Transformation de programmes logiques : application à la personnalisation et à la personnification d'agents.

Georges Dubus

N° d'ordre : 2014-17-TH

SUPÉLEC

ÉCOLE DOCTORALE STITS

« Sciences et Technologies de l'Information des Télécommunications et des Systèmes »

THÈSE DE DOCTORAT

DOMAINE : STIC
Spécialité : Informatique

Soutenue le 4 septembre 2014

par

Georges DUBUS

Transformation de programmes logiques : application à la personnalisation et à la personnification d'agents

Directrice de thèse :	Yolaine BOURDA	Professeure à Supélec
Encadrant :	Fabrice POPINEAU	Professeur à Supélec

Composition du jury :

Rapporteurs :	Zahia GUESSOUM	Maître de Conférences HDR
	Abdel-Illah MOUADDIB	Professeur des Universités
Examinateurs :	Maxime MORGE	Maître de Conférences
	Nicolas SABOURET	Professeur des Universités

Remerciements

À Fabrice Popineau, pour son l'encadrement attentif et patient. Ses enseignements dans le domaine de l'intelligence artificielle, ainsi que sur bien d'autres sujets, ont été essentiels à l'accomplissement de cette thèse, et m'ont ouvert de nouveaux horizons scientifiques et techniques.

À Yolaine Bourda, qui a dirigé et encadré cette thèse en apportant son expertise des hypermédias adaptatifs, et qui a su m'aider à remettre le pied à l'étrier dans les moments de blocage.

À Jean-Paul Sansonnet, qui a permis d'élargir le domaine d'application de cette thèse en révélant les similitudes de la personnalisation et de la personnification.

À tous les membres du département Informatique de Supélec, qui m'ont accueilli au sein du laboratoire et de l'équipe enseignante, qui m'ont guidé et aidé autant dans la recherche que l'enseignement.

À Lucie Daeye, qui a su me soutenir et me supporter pendant ces 4 ans, autant dans les bons jours que les mauvais. À mon tour maintenant.

À mes parents, pour leur soutien constant pendant cette thèse et dans toutes les étapes qui y ont mené, et pour leurs nombreuses relectures de mon manuscrit, aidant ainsi à retrouver les régionalismes perdus.

À Zahia Guessoum et Abdel-Illah Mouaddib, qui ont accepté de rapporter cette thèse, et dont les commentaires ont permis de raffiner ces travaux et leur présentation.

À tous les membres du jury : Yolaine Bourda, Zahia Guessoum, Maxime Morge, Abdel-Illah Mouaddib, Fabrice Popineau et Nicolas Sabouret, d'avoir accepté d'assister à la présentation des mes travaux lors de la soutenance.

À toutes les personnes qui ont fait le déplacement, du bureau d'à côté ou de beaucoup plus loin, pour assister à ma soutenance.

À tous les autres, qui ont participé, aidé ou simplement été là à un moment ou un autre.

À tous, mon entière reconnaissance pour les contributions à ces travaux et pour l'aide dans la traversée de ces quatre années qui ont été, sans l'ombre d'un doute, les plus intéressantes jusque là.

Table des matières

Table des figures

Première partie

Introduction

Chapitre 1

Contexte et problématique

Chaque matin, avant de partir, je vérifie les horaires de RER. L'opération est toujours la même : j'allume mon téléphone, je lance l'application RATP, je sélectionne la ligne, puis la direction, puis l'arrêt, et j'attends l'affichage des résultats. Et chaque matin, je rentre à nouveau ces mêmes informations, sachant qu'elles ne seront retenues que le temps d'une requête.

La science-fiction a longtemps imaginé des technologies qui sont aujourd'hui réelles. Nous sommes nombreux à avoir dans notre poche un dispositif doté d'une grande puissance de calcul et d'un accès au réseau global. Cependant, l'assistant virtuel qui lui est traditionnellement associé n'y est pas. On en trouve certes une ébauche dans les applications phares des géants informatiques, mais ce ne sont guère plus que des preuves de concepts qui ne vont pas au-delà de la poignée de scénarios d'utilisation prévus. Le reste des usages, la grande majorité, ressemble à mon interaction quotidienne avec l'application RATP : l'interrogation quasi directe d'un serveur à travers une application qui ne fait guère plus que mettre en forme les résultats. La machine ne s'adapte pas à moi.

Et pourtant, cette adaptation est plus que jamais nécessaire. Nous perdons de plus en plus de temps à trouver des informations dans des corpus de plus en plus larges. Nous multiplions les interactions courtes avec des machines, perdant à chaque fois le temps requis pour rappeler qui nous sommes et ce que nous voulons. Nous avons besoin de personnalisation. La personnalisation, en tant que domaine de recherche, vise à répondre à ces problèmes en adaptant le comportement des systèmes en fonction de l'utilisateur.

L'assistant virtuel n'est cependant pas si loin. Il est l'objet de recherches, et il pourra puiser dans les technologies de la personnalisation l'intelligence nécessaire pour remplir son office. Mais pour jouer aussi bien son rôle qu'un véritable assistant, il faudra qu'il parvienne à faire oublier à l'utilisateur qu'il n'est que virtuel. C'est le but de la personnification, qui cherche à doter des agents intelligents de personnalités pour les rendre plus crédibles dans le cadre de l'interaction avec des humains.

À l'heure actuelle, la plupart des services avec lesquels nous interagissons sont, sous une forme ou une autre, des applications Web. Que ce soient des applications pour téléphones portables ou des sites Web, elles sont basées sur des serveurs interagissant avec de nombreux utilisateurs. Les applications Web tournant sur ces serveurs sont au cœur des systèmes modernes, et ce sont elles qui ont besoin de personnalisation et de personnification.

1.1 Problématique

Dans le cadre de la personnalisation et de la personnification, plusieurs problèmes sont ouverts, parmi lesquels :

— De nombreux travaux sur la personnalisation ont recours à des techniques liées à l'intelligence artificielle, telles que des techniques de raisonnement sur des ontologies, sur des actions, ou des techniques de planification. Cependant, ces outils ne sont utilisés que comme sous-systèmes d'applications conventionnelles. Il n'existe pas de formalisme visant à exprimer entièrement une application Web (personnalisée ou non) interagissant avec un utilisateur comme il existe actuellement des formalismes pour exprimer le comportement de robots interagissant avec leur environnement.

