisons l'élément **event** pour définir des préconditions par événement externe :

```
<xsd:element name="event">
  <xsd:complexType content="elementOnly"/>
</xsd:element>
```

Le contenu des événements externes n'est pas précisé car il peut (et doit) se conformer à un schéma spécifique au composant, définissant les interactions possibles. Toutefois, ces événements sont nécessairement des arbres.

6.2.2.5 Gardes booléennes

Les préconditions booléennes des actions sont définies dans l'élément **guard** :

```
<xsd:element name="guard">
  <xsd:complexType>
    <xsd:group ref="vdl:operation" minOccurs="1" maxOccurs="1"/>
  </xsd:complexType>
</xsd:element>
```

6.2.2.6 Actions

Les actions sont les éléments contenant au moins une action élémentaire, une précondition ou une sous-action, comme l'indique le schéma donné en annexe et comme le montrent les exemples ci-dessus. Ils sont matérialisés par l'élément **action** contenant exactement un élément **name** pour définir le *nom* de l'action (cf. section 2.3.4.2 page 63).

Exemple

Le code du compteur *Coco* est donnée en annexe *page A-11*.

6.2.3 Bilan

6.2.3.1 Travaux connexes

De nombreux travaux ont été proposé dans la communauté du Web Sémantique autour de la notion de *service web*. Le protocole d'échange de données SOAP [147]

définit une méthode unifiée de communication entre les services. Le modèle de services UDDI⁴ permet aux clients de rechercher et de trouver dynamiquement des services fournis sur le web. Le format de déclaration de points de service WSDL [22] permet au contraire aux services de s'annoncer sur le web afin d'être utilisés par des clients. Enfin, plusieurs modèles de description de services web, comme WSFL⁵, DAML-S [26] ou WSMF [39] étendent ces protocoles de bases pour permettre aux agents de trouver, invoquer, surveiller ou composer en services plus complexes des ressources Web offrant tels services et vérifiant telles propriétés.

Dans notre modèle, nous ne nous sommes pas intéressés aux protocoles d'interaction pour publier, rechercher ou relier entre eux des services dans le web sémantique. Notre modélisation des interactions sous la forme d'événements externes est, par comparaison, très simplifiée. Au contraire, nous souhaitons dépasser la notion de services et proposer un modèle de description de composants actifs intégrés dans des pages web, en vue de répondre à des questions sur le fonctionnement provenant d'*utilisateurs ordinaires*. Ainsi, les composants décrits dans notre modèle peuvent accéder en cours d'exécution à la description de leur actions pour interagir avec l'utilisateur ou d'autres composants.

De ce point de vue, l'ontologie de services DAML-S [26] proposée dans le cadre du programme DAML permet aussi aux services « d'introspecter » leurs actions. Cependant, plusieurs problèmes restent ouverts :

1. DAML-S se rapproche des méthodes de descriptions des actions fondées sur la logique (calcul des situations [91], *etc*), issues du raisonnement de bon sens [1]. Dans ces modèles, le système ne peut raisonner que sur la partie qui a été spécifiée par l'expert de manière statique lors de la conception du système.

Au contraire, pour pouvoir répondre à des requêtes sur son fonctionnement issues de questions en langue naturelle d'un utilisateur ordinaire, le système doit pouvoir raisonner sur le code même du programme en utilisant la sémantique opérationnelle du langage.

2. DAML-S propose des *protocoles* pour publier, rechercher ou relier entre eux des services, généralement implémentés sous la forme de scripts CGI et *séparés de l'information qu'ils manipulent*. Au contraire, les composants actifs dans le web sémantique doivent faire partie intégrante de cette information. C'est la raison pour laquelle nous proposons d'utiliser une représentation unifiée qui intègre au sein des pages web statiques des fonctionnalités dynamiques.

Notre langage n'utilise pas les opérateurs usuels des langages de programmation, comme *if*, *while*, *etc*, issus du modèle *implémentatoire* classique « instruction-valeur ». Au contraire, comme les modèles de description et de composition de services web (WSFL, WSMF, DAML-S...), il se situe *au niveau conceptuel*, dans lequel la sémantique des opérations effectuées est matérialisée par leurs préconditions. Il s'agit certes d'une conception différente de la programmation d'un composant actif, cela n'en fait pas moins un véritable langage de programmation. Mais nous

⁴<http://www.uddi.org/>

⁵<http://www.ebpm1.org/wsfl.htm>

avons montré au chapitre 2 que le langage VDL (et donc notre modèle procédural en XML) est aussi expressif que CSP [63], c'est-à-dire qu'il permet de représenter simplement tous les opérateurs de combinaison d'instructions rencontrés dans les modèles de processus communicants classiques. En particulier, notre modèle de description de composant permet de décrire toutes les constructions de « *process* » proposées par DAML-S (*Sequence*, *Concurrent*, *Split*, *Split + Join*, *Unordered*, *Choice*, *If – Then – Else*, *Repeat – Until* et *Repeat – While*).

6.3 Conclusion

Le web sémantique aujourd'hui s'est beaucoup orienté sur la description de l'information structurelle sémantisée et prend peu en compte l'étude du fonctionnement des services et, plus généralement, des composants actifs interagissant avec les utilisateurs. Ces composants sont aujourd'hui généralement « **enchassés** » dans les documents sous la forme d'applets, et non **intégrés** au contenu structurel du document. C'est la raison pour laquelle nous proposons d'utiliser le formalisme VDL comme fondement théorique pour l'intégration de la structure et du fonctionnement des composants actifs au sein de pages XML.

Dans notre étude, nous nous sommes intéressés tout particulièrement à la représentation du fonctionnement dans un arbre de réécriture. Cela nous a conduit à proposer un langage de programmation spécifique qui permet d'attacher, sur les aspects structurels riches, des aspects liés au fonctionnement, afin de décrire et d'exécuter les composants actifs sous la forme de pages web en XML. Le modèle proposé reste relativement simple et devra être étendu par la suite. Par exemple, nous pourrions proposer des opérateurs de manipulation des types primitifs utilisables dans les schémas XML. En l'état, ce modèle XML procédural est suffisamment générique pour permettre de représenter une large classe de composants actifs, capables de répondre d'une part à des questions sémantisées sur leur structure, conformément aux travaux actuels du Web Sémantique et d'autre part à des questions d'utilisateurs ordinaires sur leur fonctionnement.

Chapitre 7

Implémentation de VDLp

Dans le cadre du projet *InterViews*, le langage VDLp présenté au chapitre 2 a été intégralement implémenté en Java. Nous présentons cette implémentation dans ce chapitre. Nous montrons comment l'interprète VDL en Java, englobé dans une applet, peut être relié avec le MLN à l'aide d'une architecture client-serveur HTTP. Nous présentons ensuite les modifieurs *script*, *component* et *global*, définis suite à cette implémentation, qui permettent d'utiliser directement du code Java (non introspectable) dans les composants VDL. Nous illustrons les possibilités de cette implémentation sur un composant de domotique.

7.1 Interprète VDL en Java

Nous avons choisi d'implémenter un interprète VDL en Java 1.1.7 de manière à à pouvoir englober nos vues et leur interface graphique dans des applets.¹ Nous présentons dans cette section le travail qui a été effectué.

7.1.1 Présentation des classes Java

L'interprète VDL en Java est composé de 13 classes et interfaces définies dans 7 fichiers. Ces classes sont regroupées dans un paquetage VDL. L'ensemble des méthodes utilisées dans l'interprète sont donc *protected*, pour que seuls les classes du paquetage VDL (et en particulier les observateurs) puissent les utiliser.

Nous présentons ci-après les principales classes définies dans l'interprète VDL en Java. Le diagramme UML de cet interprète VDL est donné en annexe page A-13.

¹Cette implémentation de l'interprète VDL en Java a été testée sous différentes machines virtuelles Java (appletviewer de Sun, Netscape \geq 4.0 et Internet Explorer \geq 4.0) et sous différentes architectures (Windows, Linux, Unix, Macintosh). Une démonstration en ligne est disponible à l'adresse suivante : <http://www.limsi.fr/Individu/nico/exemples>

7.1.1.1 Classe *Node*

L'arbre VDL est implémenté par une classe, appelée *Node*, qui englobe la structure récursive en Java :

```
private Vector fils = new Vector();
private String concept;
private Node pere = null;
```

L'interprétation canonique sur les termes est définie dans la classe *Node*, par la méthode *interp* qui prend en argument une pile (*java.util.Stack*) d'événements externes devant être traités. Elle retourne un ensemble de termes :

```
protected Vector interp(Stack E)
```

La fonction *calcRef* qui calcule le résultat d'une référence est aussi implémentée dans la classe *Node*, dans la méthode *calcRef*. Elle prend en argument la référence dont il faut calculer le résultat dans le nœud destinataire du message :

```
public Vector calcRef(Vector v)
```

Enfin, la classe *Node* définit un ensemble de fonctions de construction, de manipulation et d'affichage de nœuds VDL, utilisées par les autres classes.

7.1.1.2 Interface *Observateur*

Notre objectif est de définir un modèle dans lequel tout programmeur peut définir un observateur sur les termes VDL. C'est la raison pour laquelle nous avons défini la structure générale des observateurs dans l'interface *Observateur* :

```
public interface Observateur {
    public void phi(Node n);
    public void stop();
}
```

Elle impose que toute classe implémentant un observateur doit définir les méthodes *phi* et *stop*. La méthode *stop* est utilisée pour signifier à l'observateur que l'applet a été détruite (lorsque l'utilisateur quitte la page web).

7.1.1.3 La classe *AppletVDL*

Les vues sont englobées dans une applet Java, définie par la classe *AppletVDL* qui hérite de *java.applet.Applet*. Cette classe est caractérisée par quatre attributs : la vue, la liste de événements externes reçus à chaque cycle, la liste des observateurs statiques et un observateur dynamique, modélisé par une instance de la classe *Exec* présentée plus loin :

```

public Node vue;
private Vector fileEvenements = new Vector();
private Vector listeObs = new Vector();
private Exec exec = new Exec(this);

```

Dans cette implémentation, l'observateur dynamique ne peut donc pas être modifié : c'est nécessairement l'observateur *exec* défini dans la classe *Exec*.

L'interface de l'applet est composée de plusieurs composants. Le composant central est réservé pour l'affichage de l'interface graphique. L'un de ces composants est une instance de la classe *Control* qui est responsable de l'exécution automatique des vues et qui définit les boutons de contrôle de cette exécution (arrêter ou redémarrer l'interprète VDL, envoyer des événements externes directement, visualiser une représentation textuelle de la vue dans l'état courant).

Enfin, la classe *AppletVDL* est munie de la méthode polymorphe *sendEvent* qui permet à n'importe quel programme Java d'envoyer des événements externes à la vue :

```

public void sendEvent(Node n)
public void sendEvent(String s)

```

Toute *vue* doit être une instance d'une classe qui hérite de *AppletVDL* et qui définit, lors de son initialisation, l'arbre de type *Node* représentant la vue englobée dans l'applet.