— Il existe de nombreuses techniques de personnalisation, mais aucune qui ne soit vraiment générale : chaque famille de systèmes personnalisés repose sur des techniques qui lui sont propres. Ainsi, un hypermédia adaptatif et un système de recommandation ont des objectifs différents et sont implémentés de façons différentes. De plus, à l'intérieur d'une même famille, chaque système apporte sa propre technique et son implémentation ad hoc. Il n'existe pas de technique générique d'expression de la personnalisation.

— La personnalisation n'est qu'un cas particulier d'altération [1] du comportement d'un système informatique en fonction du contexte. De fait, on peut se demander si l'on peut considérer les altérations de comportement des applications de manière générique, afin d'unifier les techniques de personnalisation ainsi que d'autres techniques d'altération de comportement telles que la personnification. Il serait intéressant de partir d'un système fonctionnel et non personnalisé et de le transformer pour le doter d'un comportement personnalisé.

Nous proposons des solutions pour résoudre ces trois problèmes dans la section suivante.

1.2 Approche

Pour répondre au problème de l'expression d'applications sous la forme d'agents, nous étudions les formalismes agents habituellement utilisés dans le contexte d'interactions avec des environnements changeants, puis nous nous interrogeons sur les spécificités des interactions dans le cadre d'applications Web. À l'aide de cette étude, nous proposons WAIG (Web Apps in Golog), un formalisme agents adapté à l'expression d'interactions sur le Web. WAIG permet l'expression d'applications Web sous forme d'agents intelligents, supportant ainsi l'utilisation de techniques de l'intelligence artificielle pour le Web. WAIG offre une base intéressante pour notre approche du second problème, car il permet de considérer les applications comme des agents rationnels classiques.

La personnalisation et la personnification sont deux techniques différentes, mais qui ont un point commun : la prise en compte d'un profil dans le comportement du système. Dans la personnalisation, le profil modélise certaines caractéristiques de l'utilisateur dans le but de le servir au mieux. Dans la personnification, le profil doit influencer l'agent dans sa manière de se comporter. Nous considérons ces deux techniques comme deux variantes d'une seule idée d'altération et cherchons à mettre au point une technique d'adaptation suffisamment générale pour s'appliquer aux deux domaines.

1. Nous entendons altération au sens étymologique de changement et non pas au sens négatif de dégradation.

La personnalisation et la personnification permettent à une application de se comporter différemment en fonction d'un contexte : en fonction d'un profil de l'utilisateur dans le cas de la personnalisation, ou en fonction d'un profil psychologique pour la personnification. Notre hypothèse de travail consiste à partir d'un comportement neutre, non personnalisé ou personnifié, de l'application, et étudier l'altération de manière constructive : qu'est-ce que la personnalisation ajoute, ou enlève, à un comportement existant ?

Il existe autant de manières d'exprimer le comportement d'applications qu'il existe de langages de programmation, il est de fait impossible de traiter la question pour toute forme d'expression du comportement. Nous étudions donc ces altérations de comportement dans le cadre de la famille de langages Golog, qui sont des langages logiques offrant des possibilités de preuves sur les exécutions. Nous étudions comment cette altération se traduit au niveau des programmes implémentant les applications. Nous apportons une définition constructive de l'altération dans le cadre d'applications WAIG, et nous introduisons une technique capable d'altérer automatiquement un programme existant à l'aide de paramètres fournis par le concepteur de l'application.

Cette approche constructive de la personnalisation et de la personnification permet d'envisager facilement la création d'outils permettant l'ajout d'un comportement personnalisé et/ou personnifié à une application existante n'en disposant pas.

1.3 Contributions

Cette thèse contient deux contributions principales :

WAIG : un formalisme agents basé sur Golog spécialement adapté à l'expression et l'implémentation d'applications Web sous forme d'agents ;

PAGE : un formalisme pour des agents paramétriques, qui se divise en deux parties ;

- **PAGE framework** : un cadre formel pour manipuler des programmes Golog, comprenant notamment un ensemble de transformations et une comparaison entre les programmes permettant de déterminer si un programme est plus approprié qu'un autre pour un profil donné ;
- **PAGE process** : un processus semi-automatique d'altération d'agents permettant d'introduire un comportement personnalisé ou personnifié dans un programme agent n'en contenant pas. Cette technique est indépendante du type d'application sur lequel elle est exécutée, et indépendante du type d'altération à introduire.

La suite du manuscrit est organisée comme suit : la partie 2 présente un état de l'art de la question de la personnalisation, puis de l'expression d'agents interagissant avec des humains, pour enfin s'intéresser à l'intersection de ces deux domaines : l'altération d'agents en fonction du contexte. Dans la partie II, nous présentons nos contributions : un formalisme agents adapté à la création d'applications Web interagissant avec des utilisateurs et une étude de la transformation des programmes agents pour leur faire prendre en compte le contexte, afin d'ajouter des comportements personnalisés. Nous présentons l'utilisation de cette transformation sur des applications concrètes dans la partie III, mettant ainsi en évidence la généricité de notre approche. Enfin, la partie IV conclut et étudie les perspectives de recherche que ces travaux ouvrent.

Chapitre 2

État de l'art

2.1 Personnalisation et personnification

Cette thèse s'intéresse à la personnalisation des applications Web. La personnalisation, ou adaptation, recouvre un ensemble de techniques et de réalités dans plusieurs domaines de recherche. On en trouve donc de multiples définitions, toutes teintées par le domaine dont elles sont issues.

> By adaptive hypermedia systems we mean all hypertext and hypermedia systems which reflect some features of the user in the user model and apply this model to adapt various visible aspects of the system to the user. [1]
>
> P. Brusilovsky [Bru96], communauté des hypermédias adaptatifs

> We define personalization as delivering to a group of individuals relevant information that is retrieved, transformed, and/or deduced from information sources. [2]
>
> W. Kim [Kim02], communauté de la recherche d'information

> Personalization, a special form of differentiation, is that a website can respond to a customer's unique and particular needs. [3]
>
> Y. Cao [CL07], communauté des systèmes de recommandation

De manière générale, la personnalisation consiste à faire se comporter un système différemment en fonction de l'utilisateur qui interagit avec lui. La personnalisation repose sur un profil de l'utilisateur qui contient un certain nombre d'informations sur l'utilisateur et qui est l'instanciation d'un modèle qui dépend de l'application. Il existe une multitude de manières de personnaliser, et une multitude de techniques pour implémenter ces personnalisations, réparties dans plusieurs domaines d'études.

[CFTJ09] propose une taxonomie des différentes formes de personnalisation et des techniques permettant la mise en œuvre de ces différentes formes. À la suite de [PPPS03], cette taxonomie distingue quatre classes de fonctionnalités de personnalisation .

1. Par système d'hypermédias adaptatifs, nous entendons tous les systèmes hypertextes et hypermédias qui reflètent certaines caractéristiques de l'utilisateur dans un modèle de l'utilisateur et utilisent ce modèle pour adapter divers aspects visibles du système à l'utilisateur.

2. Nous définissons la personnalisation comme le fait de fournir à un groupe d'individus des informations pertinentes qui sont récupérées, transformées ou déduites à partir de sources d'informations.

3. La personnalisation est une forme spéciale de différenciation dans laquelle un site Web répond aux besoins uniques et particuliers d'un utilisateur.

— La mémorisation, classe la plus simple et la plus répandue, consiste à stocker des informations sur l'utilisateur telles que son nom ou son historique de navigation et d'utiliser ces informations comme rappels des interactions passées. Cela consiste par exemple à appeler l'utilisateur par son nom, lui donner un accès rapide aux contenus précédemment visités, ou lui donner, ou non, accès à des ressources protégées. On retrouve la mémorisation comme base de tout système personnalisé, car savoir à quel utilisateur on a affaire est un prérequis à toute autre forme de personnalisation.
— Le conseil [4] consiste à assister l'utilisateur en lui fournissant des informations pertinentes en fonction de ses intérêts. C'est par exemple le cas de systèmes recommandant aux utilisateurs des liens ou des ressources en fonction de ce qu'ils ont visité précédemment [MCS00], ou guidant l'utilisateur à travers un ensemble de ressources pédagogiques en fournissant des liens vers des ressources conseillées.
— La *customisation* [5] consiste à utiliser les préférences de l'utilisateur pour personnaliser le contenu ou la structure d'une page Web. On peut par exemple, en fonction de l'utilisateur, changer la mise en page d'un portail, offrir un contenu plus ou moins succinct [Sch01], ajouter et supprimer des liens [CCL99], ou offrir des prix différents sur un site commercial [AV05].
— L'aide à la réalisation de tâches [6] provient des assistants personnels [MCF+94] (une catégorie de systèmes adaptatifs) et consiste à avoir un assistant personnel du côté du client [7] qui effectue des actions à la place de l'utilisateur pour faciliter l'accès à des informations pertinentes. C'est par exemple le cas de la personnalisation de requêtes, qui consiste à modifier les requêtes d'un utilisateur à un moteur de recherche en ajoutant des informations telles que le contexte de la tâche pour obtenir des résultats plus pertinents.

Exemple

Une illustration de cette taxonomie avec quelques exemples d'applications connues est donnée en table 2.1

On retrouve des techniques de personnalisation dans plusieurs communautés ayant des approches relativement différentes, parmi lesquelles :

— La communauté des systèmes de recommandation se concentre sur la question de la recommandation d'items à un utilisateur et s'intéresse à toutes les techniques, qu'elles soient logiques ou statistiques, permettant d'améliorer la recommandation.
— La communauté des hypermédias adaptatifs s'intéresse à la personnalisation de manière plus large, cherchant à adapter certains aspects des applications Web : la présentation, la navigation et le contenu.
— La communauté de la recherche d'information (*information retrieval*) qui, de manière générale, étudie la manière de retrouver des informations dans un corpus, et notamment comment fournir les informations les plus pertinentes pour un utilisateur donné.

4. En anglais : *guidance*.
5. L'anglais *customization* se traduit par «personnalisation», «fabrication sur mesure» ou éventuellement l'anglicisme «*customisation*». Nous préférons ce dernier pour éviter l'ambiguïté.
6. En anglais : *task performance support*.
7. Côté client en terme d'architecture réseau.

	Mémorisation	Guidage	*Customisation*	Aide à la performance
Wikipédia	Se souvient de l'utilisateur A un historique de ses modifications		L'utilisateur peut définir une feuille de style personnalisée	
Hypermedia adaptatif pédagogique	Se souvient de l'utilisateur	Fournit des ressources pédagogiques appropriées	Adapte le contenu des ressources au niveau de connaissance de l'utilisateur	
Système de recommandation d'articles	Se souvient de l'utilisateur	Présente des articles à partir de ceux que l'utilisateur a aimés.		
Google	Historique des pages visitées	Adapte les résultats de recherche à l'utilisateur		Suggère des complétions de requêtes en fonction de l'utilisateur

TABLE 2.1 – Classification de systèmes personnalisés

Ces techniques sont détaillées ci-dessous.

En plus de la personnalisation, ce mémoire s'intéresse également à la personnification dont les techniques s'apparentent à celles de la personnalisation, bien que dans des buts différents. La personnification est présentée plus loin, à la section 2.1.4 tandis que les techniques sont étudiées à la section 2.3.3.

2.1.1 Systèmes de recommandation

Les systèmes de recommandation ont pour l'objectif principal de recommander des items à des utilisateurs. Nous présentons ici quelques-unes des approches de la recommandation.

- Les approches basées sur le contenu [Lan95, PB07] recommandent à l'utilisateur des items proches de ceux qu'il a appréciés. Cela requiert une description de chaque item et un moyen d'établir la proximité de deux items, par exemple en cherchant des points communs dans des mots-clefs décrivant les items. Il existe également des approches sémantiques [BFPAGS+08], dans lesquelles les items ne sont pas décrits par des mots-clefs, mais au sein des ontologies : plutôt que de recommander les items maximisant une mesure de similarité, ces approches cherchent les liens sémantiques entre les items appréciés par l'utilisateur et des candidats à la recommandation.
- Les approches collaboratives utilisent les notes attribuées, explicitement ou implicitement, par tous les utilisateurs aux items pour recommander des items. Il en existe deux variantes. Le filtrage basé sur les utilisateurs [RIS+94] cherche à rapprocher l'utilisateur d'autres utilisateurs ayant noté les mêmes items de manière similaire et à recommander les items appréciés par ces utilisateurs. Le filtrage basé sur les items [SKKR01] cherche à exploiter les notations des items pour établir des similarités entre items et proposer à l'utilisateur des items similaires à ceux qu'il a appréciés.
- Les approches démographiques [Paz99] cherchent à catégoriser les utilisateurs dans des classes à partir d'informations démographiques telles que leur âge, intérêts ou lieu d'habitation, puis à recommander des items à l'utilisateur en fonction de la classe dont il fait partie.
- Les approches basées sur la connaissance [Bur99] reposent sur un ensemble de règles qui, définies par un expert, sont utilisées pour déterminer les items à recommander en fonction de critères fournis par l'utilisateur. Le système peut éventuellement interroger l'utilisateur pour avoir des critères plus précis et affiner la recommandation.
- Il existe enfin diverses approches hybrides [MJZ03] qui réunissent différentes approches.
  - L'hybridation pondérée consiste à utiliser la somme pondérée des scores d'un item dans plusieurs approches pour déterminer le score d'un item, et garder les items ayant le plus grand score.
  - L'hybridation à bascule consiste à utiliser une approche ou une autre en fonction d'un critère.
  - L'hybridation mixée consiste à présenter ensemble les résultats de différentes approches.
  - L'hybridation par combinaison de caractéristiques consiste à utiliser les caractéristiques d'une approche dans une autre approche.
  - L'hybridation en cascade consiste à utiliser une approche pour filtrer un ensemble d'items recommandés par une autre approche.