7.1.1.4 La classe *Exec*

L'observateur dynamique *exec* que nous avons présenté dans cette thèse est défini dans la classe *Exec*, qui implémente l'interface *Observateur*. La méthode *phi* dans cette classe fait appel à deux autres méthodes :

- *construireListeActions*, définie dans la classe *Exec*, qui implémente la fonction φ définie au chapitre 2;

```

protected void construireListeActions(Node noeud)

```

- *effectuerActions*, définie dans la classe *ListeActions*, qui implémente la fonction ψ .

La liste des modifications à chaque cycle d'exécution est implémentée par une classe spécifique : *ListeModifications* qui hérite d'une table de hachage dans laquelle les actions VDL sont indexées par leur représentation textuelle. Les quadruplets $(t, a, R, E) \in \Upsilon \times Act \times \mathcal{P}(\Upsilon) \times \mathcal{P}(\Upsilon)$ de cette liste de modification sont définis dans la classe *ListeActions*.

La classe *Exec* est munie d'un attribut *applet* de type *AppletVDL* qui est utilisé dans *construireListeActions* pour obtenir l'ensemble des événements envoyés à chaque cycle d'exécution.

7.1.1.5 La classe *Interf*

L'observateur *interf* qui construit à chaque cycle l'interface de la vue est défini dans la classe *Interf*, qui implémente l'interface *Observateur*. Cette classe contient un attribut *applet* de type *AppletVDL* qui est utilisé pour envoyer l'interface construite à l'applet. La méthode *phi* définie dans cette classe construit tout d'abord une interface, sous la forme d'un *java.awt.Panel* contenant les composants graphiques définis dans la vue puis envoie cette interface à l'applet Java :

```
public void phi(Node n) {
    Panel p = new Panel();
    faireInterf(p,n,new Stack());
    p.setLayout(new GridLayout(0,1));
    applet.changerInterface(p);
}
```

L'attribut *applet* est aussi utilisé par effet de bord, dans la construction de l'interface, pour construire et mémoriser dans l'applet la liste des événements externes associés à chaque bouton, en fonction de la définition du bouton dans la vue. Les éléments de cette liste sont des objets, instances de la classe *BoutonVDL*.

Enfin, pour dessiner des figures géométriques simples, nous utilisons une classe spécifique : *VDLColoredShape* caractérisée par trois attributs, correspondant aux trois paramètres des figures géométriques en VDL :

```
private int taille;
private String forme;
private Color couleur;
```

7.1.2 Interprétation des vues

L'interprétation d'une vue englobée dans une *AppletVDL* consiste simplement en l'envoi des messages *phi(vue)* aux différentes instances des observateurs. La classe *Exec* réécrit alors l'arbre *vue* en un nouvel arbre, à chaque cycle.

Le compilateur VDL

Pour simplifier le travail du programmeur, nous avons défini un compilateur VDL qui prend en entrée un fichier ASCII *toto.vdl*, contenant la description sous forme textuelle d'une vue et qui produit un fichier *toto.java*. La classe *toto* définie dans le fichier *toto.java* hérite de *AppletVDL* et définit l'attribut *vue* dans cette classe, correspondant à la vue à l'instant initial de l'exécution.

Par exemple, le code du composant *Coco* que nous avons donné en annexe page A-1 est exactement celui qui figure dans le fichier *coco.vdl* qui est utilisé pour compiler puis exécuter la vue *Coco*.

Le compilateur VDL a été écrit en PERL. Il est relativement simple² et robuste (il

²Ce compilateur se limite en effet à 375 lignes de PERL.

détecte les principales fautes de syntaxes dans la description textuelle d'une vue). Il peut prendre en argument une option pour définir l'ensemble des observateurs statiques qui doivent être intégrés dans l'applet VDL, pour l'interprétation de la vue.

La figure 7.1 illustre la compilation et l'exécution d'une vue sous la forme d'un applet dans une page web : à partir d'un fichier ASCII *toto.vdl*, le compilateur VDL produit un fichier *toto.java* qui peut ensuite être compilé pour constituer une applet Java (*toto.class*) ayant parmi ses attributs la structure de données obtenue, qui représente la vue à l'instant initial v_0 . L'exécution de cette applet réécrit la vue à chaque cycle.

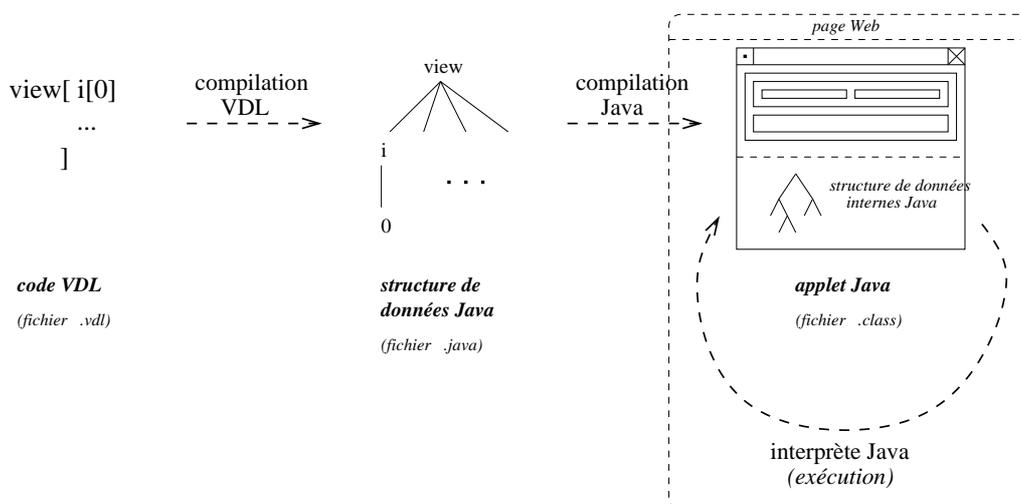


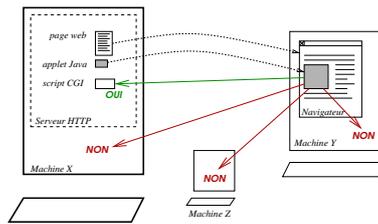
FIG. 7.1 – Principe général de l'implémentation de VDL procédural en Java

7.1.3 Architecture du projet *InterViews*

Dans le projet *InterViews*, le MLN est implémenté en MATHEMATICA 4.1 alors que l'interprète VDL et le proto-MRF sont implémentés en Java. Avec l'aide de G. Pitel, co-responsable du MLN au sein du projet *InterViews* avec S. Gérard, nous avons implémenté une architecture client-serveur permettant de relier ces deux modules.

7.1.3.1 Principe général

Il n'est pas possible de relier directement l'applet Java et le serveur MATHEMATICA : pour des raisons de sécurité évidentes, une applet peut uniquement communiquer avec le serveur HTTP d'où elle a été chargée par le navigateur :



La solution que nous avons déployée s'appuie sur une architecture client-serveur HTTP :

- L'applet communique avec un script CGI sur son serveur web ;
- Le script peut recevoir les commandes *init*, *eval* ou *close* ;
- Lorsque le script reçoit la commande *init*, il réveille le démon INET sur le serveur hébergeant le MLN. Ce démon lance une application serveur de communication et retourne au script le port de communication de cette application serveur. L'application lance le MLN sur le serveur MATHEMATICA, avec lequel il peut communiquer en utilisant l'API « *MathLink* » fournie par Wolframe Research ;
- Lorsque le script reçoit la commande *eval*, il envoie une expression MATHEMATICA à évaluer à l'application serveur, qui la transmet au serveur MATHEMATICA et retourne le résultat au script CGI, qui le donne comme résultat à l'applet Java ;
- Lorsque le script reçoit la commande *close*, il signale au serveur la fermeture de l'applet, ce qui déclenche l'arrêt du MLN pour cette vue.

Le schéma complet de l'interaction est donné sur la figure 7.2.

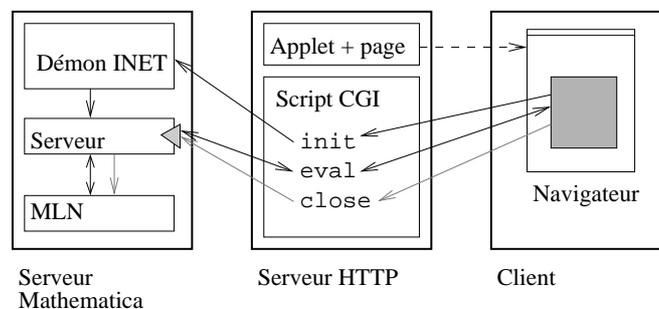


FIG. 7.2 – Interaction entre l'interprète VDL et le MLN

7.1.3.2 Classes utilisées

Pour permettre à l'interprète Java d'interagir avec le MLN, nous avons écrit une classe *NLAppletVDL*, héritée de *AppletVDL*, qui contient un attribut *cgi* de type *WebLinkCGI*. Lorsque l'utilisateur entre une phrase dans la zone de dialogue définie dans la classe *AppletVDL*, celle-ci envoie un message *parler* à l'applet qui déclenche la communication avec l'attribut *cgi*.

La classe *WebLinkCGI* est essentiellement un client CGI fondé sur l'utilisation de l'API *java.net.URL*. Nous ne la présenterons pas en détail ici. Le script CGI, en PERL, ainsi que l'application serveur, en Java, ont été réalisées par G. Pitel et ne seront pas présentées ici.

7.2 Utilisation de code Java en VDLp

Pour palier les manques de notre interprète VDL au niveau de l'interface graphique³, nous avons choisi de pouvoir utiliser directement du code Java dans les vues définies dans un fichier *.vdl*. Ce code n'est pas introspectable et n'est pas pris en compte dans les algorithmes du MRF. Il est seulement utilisé pour implémenter en VDL des composants plus riches et plus conviviaux, à l'intention des utilisateurs ordinaires. À terme, ils devraient disparaître au profit d'un plus grand nombre de modifieurs dédiés à la description de l'interface graphique.

7.2.1 Syntaxe

La définition du code Java dans les vues VDL se fait à l'aide de deux modifieurs spécifiques :

- *script* qui englobe du code Java quelconque⁴, éventuellement composé de plusieurs instructions. Par exemple :

```
script [ System.out.println("hello world !"); ]
```

Lorsque l'observateur rencontre ce modifieur, il exécute le code Java englobé ;

- *component* qui englobe du code Java, éventuellement composé de plusieurs instructions, qui retourne un composant AWT Java (*java.awt.Component* ou l'une de ses sous-classes). Ces composants seront ensuite ajoutés dans l'interface utilisateur de l'applet, comme une zone de texte ou un bouton. Par exemple :

```
component [return new java.awt.Button("hello world!");]
```

Lorsque l'observateur rencontre ce modifieur, il exécute le code Java englobé et affiche le composant retourné à l'endroit courant de l'interface graphique (selon les *H* et *V* rencontrés plus haut dans la vue).

Enfin, le mot clef *global* est utilisé lors de la compilation pour donner, sous la forme de code Java devant figurer dans le fichier *toto.java* générée, un ensemble de méthodes et d'attributs, éventuellement utilisés dans les modifieurs *script* et *component*. Le programmeur peut alors définir une classe *toto* (héritée de *AppletVDL* et donc de *Applet*) aussi riche qu'il le souhaite.

³Compte tenu de la définition de l'observateur *interf*, l'interface GUI des vues est limitée à des boutons, des zones de textes et quelques figures primitives. Au contraire, le langage Java fournit un nombre important d'API qui permettent de définir aisément des interfaces conviviales.

⁴Les crochets fermants dans le code Java inclus en VDL doivent seulement être protégés par le symbole `\`, pour éviter d'être interprétés comme des « fin de modifieur ».

Par exemple, supposons que le fichier *.vdl* compilé contienne le mot clef *global* et le code Java suivant:

```
global [ String s = "hello world !"; ]
```

Alors l'applet englobant la vue sera munie d'un attribut *s* de type *java.lang.String* et de valeur "hello world!", pouvant être utilisé dans les modifieurs *script* et *component* de la vue.

NB : Les mots-clefs *global* et leur « contenu » ne sont utilisés que lors de la compilation. Ils ne figurent pas dans l'arbre VDL de l'applet Java.

Remarque

Lorsqu'il utilise le mot-clef *global*, le programmeur doit avoir en tête le principe général de l'implémentation des vues en Java que nous avons présenté dans la section précédente, et en particulier le fait que les vues Java sont des sous-classes de la classe *AppletVDL*. Il peut ainsi redéfinir la méthode *init* (héritée de la classe *Applet*) dans le mot-clef *global* pour écrire du code devant être exécuté à l'initialisation de l'applet.

7.2.2 Sémantique

Lorsque de la compilation :

- Le code Java englobé dans le mot-clef *global* est ajouté en tête du fichier *.java* généré, parmi les attributs de la classe héritée de *AppletVDL* englobant la vue ;
- Pour chaque terme de concept *script*, le code Java englobé dans ce concept est associé à un numéro unique qui le caractérise tout au long de l'exécution. Ce code est alors incorporé dans la méthode *script*, définie de manière abstraite dans la classe *AppletVDL* et contenant simplement une instruction *switch* qui permet de déterminer quel code exécuté en fonction du numéro mémorisé dans le nœud :

```
public void script(int vdl_i, Stack E) {  
    switch(vdl_i) {  
        ...  
    } }  
}
```

Dans la vue, le code Java est remplacé par son numéro. Lors de l'interprétation, lorsque la fonction φ rencontre un modifieur *script*, elle extrait le numéro *vdl_i* donné par son unique attribut et exécute le code associé à ce numéro, par envoi du message *script(vdl_i, fileEvenements)* à l'attribut *applet* de type *AppletVDL*, défini dans l'observateur *Exec*.⁵

⁵Soulignons que, compte tenu de la signature de la fonction *script*, le programmeur ne peut pas utiliser de variable appelée *vdl_i* ou *E* dans le code englobé dans un *script* (à moins bien sûr de vouloir manipuler l'attribut du terme de concept *script* ou la file des événements envoyés au cours de ce cycle).

- Pour chaque terme de concept *component*, le code Java englobé dans ce concept est associé à un numéro unique puis inclus dans la méthode *scriptComponent* :

```
public Component scriptComponent(int vdl_i) {
    switch(vdl_i) {
        ...
    }
    return new Label();6
}
```

Dans la vue, le code Java est remplacé par son numéro. Lorsque la fonction ϕ_{interf} rencontre un modifieur *component* lors de l'interprétation, elle extrait le numéro *vdl_i* donné par son unique attribut et exécute le code associé à ce numéro, par envoi du message *scriptComponent(vdl_i)* à l'attribut *applet*. La figure 7.3 illustre la mécanique général d'utilisation de code Java dans les vues.

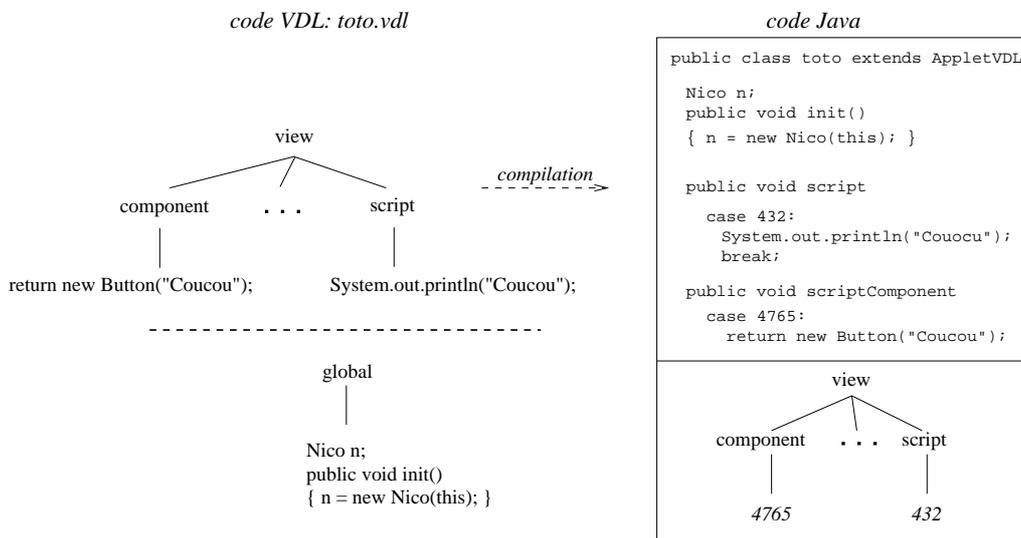


Figure 7.3: Utilisation de code Java dans les vues

7.2.3 Exemples

7.2.3.1 Un exemple simple

Voici un exemple simple utilisant les trois modifieurs *script*, *global* et *component*. Il s'agit d'une vue comprenant trois *java.awt.Checkbox* codées en Java et un bouton VDL tels que lorsque l'utilisateur clique dessus, la vue écrit le nombre de *checkbox* sélectionnés.

⁶ En cas d'échec, la fonction retourne tout de même un composant valide et affichable.

Les trois *java.awt.Checkbox* sont des objets globaux de l'applet englobant la vue: en effet, leur état est géré par l'applet et non par la partie VDL de la vue. Il faut donc les définir dans la section *global*. Le code de la vue est alors le suivant:

```

global [ Checkbox cb1 = new Checkbox("one");
        Checkbox cb2 = new Checkbox("two");
        Checkbox cb3 = new Checkbox("three");
      ]

view [ H [ component [ return cb1; ], component [ return cb2; ],
           component [ return cb3; ] ],
       button [text[count]],
       write [ event [ clic[count]],
               script [ int cpt = 0;
                        if (cb1.getState()) cpt++;
                        if (cb2.getState()) cpt++;
                        if (cb3.getState()) cpt++;
                        System.out.print(cpt+" ");
                        System.out.println("check boxes are on.");
                      ]
      ]
    ]

```

7.2.3.2 Exemple complet : *Mike* le four[144]

E. Valencia a proposé dans [144] d'utiliser les outils définis dans le projet *Inter-Views* pour implémenter des composants actifs de domotique. Le monde opératoire englobé par la vue est alors une modélisation du fonctionnement de l'appareil électroménager avec l'utilisateur souhaite interagir.

Nous présentons dans cette section le composant *Mike* [144], qui sert d'interface entre un utilisateur et un four muni d'une simple résistance. L'interface est composée de trois fonctions :

- La fonction *grill* qui allume la résistance pour chauffer le contenu du four pendant un intervalle de temps défini par l'utilisateur.
- La fonction *heat* (préchauffe) qui allume la résistance pour chauffer le contenu du four jusqu'à atteindre une température définie par l'utilisateur.
- La fonction *cook* (cuisson) qui allume la résistance pour chauffer le contenu du four à la température définie par l'utilisateur pendant un intervalle de temps défini par l'utilisateur.

Chacune de ces fonction de l'interface correspond à un processus différent dans la vue, c'est-à-dire à un mode de fonctionnement spécifique.

Interaction avec le four

L'interaction de la vue avec le four est gérée en entrée par un événement externe et en sortie par un observateur spécifique, appelé *mike*, qui utilise le modifieur

mike et allume la résistance dans le four si et seulement si un terme de concept *mike* a pour attribut l'atome *on*. Dans notre version de *Mike*, cet observateur a été implémenté en Java. E. Valencia [144] a montré qu'il pourrait être interface avec du code en *qBasic* pour permettre l'interaction avec véritable un four mécanique.

Interface Java

L'interface GUI du composant *Mike* est composée non seulement de composants graphiques VDL simples (boutons, zones de textes), mais aussi de composants Java :

- Des curseurs (*java.awt.Scrollbar*), pour contrôler la température et la durée de cuisson ;
- Des cases à cocher (*java.awt.Checkbox*) pour choisir la fonction ou arrêter le four (case *off*).

Ces composants sont définis dans des classes spécifiques : *MikeSwitch* qui englobe les quatre cases à cocher et *MikeCursor* qui englobe un curseur. Ces deux classes ont un attribut de type *AppletVDL* correspondant à la vue dans laquelle ils sont utilisées.

Ces composants interagissent avec la vue à l'aide d'événements externes :

- *fonction[x]* lorsque l'utilisateur clique sur la case à cocher *x*. Par exemple, *fonction[grill]* si l'utilisateur clique sur la case *grill*.
- *time[valeur]* lorsque l'utilisateur déplace le curseur pour changer la valeur du temps de cuisson.
- *temp[valeur]* lorsque l'utilisateur déplace le curseur pour changer la valeur de la température souhaitée.

Le code de ces composants est donné en annexe page A-14. Le code complet de la vue *Mike* est donné page A-16. La figure 7.4 page suivante montre l'interface du composant *Mike*, englobé dans une page Web. La fenêtre en haut à gauche, contrôlée par l'observateur *mike*, montre une visualisation de l'activité (en marche ou éteinte) de la résistance du four. La modification de la température dans le four est simulée par un processus aléatoire : la température augmente si et seulement si la résistance est allumée, avec une probabilité 0.3. Si la résistance est éteinte, elle diminue avec une probabilité 0.7.



FIG. 7.4 – La vue *Mike* dans une page Web.

Conclusion

La problématique de la construction automatique de réponses à des questions naturelles concernant les actions et le fonctionnement d'un composant actif n'a pas encore été abordée dans le domaine de l'interaction homme-machine. L'une des raisons est peut-être qu'elle se situe à la frontière entre deux domaines de l'intelligence artificielle qui sont restés longtemps éloignés : celui du raisonnement en logique sur les actions et le fonctionnement et celui de l'interaction entre un système et un utilisateur ordinaire.

Dans cette thèse, nous avons proposé une première approche pour apporter quelques éléments de réponse à ce problème qui devient d'autant plus crucial que les systèmes d'informations sont de plus en plus orientés vers les utilisateurs non spécialistes.

Nous avons tout d'abord défini un langage spécifique de programmation de composants actifs, appelé VDLp, qui permet d'avoir accès en cours d'exécution à une représentation intégrée de la structure et du fonctionnement. Nous avons ensuite présenté un langage de requêtes simple qui permet d'une part de modéliser une large classe de questions qu'un utilisateur humain peut poser concernant le fonctionnement d'un composant actif et d'autre part de proposer des mécanismes de traitement automatiques de ces requêtes.