— L'hybridation par ajout de caractéristiques consiste à utiliser le résultat d'une approche comme paramètre dans une autre approche.
— L'hybridation métaniveau consiste à utiliser une approche de personnalisation pour générer un modèle, et à utiliser une autre approche dans ce modèle plutôt qu'avec les données originales.

Le choix de l'approche de recommandation est une tâche complexe qui dépend de nombreux paramètres, tels que le domaine et le nombre d'utilisateurs et d'items que le système devra gérer. Un système donné suit une seule approche (éventuellement hybride) et résout un problème de recommandation, sans chercher la généralité. La majorité des systèmes de recommandation ont des approches numériques et n'utilisent pas de techniques de raisonnement. Seuls les systèmes basés sur le contenu utilisant une approche sémantique peuvent faire appel à des techniques de raisonnement dans la manipulation des informations sémantiques.

2.1.2 Hypermédias adaptatifs

Un hypermédia est un ensemble de ressources (texte, vidéo, audio, etc.) connectées par des liens. Chaque nœud contient une certaine quantité d'information et des liens vers d'autres nœuds. La définition la plus largement répandue d'hypermédia adaptatif est celle de Brusilovsky [Bru96] :

> Par système d'hypermédias adaptatifs, nous entendons tous les systèmes hypertextes et hypermédias qui reflètent certaines caractéristiques de l'utilisateur dans un modèle de l'utilisateur et utilisent ce modèle pour adapter divers aspects visibles du système à l'utilisateur.

Ces systèmes sont avantageux dans le cas où un seul système est utilisé par plusieurs utilisateurs ayant différents buts, connaissances et expériences, et où le corpus sous-jacent est suffisamment grand. Ces systèmes sont une tentative de réponse au problème «d'égarement dans l'hyperespace» : les buts et connaissances de l'utilisateur peuvent être utilisés pour modifier les liens disponibles dans un système hypermédia.

Techniques d'adaptation

Il existe de nombreuses techniques d'adaptation, regroupées en trois catégories [Bru96] : les techniques d'adaptation de contenu, de parcours, et de présentation.

L'adaptation du contenu se fait principalement de deux manières : en montrant ou ne montrant pas certaines informations, ou bien en accentuant ou en estompant certaines informations [KBP09]. Dans le premier cas, on donne ou non l'information à l'utilisateur alors que dans le second, on ne fait que lui suggérer ou non de lire. Les informations en question peuvent être des explications supplémentaires ou des variantes d'explications en fonction des buts, intérêts tâches ou connaissances de l'utilisateur [Bru92]. Par exemple, le système peut fournir des explications sur des notions prérequises à la compréhension d'une page que l'utilisateur ne maîtriserait pas [FMRR90], ou des explications comparatives mettant en relation les concepts présentés avec des concepts connus de l'utilisateur [FMRR90]. Cela peut se faire par la modification du contenu lui-même, ou par des techniques d'adaptation de la présentation, comme l'atténuation de parties du contenu, le tri de contenus en mettant ceux à lire en

premier, le zoom [TS03] (le contenu est réduit et peut être agrandi par l'utilisateur) ou le déroulement de texte (idem, mais le texte est résumé plutôt que réduit visuellement).

Comme le contenu, la navigation peut être influencée de deux manières : par la force, ou par la suggestion. Les techniques de guidage direct sélectionnent les liens adaptés pour un utilisateur et masquent les autres, offrant ainsi une structure de navigation plus contrainte et forçant l'utilisateur à suivre un chemin [FNW98]. La plupart des travaux sur l'adaptation de navigation cherchent à suggérer des chemins de navigation à l'utilisateur plutôt qu'à le contraindre, par exemple en triant une liste de liens par pertinence pour l'utilisateur, ou en annotant les liens avec une couleur ou une icône [BSW96] (métaphore du feu tricolore : les liens pertinents sont en vert, les liens moins pertinents sont en orange, et les liens déconseillés sont en rouge).

L'adaptation de la présentation peut être faite pour mettre en valeur ou non certaines parties du contenu ou certains liens, devenant ainsi un instrument de l'adaptation de contenu ou de navigation. Elle peut aussi se faire au nom de la présentation elle-même, en fournissant une mise en page adaptée à l'utilisateur ou au périphérique sur lequel il consulte le système (téléphone, tablette ou fixe).

Modèle de référence

Le modèle de référence d'hypermédia adaptatif [Bru96] considère que l'application est basée sur trois principes.

1. L'application est basée sur un modèle du domaine qui décrit comment la représentation conceptuelle du domaine de l'application est structurée. Ce modèle indique les relations entre les concepts, et comment ils sont connectés aux contenus à présenter (les pages ou fragments de pages).
2. Un modèle de l'utilisateur maintenu à jour représente les connaissances, intérêts, buts, objectifs, historique d'interaction de l'utilisateur, ou toute autre information pouvant être utile à l'adaptation.
3. Un modèle de l'adaptation exprime comment la présentation, le contenu et la navigation sont adaptés aux informations contenues dans le modèle de l'utilisateur. Ce modèle de l'adaptation met en relation le modèle du domaine et le modèle de l'utilisateur et exprime comment l'adaptation doit être générée à partir de ces derniers. La question de l'adaptation est traitée plus amplement dans la section suivante.