Nous avons alors proposé des mécanismes de traitement, fondés sur la recherche et l'analyse des actions d'un composant VDLp. Nous avons défini un cadre formel, appelé *Ground*, pour modéliser et manipuler dans les requêtes des notions de *bon sens* comme celles qui interviennent lors de l'interaction entre un composant et un utilisateur ordinaire. Nous avons aussi présenté un modèle de mémorisation de l'exécution passée des composants, fondé sur la notion de *chronique* enregistrant les situations liés aux occurrences d'actions dans l'exécution du composant. Nous avons montré comment ces chroniques pouvaient être utilisées d'une part pour extraire des comportements à partir des flux d'interaction dans un système multi-agents, et d'autre part pour répondre aux requêtes sur le passé du composant.

Nous avons enfin montré que le langage VDLp pouvait être utilisé dans un cadre moins théorique que celui proposé ici, afin de permettre à des composants réels (dans le web sémantique par exemple) d'interagir avec des utilisateurs ordinaires.

L'ensemble de ces travaux constitue donc un cadre formel et opératoire complet pour définir des composants capables de raisonner de manière automatique sur leur fonctionnement, en réponse à des requêtes formelles. Dans le cadre du projet *In-*

terViews, ce cadre opératoire est intégré avec un module de traitement de langue naturelle pour permettre aux composants d'interagir *en langage naturel* avec des utilisateurs ordinaires.

Nous nous sommes attachés ici à présenter une approche générique pour le traitement de cette problématique encore nouvelle, mais ce travail constitue seulement une toute première étape et plusieurs limites importantes subsistent :

- Bien que le langage VDLp soit Turing-complet, il n'est pas utilisable directement pour définir des composants aux interfaces conviviales en raison de l'absence de concepts graphiques. Nous envisageons d'étendre notre langage pour définir de nouveaux composants graphiques à l'aide de concepts spécifiques, comme cela se fait dans les API des langages de programmation modernes, comme C, Java ou .net. A terme, nous voudrions proposer un véritable langage de programmation de composants pour le web sémantique, intégrant structure et fonctionnement et capable ainsi de définir des interfaces riches (sous la forme de pages web).
- Le langage de requêtes proposé, fondé sur un modèle vecteur, ne permet pas de représenter toutes les questions possibles et plusieurs combinaisons sont inconsistantes. Nous envisageons de proposer un nouveau langage, orthogonal et puissant comme SQL, pour modéliser une plus large classe de questions en langue naturelle sur le fonctionnement. Toutefois, l'étude de ce langage soulève le problème de son utilisation à la fois par le module de TALN et par le module de raisonnement sur le fonctionnement.
- Les mécanismes de constructions de réponse que nous avons proposé dans cette thèse s'appuient uniquement sur une analyse statique du composant, dans la lignée des travaux dans le domaine du raisonnement sur les actions et les changements [91, 1]. Nous avons proposé au chapitre 5 un mécanisme d'extraction de comportements fondé sur une analyse dynamique, mais nous n'avons pas intégré ces deux approches pour le traitement des requêtes formelles. Nous pensons pourtant qu'il existe des problèmes, en particulier ceux liés à la prise en compte des notions de bon sens, qui ne peuvent pas être résolus simplement par une analyse statique. Nous voudrions recenser ces problèmes et essayer de proposer des solutions fondées sur une analyse dynamique, à partir de l'extraction de schéma comportementaux dans les flux d'interaction. Nous n'avons malheureusement pas pu aborder cet aspect au cours de la thèse.
- Nous voudrions enfin étudier la modélisation et le traitement des *commandes directes*, c'est-à-dire la construction d'actions élémentaires VDLp pour modifier la vue à partir des questions de l'utilisateur en langue naturelle. L'étude de cette problématique nécessite de travailler à la fois sur les aspects « raisonnement sur le fonctionnement » et sur le traitement du langage naturel, pour proposer un langage de requête suffisamment expressif mais utilisable pour la construction automatique de commandes. Nous pensons que l'étude de ce modèle de requêtes peut conduire à la définition d'un nouveau cadre formel d'interaction homme-machine, autour de la commande en langage naturel des systèmes d'information.

Bibliographie

- [1] The Common Sense Problem Page. <http://www-formal.stanford.edu/leora/cs/>.
- [2] J.F. Allen. An Interval-Based Representation of Temporal Knowledge. In *Proc. 7th IJCAI*, pages 221–226, 1981.
- [3] J.F. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence*, 23(2) :123–154, 1984.
- [4] J.F. Allen, D.K. Byron, M. Dzikovska, G. Fergusson, L. Galescu, and A. Stent. Towards Conversational Human-Computer Interaction. *AI Magazine*, 2001.
- [5] J.F. Allen, G. Fergusson, and A. Stent. An architecture for more realistic conversational systems. In *Proc. Intelligent User Interfaces 2001 (IUI-01)*, pages 1–8, 2001.
- [6] T. Andreasen, J.F. Nilsson, and H.E. Thomsen. Ontology-Based Querying. In *Proc. 4th Intl. Conf. on Flexible Query Answering Systems (FQAS'2000)*, pages 15–26, 2000.
- [7] G. Ayala and Y. Yano. Interacting with a Mediator Agent in Collaborative Learning Environments. In Y. Anzai, K. Ogawa, and H. Mori, editors, *Symbiosis of Human and Artifact : Future Computing and Design for Human-Computer Interaction*, Advances in Human Factors/Ergonomics, pages 895–900. Elsevier, 1995.
- [8] C. Baral and M. Gelfond. Reasoning About Effects of Concurrent Actions. *Journal of Logic Programming*, 31 :85–118, 1997.
- [9] M. Basseville and M.O. Cordier. Surveillance et diagnostique de systèmes dynamiques : Approches complémentaires du traitement du signal et de l'intelligence artificielle. In *Le temps, l'espace et l'évolutif, Ecole thématique du GDR I3*, volume 2, pages 119–149. Cépadues-Editions, 2000.
- [10] Y. Bellik. *Interfaces multimodales : concepts, modèles et architectures*. PhD thesis, Université Paris XI, 1995.
- [11] T. Berners-Lee. *Weaving the Web*. Harper San Francisco, 1999.
- [12] H. Bestougeff and G. Ligozat. *Outils logiques pour le traitement du temps, de la linguistique à l'intelligence artificielle*. Etudes et recherches en informatique. Masson, 1989.
- [13] C. Brassac and S. Pesty. De la coaction à la coopération. In *Proc. JFIADSMA '96*, pages 251–263. Hermes, 1996.
- [14] A. Cardon and Z. Guessoum. Systèmes multi-agents adaptatifs. In *Proc. JFIADSMA 2000*, 2000.

- [15] P. Carle, P. Benhamou, M. Ornato, and F.X. Dolbeau. La Reconnaissance d'Intentions comme Dynamique des Organisations. In *Actes des 11emes Journées Franco-phones IAD/SMA*, pages 311–326. Hermès, 1998.
- [16] 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.
- [17] M.A. Castilho. *Modèles logiques pour le raisonnement sur les actions*. PhD thesis, IRIT, Toulouse III, 1998.
- [18] M.A. Castilho, L. Fariñas 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.
- [19] M.A. Castilho, O. Gasquet, and A. Herzig. Formalizing action and change in modal logic : the frame problem. Technical report, IRIT, 1998.
- [20] B. Chaib-Draa, B. Moulin, R. Mandiau, and P. Millot. Trends in Distributed Artificial Intelligence. *Artificial Intelligence Review*, 6 :35–66, 1992.
- [21] J. Charlet, M. Zacklad, G. Kassel, and D. Bourigault, editors. *Ingénierie des connaissances : Évolutions et récents défis*. Eyrolles, 2000.
- [22] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl). <http://www.w3.org/TR/wsdl>, 2001.
- [23] P.R. Cohen and H.J. Levesque. *Intentions in communication*, chapter Persistence, Intension and Commitment, pages 33–69. MIT Press, 1990.
- [24] M.O. Cordier and P. Siegel. A Temporal Revision Model for Reasoning about World Change. *International Journal of Intelligent Systems*, 9 :131–142, 1994.
- [25] K.W. Curie and A. Tate. O-Plan : the Open Planning Architecture. *Artificial Intelligence*, 52(1) :49–86, 1991.
- [26] DAML-S Coalition. *DAML-S : Web Service Description for the Semantic Web*, 2002.
- [27] E. Davis. Knowledge Preconditions for Plans. *Journal of Logic and Computation*, 4(5) :721–766, 1994.
- [28] S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker : The Very High Idea. In *Proc. FLAIRS-98*, pages 131–135, 1998.
- [29] S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker : Ontology Based Access to Distributed and Semi-Structured Information. In R. Meersman et al., editor, *Semantic Issues in Multimedia Systems*, pages 351–369. Kluwer Academic, 1999.
- [30] G. Delcastillo, I. Durdanovic, and U. Glässer. An Evolving Algebra Abstract Machine. In *Selected papers from Computer Science Logic 95*, volume 1092 of *LNCS*, pages 191–214. Bünig editor, 1996.
- [31] 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.
- [32] 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.

- [33] C. Dousson. *Suivi d'évolution et reconnaissances de chroniques*. PhD thesis, LAAS, Toulouse, 1994.
- [34] C. Dousson, P. Gaborit, and M. Ghallab. Situation Recognition : representation and algorithms. In *Proc. 13th IJCAI*, pages 166–172, 1993.
- [35] B. Dunin-Keplicz and R. Verbrugge. Collective intention constitutes a group. In Demazeau et al., editor, *Proc. MAAMAW'01*, 2001.
- [36] B.C. Falkenhainer and R.S. Michalski. Integrating Quantitative and Qualitative Discovery in the ABACUS System. In Y. Kodratoff and R.S. Michalski, editors, *Machine Learning : An Artificial Intelligence Approach*, volume 3, pages 153–190. Morgan Kaufmann, 1990.
- [37] A. El Fallah, S. Haddad, and H. Mazouzi. Observation répartie et analyse des interactions dans un système multi-agents. In *Proc. JFIADSMA '98*, pages 281–294, 1998.
- [38] C. Fellbaum, editor. *WordNet, An Electronic Lexical Database*. MIT-Press, 1998.
- [39] D. Fensel and C. Bussler. The Web Services Modeling Framework WSMF. In *1st meeting of the "Semantic Web enabled Web Services workgroup"*, 2002.
- [40] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 3 :251–288, 1972.
- [41] R.E. Fikes and N.J. Nilsson. STRIPS : A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2 :189–208, 1971.
- [42] J.J. Finger. *Exploiting Constraints in Design Synthesis*. PhD thesis, Stanford, 1987.
- [43] D. Fisher and J. Schlimmer. Models of Incremental Concept Learning : A coupled research proposal. Technical report, Vanderbilt University, 1988.
- [44] K.D. Forbus. Qualitative Process Theory. *Artificial Intelligence*, 24 :85–167, 1984.
- [45] M. Gelfond and V. Lifschitz. Representing Action and Changes by Logic Programs. *Journal of Logic Programming*, 17 :301–321, 1993.
- [46] M. Gelfond and V. Lifschitz. Action languages. *ETAI*, 3(16), 1998.
- [47] M.R. Genesereth and R.E. Fikes. *Knowledge Interchange Format Reference Manual v3.0*, 1992.
- [48] M. Ghallab. Planification de tâches en robotique. In *Le temps, l'espace et l'évolutif*, volume 2, pages 151–170. Cépadués-Editions, 2000.
- [49] J.J. Gibson. *The Ecological Approach to Visual Perception*. Houghton Mifflin, Boston, 1979.
- [50] E. Giunchiglia, G.N. Kartha, and V. Lifschitz. Representing Action : Indeterminacy and Ramifications. *Artificial Intelligence*, 95(2) :409–438, 1997.
- [51] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation : preliminary report. In *Proc. AAAI-98*, pages 623–630, 1998.
- [52] S. Gérard 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.
- [53] Z. Guessoum and J.P. Briot. From Active Objects to Autonomous Agents. *IEEE Concurrency*, 7(3) :68–76, 1999.

- [54] Y. Gurevich. Evolving Algebra Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
- [55] Y. Gurevich and J.K. Huggins. The semantics of the C programming language. volume 702, pages 274–308. Springer-Verlag, 1993.
- [56] R.J. Hall. Trusting your agent. In *Proc. 11th Knowledge-based Software Engineering Conference*. IEEE Computer Press, 1996.
- [57] S. Hanks and D.V. McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In *Proc. AAAI'86*, pages 328–333, 1986.
- [58] D. Harel. Dynamic Logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 497–604. Reidel, 1984.
- [59] J.P. Haton, N. Bouzid, F. Charpillet, M.C. Haton, B. Laasri, H. Laasri, P. Marquis, T. Mondot, and A. Napoli. *Le raisonnement en Intelligence Artificielle*. Informatique Intelligence Artificielle. InterEdition, 1991. chap. 5, p. 183-229 et chap. 7, p. 285-312.
- [60] P.J. Hayes. The Naive Physics Manifesto. In D. Michie, editor, *Expert Systems in the Micro-Electronic Age*, pages 242–270. Edinburgh University Press, 1979.
- [61] C. Hewitt. The challenge of open systems. *Byte*, 1985.
- [62] H. Hexmoor, J. Lammens, G. Caicedo, and S. Shapiro. Behaviour based AI, cognitive processes, and emergent behaviors in autonomous agents. In *Applications of Artificial Intelligence in Engineering*, pages 447–461, 1993.
- [63] C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall, 1986.
- [64] J. Hobbs. Causality. In *Proc. CommonSense 2001*, pages 145–155, 2001.
- [65] H.O. Ismail and S. Shapiro. Two Problems with Reasoning and Acting in Time. In A.G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proc. KR 2000*, pages 235–365. Morgan Kaufmann, 2000.
- [66] J. De Kleer. How Circuits Work. *Artificial Intelligence*, 24(1-3) :205–280, 1984.
- [67] K. Jokinen. Cooperative Response Planning in CDM : Reasoning about Communicative Strategies. In *Proc. Twente Workshop on Language Technology 11 : Dialogue Management in Natural Language Systems*, pages 159–168, 1996.
- [68] A. Kakas and R. Miller. A Simple Declarative Language for Describing Narratives with Actions. *Journal of Logic Programming*, 31 :157–200, 1997.
- [69] A.M. Kappel. Executable Specifications based on Dynamic Algebras. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 698 of *LNAI*, pages 229–240. Springer-Verlag, 1993.
- [70] D. Kayser. *La représentation des connaissances*. Collection Informatique. Hermès, 1997. sect. 6.5, p. 213-223.
- [71] W. Kintsch. The role of knowledge in discourse comprehension : a construction-integration model. *Psychological review*, 95 :163–182, 1988.
- [72] Y. Kodratoff and C. Nedellec. Modeling Machine Learning and Comprehensibility. In *Proc. IJCAI'95*, 1995.
- [73] R.A. Kowalski and M.J. Sergot. A logic-Based Calculus of Events. *New Generation Computing*, 4 :67–95, 1986.

- [74] B. Kuipers. Commonsense reasoning about causality : deriving behavior from structure. *Artificial Intelligence*, 24(1-3) :169–203, 1984.
- [75] B. Kuipers. *Qualitative Reasoning : Modeling and Simulation with Incomplete Knowledge*. MIT Press, 1994.
- [76] B. Lemaire. Hypothetical reasoning within the blackboard model for constructing explanations. In *Proc. ECAI'92*, pages 244–248, 1992.
- [77] B. Lemaire, B. Safar, and C. Yonnet. L'acquisition des connaissances d'explication. In N. Aussenac-Gilles and P. Laublet, editors, *Acquisition et ingénierie de la connaissance - Tendances actuelles*, pages 367–384. Cépadués, 1996.
- [78] E.J. Lemmon and D.S. Scott. *The Lemmon Notes : an introduction to Modal Logic*. Oxford University Press, Oxford, 1977.
- [79] 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.
- [80] A. Levy and M.C. Rousset. Combining Horn Rules and Description Logics in CARIN. *Artificial Intelligence*, 104 :165–209, 1998.
- [81] H. Lieberman. Autonomous Interface Agents. In *Proc. ACM Conference on Human-Computer Interface (CHI'97)*, 1997.
- [82] V. Lifschitz. Situation calculus and causal logic. In *Proc. KR'98*, pages 536–546, 1998.
- [83] K.E. Lochbaum, B.J. Grosz, and C.L. Sidner. Models of Plans to Support Communication : An Initial Report. In *Proc. AAAI'90*, pages 485–490, 1990.
- [84] S. Luke, L. Spector, D. Rager, and J. Hendler. Ontology based Web Agents. In *Proc. 1st Intl Conf. on Autonomous Agents*, 1997.
- [85] P. Maes. Agents that reduce workload and information overload. *Communications of the ACM*, 37(7), 1994.
- [86] N. McCain and H. Turner. A causal theory of ramifications and qualifications. In *Proc IJCAI'95*, pages 1978–1984, 1995.
- [87] J. McCarthy. Addendum : Circumscription and other Non-Monotonic Formalisms. *Artificial Intelligence*, 13(1-2) :171–172, 1980.
- [88] J. McCarthy. Circumscription - A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13(1-2) :27–39, 1980.
- [89] J. McCarthy. Applications of circumscription to formalizing commonsense knowledge. *Artificial Intelligence*, 1986.
- [90] J. McCarthy. Elephant 2000 : A programming language based on speech acts. Technical report, Stanford University, 2000. draft version.
- [91] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. *Machine Intelligence*, 4 :463–502, 1969.
- [92] D.V. McDermott. Temporal Logic for Reasoning about Processes and Plans. *Cognitive Science*, 6 :101–155, 1982.
- [93] 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.

- [94] J. Meseguer. Conditional Rewriting Logic : Deduction, Models and Concurrency. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems : Proc. of the 2nd International CTRS Workshop 1990*, pages 64–91. Springer-Verlag, 1991.
- [95] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1) :73–155, 1992.
- [96] J. Meseguer. Rewriting Logic and Maude : Concepts and Applications. In *Proc. RTA 2000*, pages 1–26, 2000.
- [97] M. Moulet. *ARC : un système de découverte empirique de lois numériques ou ABA-CUS Revu et Corrigé*. PhD thesis, LRI, Paris XI, 1993.
- [98] D.A. Norman. *The invisible computer : why good products can fail, the personal computer is so complex, and information appliances are the solution*. MIT Press, Cambridge, 1998.
- [99] J. Pearl. Reasoning With Cause and Effect. In *Proc. IJCAI'99*, pages 1437–1449, 1999.
- [100] J. Pinto. *Temporal Reasoning in the Situation Calculus*. PhD thesis, Dept. of Computer Science, Univ. of Toronto, 1994.
- [101] J. Pinto and R. Reiter. Reasoning about Time in the Situation Calculus. *Annals of Mathematics and Artificial Intelligence*, 14 :251–268, 1995.
- [102] G. Pitel. Un Modèle Distribué d'Interprétation de Requêtes fondé sur la notion d'Observateur. In *Poster à RECITAL 2002*, 2002.
- [103] A. Popescu-Belis, I. Robba, and G. Sabah. Reference Resolution Beyond Coreference : a Conceptual Frame and its Application. In *Coling-ACL '98*, pages 1046–1052, 1998.
- [104] V.R. Pratt. Semantical considerations on Floyd-Hoare Logic. In *Proc. 17th FOCS*, pages 109–121. IEEE, 1976.
- [105] A.N. Prior. *Time and Modality*. Clarendon, Oxford, 1957.
- [106] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1 :81–106, 1986.
- [107] J.R. Quinlan. *C4.5 : Programs for machine learning*. Morgan Kaufmann, 1993.
- [108] R. Van De Sluis, B. Eggen, J. Jansen, and H. Kohar. User Interface for an In-Home Environment. In *Proc. INTERACT'01*, pages 383–390. IOS Press, 2001.
- [109] R. Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13(1-2) :81–132, 1980.
- [110] R. Reiter. The Frame Problem in the Situation Calculus : A Simple Solution (sometimes) and a Completeness Result for Goal Regression. In *Papers in Honor of John McCarthy*, Artificial Intelligence and Mathematical Theory of Computation, pages 359–380. 1991.
- [111] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Proc. 5th International Conference on Principles of Knowledge Representation and Reasoning*, 1996.
- [112] R. Reiter. *Knowledge in Action : Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

- [113] C. Reynaud, B. Safar, and H. Gagliardi. Une expérience de représentation d'une ontologie dans le médiateur PICSEL. In *Proc. IC'2001*, pages 329–348, 2001.
- [114] S. Russel and P. Norvig. *Artificial Intelligence : A Modern Approach*. Prentice Hall, 1995.
- [115] N. Sabouret. VDL Procédural : syntaxe et sémantique opérationnelle. Technical Report 2000-09, LIMSI-CNRS, 2000.
- [116] N. Sabouret. A model of requests about actions for active components in the semantic web. In *Proc. STAIRS 2002*, pages 11–20, 2002.
- [117] N. Sabouret and J.P. Sansonnet. Automated Answers to Questions about a Running Process. In *Proc. CommonSense 2001*, pages 217–227, 2001.
- [118] 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.
- [119] E. Sandewall. Features & Fluents. Technical Report R-91-29, LiTH-IDA, Linköpings universitet, 1991.
- [120] J.P. Sansonnet. Description Scientifique du Projet InterViews — The InterViews Project. Technical Report 99-01, LIMSI-CNRS, 1999.
- [121] J.P. Sansonnet. Présentation de VDL 0.1. Technical Report 99-10, LIMSI-CNRS, 1999.
- [122] J.P. Sansonnet. Notes sur VDL 0.2. draft version, 2000.
- [123] J.P. Sansonnet. VDL 0.3. draft version, 2000.
- [124] J.P. Sansonnet. VDL 0.4. draft version, 2001.
- [125] J.P. Sansonnet, N. Sabouret, and G. Pitel. An Agent Design and Query Language dedicated to Natural Language Interaction. In *Poster at the AAMAS Conference*, 2002.
- [126] J.P. Sansonnet and E. Valencia. An Experimental Framework for Behavior Extraction over Communication Flows in Multi-Agent Systems. In *Proc. Agent2000 Workshop on Agent Communication*, 2000.
- [127] A. Schönegge. Extending Dynamic Logic for Reasoning about Evolving Algebras. Technical report, ILKD, Universität Karlsruhe, 1995.
- [128] B. Schneiderman. The limit of speech recognition. *Communications of the ACM*, 43(9), 2000.
- [129] J.R. Searle. *Speech Acts*. Cambridge University Press, 1969.
- [130] J.R. Searle and D. Vanderveken. *Foundations of illocutionary logic*. Cambridge University Press, 1985.
- [131] M. Shanahan. Explanation in the Situation Calculus. In *Proc. IJCAI'93*, pages 160–165, 1993.
- [132] S. Shapiro. Formalizing English. *International Journal of Expert Systems*, 9(1) :151–171, 1996.
- [133] S. Shapiro and W.J. Rapaport. The SNePS Family. *Computers & Mathematics with Applications*, 23(2-5) :243–275, 1992.