AHAM [WHDB98, DBHW99] est un des modèles de référence pour la création d'hypermédias adaptatifs. Le domaine y est représenté comme un ensemble de concepts ayant entre eux des relations (la nature des relations dépend du domaine, celles-ci peuvent être par exemple des relations de prérequis ou de composition) ; à ces concepts sont rattachées des pages composées de fragments. Le modèle de l'utilisateur est défini par un ensemble d'attributs rattachés aux concepts : chaque attribut a une valeur pour chaque concept et pour chaque utilisateur. Le modèle de l'adaptation est un ensemble de règles définissant comment les concepts, les relations et les attributs sont reliés. Par exemple, une règle peut spécifier que dans le cadre d'une relation de prérequis entre $C1$ et $C2$, pour un utilisateur donné, l'attribut $C2.readyToRead$ est vrai si l'attribut $C1.read$ est vrai. Le modèle ne spécifie pas comment construire les pages.

Comment exprimer l'adaptation ?

Le plus souvent, l'adaptation dans les systèmes d'hypermédias adaptatifs est exprimée à l'aide de règles de la forme condition-action ou évènement-condition-action [WHDB98]. Ces règles peuvent aussi être des formules de logique du premier ordre, utilisant alors un raisonnement pour l'adaptation [HDN04]. GLAM [Jac06] propose l'utilisation de règles en logique du premier ordre reliées à différents stéréotypes de profils, avec des métarègles pour déterminer quelles règles utiliser en cas de conflit entre règles. L'expression de l'adaptation par règles étant fastidieuse et difficile pour des intervenants non techniciens, des techniques d'aide à l'écriture des règles ont été mises au point. Des éditeurs graphiques de règles [DBSS05] permettent l'application de patrons prédéfinis d'adaptation à un modèle donné. Des systèmes de patrons d'adaptation permettent au concepteur d'application de spécialiser des modèles génériques d'adaptation à ses propres modèles [ZBR+09]. Des langages d'adaptations tels que LAG [CV04] ou GAL [VDSHLH09] offrent une alternative de plus haut niveau à l'écriture de règles, en offrant des constructions permettant d'exprimer directement certaines techniques d'adaptation.

Cependant, ces techniques d'expression de l'adaptation ne permettent qu'une adaptation ad hoc, c'est à dire spécifique à un modèle du domaine et de l'utilisateur donné. Ces outils sont le plus souvent des langages facilitant l'expression de l'adaptation, mais demandent toujours au concepteur d'application de coder l'intégralité de la logique d'adaptation de l'application.

2.1.3 Personnalisation et Web sémantique

Au cours de ces dernières années, de nombreux travaux se sont intéressés à l'utilisation des technologies du Web sémantique comme base pour la personnalisation [BBH05]. Le Web sémantique offre des outils permettant de représenter des connaissances sous la forme de triplets *sujet-prédicat-objet* (RDF [BM04]) : chaque triplet est un prédicat binaire. Ces prédicats représentent des faits logiques. OWL [G+09] est un langage de représentation de connaissance construit sur RDF, permettant de fixer des modèles pouvant notamment être utilisés pour exprimer le profil de l'utilisateur et le domaine de l'application. Des techniques de raisonnement [FU10] peuvent manipuler ces connaissances, offrant ainsi une base pour exprimer l'adaptation : SWRL [HPSB+04] offre un langage de règles permettant l'inférence et SPARQL [PS+08] est un langage de requêtes permettant de sélectionner des ressources ou des triplets, en offrant également certains types d'inférence.

[ABB+04] se penche sur la question de l'outillage de la couche logique du Web sémantique, notamment pour supporter la personnalisation, et propose une architecture pour les services web sémantiques personnalisés. Les outils utilisés sont principalement des langages de règles (tels que SWRL) permettant de déduire de nouvelles connaissances à partir d'une base de connaissance et de règles, et des langages de requêtes et de transformations, permettant d'extraire des connaissances d'une base. L'importance du raisonnement sur des actions pour la personnalisation y est soulignée.

Plusieurs travaux utilisent des systèmes à base de règles pour faire du raisonnement et introduisent le profil de l'utilisateur dans les règles pour fournir un comportement personnalisé. Une étude de l'utilisation des technologies du web sémantique pour l'adaptation [DHNS03] propose l'utilisation d'ontologies exprimées dans un langage du Web sémantique pour la représentation du domaine d'une application et du profil de l'utilisateur, et du langage de règles TRIPLE pour le raisonnement. Dans l'exemple fourni, des ressources pédagogiques sont représentées sous forme de triplets RDF et des règles TRIPLE sont utilisées pour déterminer les ressources appropriées pour un utilisateur donné. Les

requêtes sur les ressources dépendent de l'utilisateur, ce qui permet de donner des résultats différents à des utilisateurs différents. [HDN04] et [DHN03] étendent cette idée en représentant explicitement les caractéristiques des systèmes hypertextes : les interactions avec l'utilisateur et les structures hypertextes ont leurs propres ontologies, ce qui permet de raisonner sur l'historique des interactions avec l'utilisateur dans les règles et la génération des pages à renvoyer à l'utilisateur.

2.1.4 Personnification

L'idée de la personnification, apparue avec [RH96], consiste à donner à des agents rationnels des traits de personnalité ou des traits psychologiques, de manière à leur donner des comportements moins prévisibles et plus humains. D'après [Hay04], la modification du comportement d'un agent pour simuler des caractéristiques psychologiques telles que des traits de personnalités donne à des interlocuteurs humains l'impression que l'agent est vivant, augmentant son facteur d'acceptation [8] [Dav89]. Cela se rapproche de la notion d'agents crédibles, développée par [Bat94], qui sont des personnages crédibles plutôt que de simples interfaces.

La personnification a des applications dans de nombreux domaines tels que les *serious games* [VdBBMH12], où la présence de personnages avec des personnalités permet de rendre l'expérience plus réaliste et plus diverse, ou les soins médicaux [BVJPO13], où l'utilisation d'agents virtuels améliore l'acceptation du personnel médical face à l'introduction d'automatisation informatique dans leur travail.

La personnification est comparable à la personnalisation dans le sens où elle consiste à modifier le comportement d'un système en fonction d'un profil : le profil psychologique du personnage à jouer.

2.1.5 Conclusion

La personnalisation est présente dans plusieurs domaines et techniques, et le terme désigne des choses différentes pour des communautés différentes. Selon les systèmes, l'interprétation de ce que signifie la personnalisation diffère, et il n'existe pas de définition de la personnalisation qui englobe tous les usages et qui soit suffisamment précise pour l'étude formelle que nous souhaitons mener.

De plus, les systèmes personnalisés sont généralement construits directement pour offrir de la personnalisation, et bien souvent de manière ad hoc, plutôt que d'être conçus comme une évolution de systèmes n'offrant pas de personnalisation. Une telle approche aurait pourtant l'avantage de permettre d'offrir de la personnalisation dans des systèmes n'étant pas initialement prévus pour en offrir, ouvrant ainsi une nouvelle voie pour l'aide à la construction de systèmes personnalisés.

La personnification cherche aussi à modifier un système en fonction d'un profil, en modélisant non plus l'utilisateur, mais la manière dont l'agent devrait se comporter. Une définition de l'adaptation suffisamment large pourrait, en ne spécifiant pas le contenu des profils, s'appliquer autant à la personnalisation qu'à la personnification.

8. La propension qu'a un utilisateur à accepter l'utilisation d'un système.

2.2 Expression logique des agents interagissant avec des humains

Plusieurs parties de la personnalisation sont liées à l'intelligence artificielle : le Web sémantique qui peut être utilisé pour la modélisation du domaine ou de l'utilisateur a des racines dans l'intelligence artificielle (notamment dans les logiques de description) et les systèmes de règles utilisés par de nombreux systèmes d'hypermédias adaptatifs sont des sous-ensembles de la logique des prédicats d'ordre 1. Nous avons choisi d'étudier la personnalisation avec les outils de l'intelligence artificielle. La modélisation logique d'applications Web personnalisées offre de nouvelles possibilités, telles que la preuve automatisée de propriétés de l'application.

De fait, nous nous plaçons dans le cadre de l'architecture classique de l'intelligence artificielle, celle d'un agent interagissant avec un environnement, telle que décrite dans [RN10]. De manière générale, un agent agit sur un environnement et reçoit en retour des perceptions de cet environnement dans le but d'accomplir une certaine tâche. Dans notre cas, l'agent est une application Web qui interagit avec un utilisateur dans le but d'accomplir une certaine tâche pour l'utilisateur. La personnalisation consiste ici à prendre en compte les particularités de l'utilisateur pour accomplir la tâche le mieux possible.

Dans le cadre de l'intelligence artificielle, l'interaction avec des humains est un problème complexe, car les humains sont imprévisibles et difficiles à modéliser, bien loin de l'environnement prévisible et entièrement connu qui forme le cas le plus simple d'environnement. Dans cette section, nous étudions les différentes approches de la modélisation et de l'expression d'agents interagissant avec des humains.

2.2.1 Le problème du sensing

Un agent interagissant avec un environnement peut ne pas tout savoir de l'environnement et des conséquences de ses actions, et donc ne pas pouvoir faire de plans sans prendre en compte des éléments extérieurs : ce problème est appelé problème du sensing. Il est commun à de nombreuses théories d'agents. L'interaction avec des humains pose le problème du sensing : en interagissant avec un utilisateur, l'agent ne peut prévoir sa réponse, et doit donc agir avec une certaine incertitude (il ne peut choisir avec certitude une marche à suivre pour l'intégralité de son exécution). Ici, nous présentons rapidement le problème et quelques-unes de ses solutions. Ces solutions seront présentées plus en détail dans les descriptions des formalismes dans les sections suivantes.

Les formalismes d'actions sont des formalismes permettant de modéliser un univers en définissant des actions, leurs préconditions et leurs effets [9]. Dans ces formalismes, l'univers évolue uniquement par les actions : une action fait passer l'univers d'un état à un autre, et un état peut être défini comme la séquence d'actions qui y a mené à partir d'un état initial. Dans la plupart des formalismes, les définitions des effets des actions sont déterministes : l'application d'une action dans un état précis a toujours les mêmes effets. De fait, de tels formalismes ne peuvent pas exprimer des agents ne connaissant pas les effets de leurs actions, ou ne connaissant qu'une partie de leur environnement.

Une des solutions à ce problème implique la notion de connaissance [Moo79] : il s'agit de faire la différence entre ce qui est vrai dans l'environnement et ce que l'agent sait : il peut y avoir des faits vrais que l'agent ne sait pas encore. Une des formalisations de cette solution est donnée en 2.2.3. La

9. Le calcul des situations, présenté plus loin, est un exemple de formalisme d'action

connaissance de l'agent y est définie dans les mêmes termes que l'environnement, sous la forme d'un fluent [10] spécial nommé fluent épistémique. De même, l'influence des actions sur la connaissance est définie de la même manière que l'influence des actions sur l'environnement. Les actions qui influent sur la connaissance sont appelées actions de *sensing*

Selon les formalismes, deux formes de modifications de la connaissance sont possibles : la mise à jour de la connaissance (*knowledge update*) et l'extension de la connaissance (*knowledge extension*). La mise à jour de la connaissance se produit dans le cas où un fait est oublié au profit d'un autre qui le remplace (par exemple : «la porte est maintenant ouverte, alors qu'elle était fermée auparavant»). L'extension de la connaissance se produit quand un nouveau fait est connu sans en invalider un autre (par exemple : «je découvre que la porte est ouverte, sans connaître son état précédent»). Certains formalismes ne permettent que l'extension de la connaissance, tandis que d'autres permettent les deux formes.

2.2.2 Agents en Golog

Nous présentons ici le langage d'agent Golog, ses successeurs, ConGolog et IndiGolog, ainsi que la théorie logique sur laquelle ils se basent, le calcul des situations. Golog permet d'exprimer des agents interagissant avec un environnement à l'aide d'actions qui peuvent être composées pour créer un programme complexe. Golog a déjà été utilisé dans le domaine de la personnalisation pour exprimer des hypermédias adaptatifs [JBP+06].

Calcul des situations

Le calcul des situations [MH68] est un ensemble de formules du premier ordre conçu pour représenter des mondes changeant dynamiquement. Tous les changements sont le résultat *d'actions* nommées. Un état du monde est défini par l'historique des actions qui y ont mené et est représenté par un terme du premier ordre nommé *situation*. La constante s_0 représente la *situation initiale*, dans laquelle aucune action n'a eu lieu. Le symbole fonctionnel *do* est utilisé pour représenter l'application d'une action dans une situation : $do(a, s)$ est la situation obtenue par l'application de l'action a dans la situation s. De fait, toute situation peut être écrite sous la forme

$$do(a_n, do(a_{n-1}, \ldots do(a_0, s_0)\ldots)),$$

où s_0 est la situation initiale et $a_0 \ldots a_n$ est l'ensemble des actions ayant été exécutées pour arriver dans la situation en question.