- [134] Y. Shoham. Temporal Logics in Artificial Intelligence : Semantical and Ontological Considerations. *Artificial Intelligence*, 33 :89–104, 1987.
- [135] M. Sintek and S. Decker. TRIPLE – an RDF Query, Inference, and Transformation Language. In *Proc. DDLP'2001*, 2000.
- [136] Léa Sombé, alias Ph. Besnard, L. Cholvy, M-O. Cordier, D. Dubois, L. Farinas del Cerro, Ch. Froidevaux, Y. Moinard, H. Prade, C. Schwind, and P. Siegel. Reasoning under Incomplete Information in Artificial Intelligence. *Special Issue of the International Journal of Intelligent Systems*, 5(4), 1990.
- [137] J.F. Sowa. *Conceptual Structures : Information Processing in Mind and Machines*. Addison-Wesley, 1984.
- [138] R.C. Stalnaker. Assertion. In P. Cole, editor, *Syntax and Semantics : Pragmatics*, volume 9, pages 315–332. New York : Academic, 1978.
- [139] A. Thayse, editor. *Approche logique de l'intelligence artificielle*. Dunod, 1988-1991.
- [140] The DARPA Knowledge Sharing Initiative External Interfaces Working Group. Specification of the KQML Agent-Communication Language. Technical report, DARPA, 1993.
- [141] M. Thielscher. A Theory of Dynamic Diagnosis. *Electronic Transactions on Artificial Intelligence (ETAI)*, 1997.
- [142] P. Utgoff. ID5 : An Incremental ID3. In J. Laird, editor, *Proc. 5th International Workshop on Machine Learning*, pages 107–120. Morgan Kaufmann, 1988.
- [143] P. Utgoff. An Improved Algorithm for Incremental Induction of Decision Trees. In *Proc. 11th ICML*, pages 318–325. Morgan Kaufmann, 1994.
- [144] E. Valencia and J.P. Sansonnet. Mike or InterViews for Home Automation Applications. Technical report, LIMSI-CNRS, sept. 1999. draft version.
- [145] A. Vilnat. *L'élaboration d'interventions pertinentes dans une conversation homme-machine*. PhD thesis, Université Paris VI, 1984.
- [146] W3C. XML Path Language (XPath). <http://www.w3.org/TR/xpath>, 1999.
- [147] W3C. Simple object access protocol (soap). <http://www.w3.org/TR/SOAP>, 2000.
- [148] W3C. Document Object Model (DOM) core specifications. <http://www.w3.org/TR/xslt>, 2002.
- [149] T. Winograd. *Understanding Natural Language*. New York Academic Press, 1972.

Annexes

Code VDL de *Coco*

```
view[definition["Coco the counter"],

    value[0],
    speed[1],
    running[true],

    text["Coco the counter"],
    H[ text[value:],text[get[value]] ],
    H[ text[speed:],text[get[speed]] ],
    H[ text[running:],text[get[running]] ],

    H[ button[text[stop]], button[text[start]],
      button[text[+] ], button[text[-]] ],

    count[ guard[get[running]],
           put[path[value], plus[get[value],get[speed]]] ],

    stop[ event[stop,clic[stop]],
          put[path[running],false] ],
    start[ event[start,clic[start]],
          put[path[running],true] ],
    "speed up"[ event[faster,+,clic[+] ],
              put[path[speed],plus[get[speed],1]] ],
    "slow down"[ event[slower,-,clic[-]],
                put[path[speed],plus[get[speed],-1]] ]

]
```

Corpus de *Coco*

<i>Num</i>	<i>Questions</i>	<i>Réponses</i>
1	Qu'est-ce que tu fais ?	Je compte.
2	Qu'est-ce que tu comptes ?	Je compte ma valeur.
3	Pourquoi tu comptes ?	Parce que je suis en marche.
4	Qui es-tu ?	Je suis <i>Coco</i> le compteur.
5	Pourquoi es-tu là ?	Je ne sais pas.
6	Vas-t-en !	Je ne comprends pas.
7	Tu sers à quoi ?	À compter.
8	Comment fais-tu pour compter ?	J'incrémente ma valeur avec ma vitesse.
9	Est-ce que tu es en train de compter ?	Oui.
10	Est-ce que tu peux compter moins vite ?	Oui. Tu veux ?
11	Tu es en train de compter ou de dormir ?	De compter.
12	Pourquoi est-ce que tu ne fais plus rien ?	Parce que je suis arrêté.
13	Pourquoi est-ce que tu ne comptes plus ?	Je compte mais ma vitesse est nulle. Alors ma valeur augmente de 0.
14	Pourquoi est-ce que tu comptes à l'envers ?	Parce que ma vitesse est négative.
15	Comment faire pour t'arrêter ?	Clique sur le bouton « stop ».
16	Tu es arrêté, là ?	Oui.
17	Quelle est ta valeur ?	47.
18	Est-ce que ta valeur est 52 ?	Non. Ma valeur est 47.
19	Est-ce que ta valeur est plus grande que 10 ?	Oui.
20	Pourquoi est-ce que la valeur a changé ?	Parce que je compte.
21	Est-ce que tu peux compter ?	Non parce que je ne suis pas en marche.
22	Pourquoi ne peux-tu pas compter ?	Parce que je suis arrêté.

<i>Num</i>	<i>Questions</i>	<i>Réponses</i>
23	A quelle vitesse vas-tu ?	Ma vitesse est 3.
24	Pourquoi est-ce que tu es arrêté ?	Parce que tu as cliqué sur « stop ».
25	Va plus vite !	OK.
26	Arrête-toi ! Stop !	OK.
27	Tu ne comptes plus ?	Non. Effectivement.
28	Recommence à zéro !	Je ne sais pas le faire.
29	Pourquoi est-ce que tu n'as pas recommencé à compter ?	Parce que tu ne l'as pas demandé. Tu veux que je recommence ?
30	Je veux que tu t'arrêtes quand tu arrives à 10.	Je ne sais pas le faire.
31	Pourquoi fais-tu tout ce que je dis ?	Je suis là pour ça.
32	Qu'est-ce que tu pourrais faire ?	Je peux compter.
33	Tu peux compter moins vite ?	OK.
34	Qu'est-ce que je peux faire ?	Tu peux me demander de m'arrêter d'aller plus vite ou moins vite...
35	Quelle était ta valeur il y a 10 minutes ?	Ma valeur était 45 il y a 10 minutes.
36	Quelle sera ta valeur dans 3 minutes ?	Si tu ne changes rien, ma valeur sera 38.
37	Quelle était ta valeur 5 minutes avant que je ne dise « stop » ?	Il y a 8 minutes, ma valeur était 32.
38	Quelle était ta valeur quand j'ai changé ta vitesse à 2 ?	Il y a 2 minutes, ma valeur était 12.
39	Combien vaudras-tu quand je changerai ta vitesse ?	Je ne peux pas le prévoir.
40	Quand ais-je changé ta vitesse ?	Ma vitesse a été modifiée il y a 7 minutes.
41	Est-ce que tu comptais il y a 3 minutes ?	Oui.
42	Qu'est-ce que tu faisais il y a 5 minutes ?	Rien.
43	Est-ce que tu peux me faire un café ?	Je ne sais pas faire un café.
44	Est-ce que tu peux augmenter mon salaire ?	Je n'ai pas de salaire.

Fonction χ_{proc}

La fonction $\chi_{proc} : \Upsilon \times \mathcal{P}(\Upsilon) \times \mathcal{P}(\mathcal{P}(\Upsilon)) \longrightarrow \mathcal{P}(\mathcal{P}(\Upsilon))$ recherche les actions dans un terme (t) en filtrant par les éléments du second argument (l , initialement égal à l'objet ω de la requête), lorsque ce n'est pas l'ensemble vide. Elle retourne un ensemble d'ensembles A_i d'actions. Chaque A_i contient les actions correspondant au $i^{\text{ème}}$ élément du filtre. Lorsque $l = \emptyset$, χ_{proc} retourne un simpleton $\{A_0\}$ contenant toutes les actions trouvées.

Le troisième argument de la fonction, initialement égal à $\{\emptyset, \dots, \emptyset\}$, est utilisé pour construire de manière récursive le résultat global.

Algorithme

```
fonction  $\chi_{proc}(t \in \Upsilon, l \in \mathcal{P}(\Upsilon), res \in \mathcal{P}(\mathcal{P}(\Upsilon)))$ 

  si  $\text{fils}(t) = \emptyset$  ou  $\text{concept}(t) \in \{\text{guard}, \text{event}\}$ , retourner  $res$ 

   $E = \{t' \in \text{fils}(t) \text{ tq } \text{concept}(t') = \text{event}\}$ 
   $G = \{t' \in \text{fils}(t) \text{ tq } \text{concept}(t') = \text{guard}\}$ 
   $A = \{t' \in \text{fils}(t) \text{ tq } \text{concept}(t') \in \{\text{add}, \text{put}, \text{del}\}\}$ 

  si  $G = A = E = \emptyset$  et  $\text{concept}(t) \notin \{\text{add}, \text{put}, \text{del}\}$ 
    /* ce n'est pas une action */
    pour  $i$  allant de 1 à  $|\text{fils}(t)|$ 
       $res = \chi_{proc}(\text{fils}(t)[i], l, res)$ 
    /* rechercher dans les attributs */
  sinon
    si  $l = \emptyset$ 
       $res[1] = res[1] \cup \{t\}$ 
    sinon /* filtre */
       $ok = \perp$ 
      si  $\text{concept}(t) \in \{\text{add}, \text{put}, \text{del}\}$ 
        /* action élémentaire  $\rightarrow$  subsomption */
        pour  $i$  allant de 1 à  $|l|$ 
          si  $t \preceq l[i]$ 
             $res[i] = res[i] \cup \{t\}$ 
```

```

        ok = ⊥
    sinon /* action VDL → simple comparaison de concepts */
        pour i allant de 1 à |l|
            si concept(t)=concept(l[i])
                res[i] = res[i] ∪ {t}
            ok = ⊥
    si (non ok) /* l'action courante n'est pas dans le filtre */
        si concept(t)∉{add,put,del,event,guard}
            /* sous-actions possibles */
            X = fils(t)\(GUA)
            pour i allant de 1 à |X|
                res =  $\chi_{proc}$ (X[i],l,res)

retourner res

```

Schéma *vdl.xsd*

```
<xsd:schema xmlns="http://www.limsi.fr/Individu/nico/xml/vdl.xsd"
            xmlns:vdl="http://www.limsi.fr/Individu/nico/xml/vdl.xsd"
            targetNamespace="http://www.limsi.fr/Individu/nico/xml/vdl.xsd"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Procedural VDL schema for describing actions in XML documents.
      Copyright 2002 LIMSI-CNRS.
    </xsd:documentation>
  </xsd:annotation>

  <!-- The view -->
  <!-- This view element is not required by the operational semantics.
        However, it is required by the "each document should have one root"
        rule, when describing purely procedural components (using only the
        vdl schema) -->

  <xsd:element name="view">
    <xsd:complexType content="elementOnly"/>
  </xsd:element>

  <!-- Arithmetic operations -->

  <!-- We do not take into account any type: everything is a "value" :- (
        In future work, we shall define each operation wrt to each
        primitive type... -->

  <xsd:group name="operation">
    <xsd:choice>

      <xsd:element name="equals" type="vdl:binaryOpType"/>
    
```

```

<xsd:element name="plus" type="vdl:binaryOpType"/>
<xsd:element name="times" type="vdl:binaryOpType"/>
<xsd:element name="minus" type="vdl:unaryOpType"/>
<xsd:element name="inverse" type="vdl:unaryOpType"/>
<xsd:element name="and" type="vdl:binaryOpType"/>
<xsd:element name="or" type="vdl:binaryOpType"/>
<xsd:element name="not" type="vdl:unaryOpType"/>
<xsd:element name="size" type="vdl:binaryOpType"/>

<!-- The "get", "event-get" and "size-ref" elements' contents are
      references and should conform to the XPath syntax. However, the
      "xpath" primitive type that would correspond is not given in
      detail here. -->
<xsd:element name="get" type="xpath"/>
<xsd:element name="event-get" type="xpath"/>
<xsd:element name="size-ref" type="xpath"/>

<!-- For "greater-than" and "is-a", we will use the order of the
      sub-elements: [1]>[2] -->
<xsd:element name="greater-than" type="vdl:comparisonOpType"/>
<xsd:element name="ia-a" type="vdl:comparisonOpType"/>

<!-- The "random" operation does not take any argument -->
<xsd:element name="random">
  <xsd:complexType content="empty"/>
</xsd:element>

<!-- The "node" operator requires one concept and possibly attributes -->
<xsd:element name="node">
  <xsd:complexType>
    <xsd:element name="concept" type="vdl:comparisonOpType"
      minOccurs="1" maxOccurs="1"/>
    <xsd:group ref="vdl:operation" minOccurs="0"/>
  </xsd:complexType>
</xsd:element>

<!-- Finally, here are the values (no type) -->
<xsd:element name="value" type="xsd:anyType"/>

</xsd:choice>
</xsd:group>

<xsd:complexType name="unaryOpType">
  <xsd:group ref="vdl:operation" minOccurs="1" maxOccurs="1"/>
</xsd:complexType>

```

```

<xsd:complexType name="comparisonOpType">
  <xsd:group ref="vdl:operation" minOccurs="2" maxOccurs="2"/>
</xsd:complexType>

<xsd:complexType name="binaryOpType">
  <xsd:group ref="vdl:operation" minOccurs="2"/>
</xsd:complexType>

<!-- Actions -->

<!-- 1. the types for basic actions -->
<xsd:complexType name="basicActionType">
  <xsd:element name="path" type="xpath" minOccurs="1"/>
  <!-- The "path" elements' contents are references and should
        conform to the XPath syntax -->
</xsd:complexType>
<xsd:complexType name="addActionType">
  <xsd:complexContent>
    <xsd:extension base="vdl:basicActionType">
      <xsd:group ref="vdl:operation" minOccurs="1"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- 3. actions -->

<xsd:element name="action"/>
<xsd:complexType>
  <xsd:element name="name" type="xsd:string"/>
  <xsd:group minOccurs=1>
    <xsd:choice>

      <!-- 2.1. preconditions -->
      <xsd:element name="guard">
        <xsd:complexType>
          <xsd:group ref="vdl:operation" minOccurs="1" maxOccurs="1"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="event">
        <xsd:complexType content="elementOnly"/>
      </xsd:element>

      <!-- 2.2. basic actions -->
      <xsd:element name="add" type="vdl:addActionType"/>
    </xsd:choice>
  </xsd:group>
</xsd:complexType>

```

```
<xsd:element name="put" type="vdl:addActionType"/>
<xsd:element name="del" type="vdl:basicActionType"/>

<!-- 2.3. sub-actions -->
<xsd:element ref="vdl:action"/>

</xsd:choice>
</xsd:group>
<xsd:complexType>
</xsd:element>

</xsd:schema>
```

Exemple d'action en XML

L'exemple suivant décrit un vase contenant une rose rouge et une action qui ajoute une rose blanche dans le vase. La partie structurale (la description des objets) dépasse le simple cadre de ce qui est imposé dans notre schéma *vdl.xsd* :

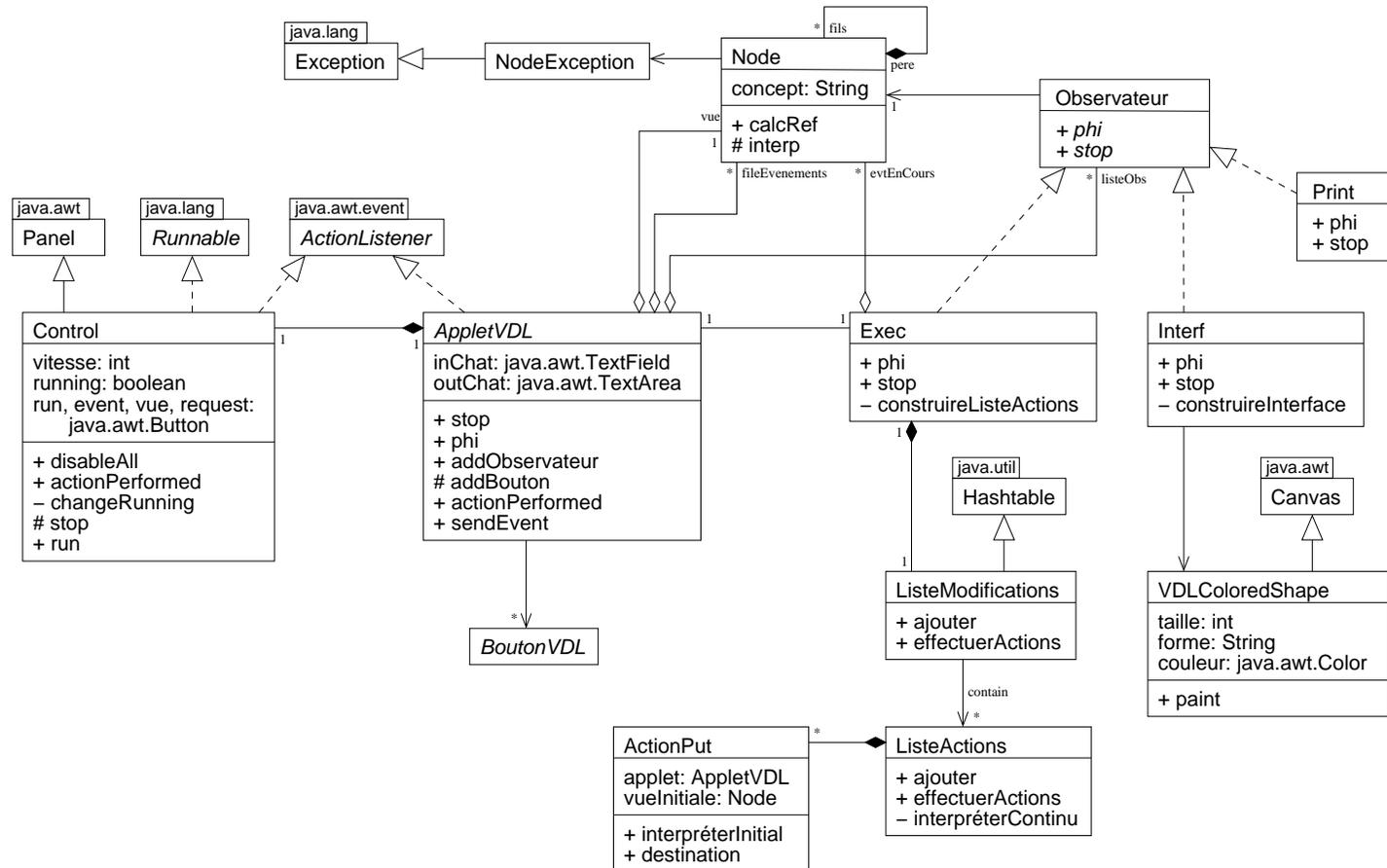
```
<?xml version="1.0"?>
<view xmlns:vdl="http://www.limsi.fr/Individu/nico/xml/vdl.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <object type="container" ref="blue_vase">
    <name>vase</name>
    <color>blue</color>
    <shape>cylinder</shape>
    <description>This is my preferred vase</description>
    <relation name="contains">
      <object ref="rose1">
        <name>rose</name>
        <color>red</color>
        <shape xsi:nil="true"/>
        <description>This is a red rose</description>
      </object>
    </relation>
  </object>
  <vdl:add>
    <vdl:path>
      //object[@ref="blue_vase"]//relation[@name="contains"]
    </vdl:path>
    <vdl:value>
      <object ref="rose2">
        <name>rose</name>
        <color>white</color>
        <shape xsi:nil="true"/>
        <description>This is a white rose</description>
      </object>
    </vdl:value>
  </vdl:add>
</view>
```

Coco en XML

```
<view xmlns:vdl="http://www.limsi.fr/Individu/nico/xml/vdl.xsd">
  <variable type="integer" name="value">0</variable>
  <variable type="integer" name="speed">1</variable>
  <variable type="boolean" name="running">true</variable>
  <action>
    <name>count</name>
    <guard><get>//variable[@name="running"]</get></guard>
    <put>
      <path>//variable[@name="value"]</path>
      <plus>
        <get>//variable[@name="value"]</get>
        <get>//variable[@name="speed"]</get>
      </plus>
    </put>
  </action>
  <action>
    <name>stop</name>
    <event><stop/></event>
    <put>
      <path>//variable[@name="running"]</path>
      <value>>false</value>
    </put>
  </action>
  <action>
    <name>start</name>
    <event><start/><go/></event>
    <put>
      <path>//variable[@name="running"]</path>
      <value>>false</value>
    </put>
  </action>
  <action>
    <name>speed up</name>
```

```
<event><faster/></event>
<put>
  <path>//variable[@name="speed"]</path>
  <plus>
    <get>//variable[@name="speed"]</get>
    <value>1</value>
  </plus>
</put>
</action>
<action>
  <name>slow down</name>
  <event><slower/></event>
  <put>
    <path>//variable[@name="speed"]</path>
    <plus>
      <get>//variable[@name="speed"]</get>
      <value>-1</value>
    </plus>
  </put>
</action>
</view>
```