Les prédicats et fonctions dont la valeur dépend de la situation sont appelés *fluents*. Par convention, l'argument de situation est placé en dernier. Les prédicats qui dépendent de la situation sont appelés *fluents relationnels*, tandis que les fonctions qui dépendent de la situation sont appelées *fluents fonctionnels*. Par exemple, $holding(x, s)$ pourrait indiquer qu'un agent robot porte un objet x dans une situation s. La valeur d'un fluent est définie par sa valeur dans la situation initiale et par la manière dont elle est affectée par chaque action. Chaque situation étant le résultat d'une suite d'actions à partir de la situation initiale, il est possible d'obtenir la valeur d'un fluent dans une situation en remontant la

10. Un fluent est un prédicat qui dépend de l'état. L'environnement est défini par un ensemble de fluents, permettant de connaître les valeurs des différentes choses modélisées.

suite d'actions pour obtenir la valeur du fluent en fonction de sa valeur dans la situation initiale ; cette opération est appelée *régression*.

Chaque action est définie par deux éléments : ses préconditions et ses effets. Les préconditions sont définies à l'aide d'un prédicat spécial $Poss(a, s)$ qui indique si une action a est possible dans une situation s.

Exemple

La formule

$$\forall x\ Poss(pick(x), s) \equiv \neg holding(x, s) \wedge near(x, s)$$

indique que l'action $pick(x)$ est possible si le robot ne porte pas déjà x et que x est suffisamment proche.

Les effets d'une action sont indiqués par un ensemble d'axiomes indiquant l'effet de chaque action sur chaque fluent.

Exemple

La formule

$$fragile(x) \implies broken(x, do(drop(x), s))$$

exprime que lâcher un objet fragile le casse.

Formellement, cette manière d'exprimer les effets des actions exige d'exprimer également les effets que les actions n'ont pas, comme le fait que lâcher un objet ne change pas sa couleur, sans quoi la régression devient impossible. Cela conduit à $2 \times A \times F$ axiomes, où A est le nombre d'actions et F le nombre d'axiomes, ce qui rend l'expression des effets des actions fastidieux et le raisonnement inefficace. On appelle ce problème *frame problem*, ou problème de la persistance. Une solution à ce problème [Rei91] consiste à regrouper ces effets en un axiome de l'état successeur par fluent énumérant les actions changeant la valeur du fluent et explicitant que les autres actions n'ont pas d'effet.

Exemple

Pour le fluent *cassé*, la formule

$$\begin{aligned} broken(x, (do(a, s))) \equiv\ & a = drop(x) \wedge fragile(x) \\ & \vee \exists b\ a = explode(b) \wedge near(a, b, x) \\ & \vee broken(x, a) \wedge a \neq fix(x) \end{aligned}$$

exprime que lâcher un objet fragile le casse, qu'un objet est cassé par un autre objet explosant à proximité, qu'un objet n'est plus cassé après avoir été réparé, et qu'aucune autre action n'affecte le fluent *broken*.

En général, les implémentations du calcul des situations demandent de n'exprimer que les cas dans lesquels les actions affectent les valeurs des fluents et considèrent que les valeurs ne changent pas dans les autres cas.

Un domaine d'application est modélisé par une *théorie de l'action* composée des éléments suivants :

— des axiomes décrivant la situation initiale s_0 ;
— un axiome de précondition d'action par action a, exprimant $Poss(a, s)$;
— un axiome de l'état successeur par fluent F exprimant dans quelles conditions $F(\vec{x}, do(a, s))$ est vrai en fonction de la situation s ;
— des axiomes explicitant que différents noms d'action représentent différentes actions ;
— des axiomes fondamentaux, indépendants du domaine.

Le calcul des situations seul permet d'exprimer un domaine dans le but de faire de la planification. Un problème de planification consiste à trouver une séquence d'action menant à une situation s dans laquelle certaines propriétés voulues ($\phi(s)$) sont vérifiées. Formellement, il s'agit de trouver, pour une théorie de l'action $\mathscr{D}$, une séquence d'action $\vec{a} = [a_1, \ldots, a_n]$ telle que :

$$\mathscr{D} \models Legal(\vec{a}, s_0) \wedge \phi(do(\vec{a}, s_0))$$

où $do(\vec{a}, s)$ est une abréviation de

$$do(a_n, do(a_{n-1}, \ldots, do(a_1, s)\ldots)),$$

$Legal(\vec{a}, s)$ est une abréviation de

$$Poss(a_1, s) \wedge \ldots \wedge Poss(a_n, do([a_1, \ldots, a_{n-1}], s))$$

et ϕ est une formule qui doit être vraie dans l'état final.

Golog

Golog [LRL+97] est un langage de programmation logique dont les actions primitives sont issues d'une théorie de l'action du calcul des situations. Il permet la construction de programmes complexes présentant des choix non-déterministes.

La présence de constructions non-déterministes dans un programme Golog mène à de multiples exécutions possibles pour le programme (une exécution pour chaque valeur possible du choix). Lors de l'exécution, chaque branche de chaque choix est évaluée de manière à trouver une exécution complète du programme. Ces constructions permettent au programme Golog de définir une exécution dans les grandes lignes en laissant l'interpréteur chercher une résolution appropriée du non déterminisme.

Les constructions suivantes sont disponibles en Golog. δ représente un programme Golog.

— a, une action primitive (également appelée action) issue d'une théorie de l'action. Dans un programme Golog basé sur la théorie de l'action évoquée dans la section précédent, $pick(x)$ est une action primitive.
— $\phi?$, un test : le programme échoue si ϕ est faux dans la situation courante [11]. Puisque l'interpréteur Golog évalue plusieurs branches possibles d'exécution, $\phi?$ permet de rejeter toutes les branches où ϕ est faux.

11. ϕ est une expression pseudo-fluent, c'est-à-dire une formule du calcul des situations où tous les arguments de situations ont été supprimés. L'expression $\phi[s]$ désigne la formule du calcul des situations obtenue en remettant s à la place de la variable de situation supprimée dans tous les fluents de ϕ.

— $(\delta_1;\delta_2)$, une séquence : δ_1 est exécuté, puis δ_2.