Diagramme UML de l'interprète VDL



Code des composants Java

MikeCursor.java (les curseurs pour Mike)

```
import java.awt.* ;
import java.awt.event.* ;
import VDL.* ;

public class MikeCursor extends Panel implements AdjustmentListener
{
    private Scrollbar bar ;
    private AppletVDL applet ;
    private String name ;

    public MikeCursor(AppletVDL a, String n ,int minVal, int maxVal)
    {
        applet = a ;
        name = n ;
        setLayout(new GridLayout(1,1)) ;
        bar = new Scrollbar(Scrollbar.HORIZONTAL,minVal,0,minVal,maxVal) ;
        bar.addAdjustmentListener(this) ;
        add(bar) ;
    }

    public void adjustmentValueChanged(AdjustmentEvent e)
    { applet.sendEvent(name+"["+bar.getValue()+"]") ; }

}
```

MikeSwitch.java (les *checkbox* pour Mike)

```
import java.awt.* ;
import java.awt.event.* ;
import VDL.* ;
```

```

public class MikeSwitch extends Panel implements ItemListener
{
    private CheckboxGroup cbg = new CheckboxGroup();
    private Checkbox off, heat, grill, cook;
    private AppletVDL applet;

    public MikeSwitch(AppletVDL a)
    {
        applet = a;
        setLayout(new GridLayout(2,2));
        off = new Checkbox("off", cbg, true);
        off.addItemListener(this); add(off);
        grill = new Checkbox("grill", cbg, false);
        grill.addItemListener(this); add(grill);
        heat = new Checkbox("heat", cbg, false);
        heat.addItemListener(this); add(heat);
        cook = new Checkbox("cook", cbg, false);
        cook.addItemListener(this); add(cook);
    }

    public void reset()
    { cbg.setSelectedCheckbox(off); }

    public void itemStateChanged(ItemEvent e)
    { applet.sendEvent("function["+cbg.getSelectedCheckbox()
        .getLabel()+"]"); }
}

```

Code VDL de *Mike*

```
global [ private MikeSwitch func ;
         private MikeCursor temp ;
         private MikeCursor time ;

         public void init() {
             func = new MikeSwitch(this) ;
             temp = new MikeCursor(this,"temp",0,230) ;
             time = new MikeCursor(this,"time",0,120) ;
         }
]

view [
    interface [ H [ text [Temperature],
                 V [text [get [user,temperature]],
                   component [return temp ;]],
                 H [text [get [oven,temperature]], text [°C]]
                 ],
              H [ text [Time],
                 V [text [get [user,time]], component [return time ;]],
                 H [text [get [oven,time]], text [mn]]
                 ],
              H [ text [Function], component [return func ;],
                 button [start]
              ],
    ],

    variables [ user [temperature [0],time [0]],
               oven [temperature [19],time [0]],
               function [off], running [false] ],

    "external events" [
        "change oven temperature" [
            event ["oven's temperature"],
            put [path [oven,temperature],
               "event get" ["oven's temperature"]] ],
    ],
```

```

"change function"[
    event [function[grill],function[cook],function[heat]],
    put [path [function], "event get" [function]]
],
"change user temperature"[
    event [temp],
    put [path [user,temperature], "event get" [temp]]
],
"change user time"[
    event [time],
    put [path [user,time], "event get" [time]]
],
start[ event [clic[start],start],
    guard [not [equals [get [function],off]]],
    put [path [oven,time],0],
    put [path [mike],on],
    put [path [running],true]
],
stop[ event [function[off],stop],
    put [path [mike],off],
    put [path [running],false]
]
],

print[ mike[off] ],

actions[ guard[get[running]],

    grill[ guard [equals [get [function],grill]],
        put [path [oven,time],plus [get [oven,time],1]],
        stop[ guard ["greater than" [get [oven,time],
            get [user,time]]],
            put [path [function],off],
            put [path [mike],off],
            script [func.reset() ;],
            put [path [running],false] ]
    ],

    heat[ guard [equals [get [function],heat]],
        stop[ guard ["greater than" [get [oven,temperature],
            get [user,temperature]]],
            put [path [function],off],
            put [path [mike],off],
            script [func.reset() ;],

```

```

        put [path [running], false] ]
    ],
    cook[ guard [equals [get [function], cook]],
        put [path [oven, time], plus [get [oven, time], 1]],
        heat[ guard ["greater than" [get [user, temperature],
            get [oven, temperature]]],
            put [path [mike], on] ]
    cool[ guard ["greater than" [get [oven, temperature],
        get [user, temperature]]],
        put [path [mike], off] ]
    stop[ guard ["greater than" [get [oven, time],
        get [user, time]]],
        put [path [function], off],
        put [path [mike], off],
        script [func.reset() ;],
        put [path [running], false] ]
    ]
]
]

```

Index

- Acte (α), 104
- Actif, voir Composant
- Action
 - action de bon sens, 134
 - action élémentaire, 62, 63, 78, 241
 - action VDL, 36, 51, 62–64
 - composition, voir CSP
 - explication, 189
 - nom, 63, 162
 - objet, 169
 - possible, 172
 - recherche, 162, 223, 226
 - XML, 242
- Activité, 63, 105, 110, 157, 169, 214
- add, 62
- Agent
 - agent dialogique, 10
 - agent réflexif, 214
- Appariement
 - pattern, 139
 - terme, voir Subsumption
- Approche systématique, 21
- Argument, voir Relation
- Atome, 45
- Attribut, 45
- Auto-référence, 55

- Bon sens, 6, 7, 19–24, 28, 128
 - CHM, 10
 - connaissances de bon sens, 128
 - requête, voir CSR
- Bouton, 82, 85, 248
- But, voir Planification

- Cadre, voir Décors
- Calcul des situations, 16
- Canonique, voir Interprétation

- Causalité, 23
- Cause, voir Explication, 191, 208
 - modèle causal, 28, 193
 - planification, 27
- Changement, 14, 64, 133, 231
- Chronique, 30, 204, 206
 - compacte, 210
 - extensionnelle, 208
 - intensionnelle, 213
- Circonscription, 9
- Commande, 99, 116
- Commande (questions), 98
- component, 251, 253
- Comportement, 34, 63, 105, 153, 214
 - comportement collectif, 215–219
 - schéma comportemental, 204, 209, 216
- Composant actif, 10, 33, 236, 254
- concept, 60
- Concept VDL, 43
- Concurrence, 65–68
- Conséquence fixe, 176
- Contrôle (questions), 98
- CSP, 77, 244
- CSR, 131, 133–160
- Cycle d'exécution, 48, 208, 237, 247

- Définition d'un terme, 168
- Date, 181, 208, 220–222, 228
- Date (δ), 106
- Décidable, semi-décidable, 221
- Décors, voir Problème
- Défaut, voir Logique
- del, 62
- Diagnostic, 15, 27, 104, 191
- Dialogique, voir Agent
- Dynamique

- approche, 160, 206, 230
 - logique, voir PDL
 - modèle, 11, 35
 - observateur, 48, 50, 237, 247
- élémentaire, voir Action
- Émergent, voir Processus
- equals, 59
- État, 16, 27, 29, 38, 64, 104, 206, 222, 228
 - explication, 187
- Événement, 29, 30, 204
 - événement externe, 51, 62, 81, 99, 180, 242, 247
 - explication, 185, 190
 - file, 63
 - mémorisation, 161
 - Précondition, 164
 - réaction, voir Réaction
- event, 62
- event get, 57
- Exécution
 - modèle, 48, 50, 237
 - observateur, voir Observateur
- Explication, 14, 28, 104, 114, 178, 182, 185–191
- Extension, voir Intension
- figure, 85
- Flottant (terme), 46, 71, 241
- Fluent, 16, 21
- Garde, 23, 62, 164, 242
- get, 57
- global, 251
- greater than, 60
- Ground, 135–137
 - \mathcal{G}_p , 151
 - \mathcal{G}_{rp} , 159
 - \mathcal{G}_{rs} , 145
- guard, 62
- GUI, 5, 80, 238, 251
- Hypothèse d'inertie, 20
- IHM, 5
- Incomplétude (théorème), 55
- Inertie, voir Hypothèse
- Instance, voir Pattern
- Intension, 205
- Intensionnalisation, 206, 211–214
- Interface
 - see Observateur, 1
- Interprétation canonique, 56, 223, 241, 246
- Interprète VDL, 245
- InterViews, 33
- InterViews (projet), 10–14, 236, 249
- is a, 59
- Itérateur, 211
- Logique
 - des défauts, 9
 - des prédicats (LPPO), 7
 - dynamique, voir PDL
 - non monotone, 8
 - réifiée, 29
- Magicien d'Oz, 95
- MLN, 13, 64, 92, 250
- Modal, 50
- Modification
 - arbre des modifications, 223
 - liste des modifications, 66, 208
- Modifieur VDL, 47
- MRF, 14, 160
- node, 60
- Nœud VDL, 38, 45, 138, 224, 246
- Nom, voir Action
- Objet (ω), 106
- Objet (ω), 131, 161
- Objet d'action, voir Action
- Observateur, 47–50
 - exec, 50–51, 238, 247
 - interf, 80–81, 248
 - XML, 237
- Ordinaire, voir Utilisateur
- path, 62

Pattern, 137–141
 PDL, 17, 39
 Pertinence, voir Réponse
 Planification, 15, 24–27, 104, 114, 183
 Possibilité, 111, 172
 Possible, voir Action
 Possibilité, 64
 Postcondition, 17
 Précondition, 17, 63, 161, 189, 208, 242
 recherche, 163, 223, 227
 Proactif, 50, 204
 Problème
 de la qualification, 19
 des ramifications, 22, 191, 230
 du décors, 20
 Processus, 29, 64
 émergeant, 214, 230
 Progression, voir Régression
 put, 62, 79

 Question, 13, 92, 97

 Ramification, voir Problème
 Réactif, 50, 214
 Réaction, 64, 110
 réaction possible, 172
 réaction principale, 180
 Référence, 52, 223, 240, 246
 problème, 92, 165
 Regression, 26
 Relation
 argument, 152
 explication, 188
 procédurale, 134, 156–158, 170, 177, 179, 188
 statique, 134, 145, 187
 Réponse
 formelle, 13, 103
 naturelle, 128
 pertinente, 28, 95, 128, 161, 166, 185, 191, 193, 195
 Requête, 13, 92, 103–107, 132
 de bon sens, voir CSR
 multiple, 107

 Schéma
 comportemental, 204, 209, 214, 216
 XML, 238
 script, 251, 252
 Sémantique, 47, 65, 235
 explicationnelle, 35, 109, 128, 131
 formelle, 7, 21
 opérationnelle, 14, 35, 100, 131, 241
 web, voir Web sémantique
 Situation, 16, 208
 SMA, 14, 215
 Statique
 approche, 160, 191, 204, 230, 243
 observateur, 47, 49, 80, 237
 structure, 11, 34, 235, 243
 STRIPS, 25
 Structure, voir Statique
 Subsumption, 51
 Sujet (σ), 105, 175
 Sujet (σ), 167
 Systématique, voir Approche
 Systématique (approche), 21

 TALN, 2, 12, 92
 Terme VDL, 45
 flottant, voir Flottant
 Type (τ), 104
 Type (τ), 147

 Utilisateur ordinaire, 5, 10, 128, 147, 161, 236

 Valeur, voir variable
 Valide (précondition), 62, 164
 Variable, 64, 68, 207, 239
 VDL, 11–12, 34
 VDL procédural, 33
 view, 45
 Vue, 11, 37

 Web sémantique, 235