— $(\delta_1 \mid \delta_2)$, un choix non-déterministe de programme : l'un ou l'autre de δ_1 et δ_2 est exécuté. Cette construction permet de donner plusieurs alternatives à l'agent, ce dernier choisissant à l'exécution une alternative permettant d'accomplir son but. Par exemple, à l'exécution du programme $(\mathit{fix}(x) \mid \mathit{nil});\neg \mathit{broken}(x)?$ [12] l'agent répare x, ou ne fait rien, de manière à ce que x ne soit pas cassé, c'est-à-dire que l'agent répare x si nécessaire. L'utilisation de $\phi?$ avec un choix non-déterministe de programme permet de contraindre les choix : par exemple «si ϕ alors δ_1 sinon δ_2» s'écrit $(\phi?;\delta_1) \mid (\neg\phi?;\delta_2)$.

— $\pi v.\delta$, un choix non-déterministe d'argument : δ est exécuté pour une certaine valeur de v, où v est une variable libre dans δ. Par exemple, $\pi p.\mathit{goTo}(p);\mathit{pick}(x)$ permet d'exprimer que l'agent doit choisir une position p à laquelle se rendre pour pouvoir ramasser x, c'est-à-dire que l'agent doit choisir la position de x. L'utilisation de $\phi?$ avec le choix non-déterministe d'argument permet de contraindre le domaine sur lequel se fait le choix : $\pi v\, \phi(v)?;\delta(v)$ exécute $\delta(v)$ pour un v tel que $\phi(v)$. Par exemple, pour choisir une pelle à ramasser, on peut écrire $\pi x\, \mathit{shovel}(x);\mathit{pick}(x)$.

— δ^*, une itération non-déterministe : δ est exécuté un nombre non-déterministe de fois. Par exemple pour exprimer qu'il faut continuer à détruire des objets jusqu'à ce que tout soit cassé, on peut écrire $(\pi x\, \mathit{item}(x)?;\mathit{destroy}(x))^*;(\forall x\, \mathit{item}(x) \implies \mathit{broken}(x))?$.

— **proc** $P(\vec{v})\delta$ **endProc**, une procédure P qui pourra être appelée dans le reste du programme.

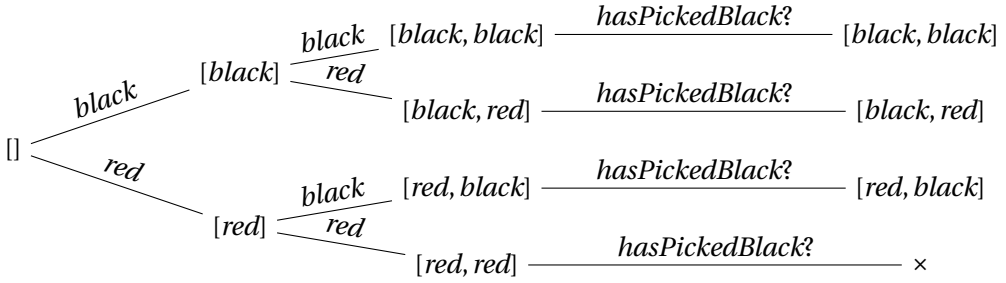

FIGURE 2.1 – Branches non-déterministes d'un agent tirant des cartes

Les situations sont représentées comme des séquences d'action primitives $[a_0,\ldots,a_n]$, sur les arêtes apparaissent le morceau de programme menant à la situation suivante. Le symbole × indique une situation inaccessible (un test ayant échoué).

Exemple

1. Pour illustrer le fonctionnement du non déterminisme, prenons un agent qui tire des cartes. Il peut soit tirer une carte rouge (*red*), soit tirer une carte noire (*black*). Le fluent *hasPickedBlack* est vrai si au moins une carte noire a été tirée. Le programme est le suivant :

$$(\mathit{red} \mid \mathit{black});(\mathit{red} \mid \mathit{black});\mathit{hasPickedBlack}?$$

12. *nil* est l'action «ne rien faire», qui n'a aucune précondition et aucun effet

Exemple (cont)

l'agent tire deux cartes et doit avoir tiré au moins une carte noire pour que le programme se termine. La figure 2.1 illustre les différents déroulements possibles. Dans une implémentation réelle, les branches sont évaluées les unes après les autres, et la première qui mène à une exécution complète est utilisée.

2. Soit *down* une action permettant de descendre un ascenseur d'un étage, nous définissons la procédure $d(n)$ qui permet de descendre de n étages.

$$\textbf{proc}\ d(n)\ (n=0)?\mid (d(n-1);down)\ \textbf{endProc}$$

3. Ranger un ascenseur au rez-de-chaussée.

$$\textbf{proc}\ park\,\pi\, m[atFloor(m)?; d(m)]\ \textbf{endProc}$$

4. Définition de $above(x,y)$ comme une action de test qui serait la clôture transitive de *on* (bloque le programme si x n'est pas au-dessus de y).

$$\textbf{proc}\ above(x,y)\ (x=y)?\mid (\pi z\ on(x,z)?; above(z,y))\ \textbf{endProc}$$

5. La procédure *clean* range tous les blocs dans une boite.

$$\begin{array}{l}\textbf{proc}\ clean\\ \quad (\forall x\ block(x)\implies in(x,Box))?\mid\\ \quad \pi x\ (\forall y \neg on(x,y))?; put(x,Box); clean\\ \textbf{endProc}\end{array}$$

La sémantique d'exécution de Golog est la suivante. L'exécution d'un programme δ basé sur une théorie $\mathscr{D}$ consiste à trouver une séquence d'actions $\vec{a}$ telle que :

$$\mathscr{D} \models Do(\delta, s_0, do(\vec{a}, s_0))$$

où $Do(\delta, s, s')$ signifie qu'il existe une exécution de δ à partir de s qui se termine légalement en s'. La sémantique de Golog est définie sous la forme d'une définition de $Do(\delta, s, s')$ par induction structurelle pour chaque construction de Golog.

L'utilisation de Golog offre une alternative plus rapide à la planification en limitant les choix aux opérateurs non-déterministes, réduisant ainsi l'espace de recherche. Il permet d'exprimer un squelette de plan tout en laissant l'exécution se charger de trouver un plan précis. Pour cette raison, Golog a été utilisé pour diverses tâches, notamment la programmation de robots [GLL00], d'agents pour le Web sémantique [MS01], ou d'hypermédias adaptatifs [JBP+06].

ConGolog et le changement de sémantique

ConGolog [GLL00] est une extension de Golog qui permet de gérer la concurrence. Il introduit plusieurs opérateurs permettant d'exprimer que différentes parties d'un programme s'exécutent

simultanément et propose une nouvelle sémantique pour ces opérateurs et pour les opérateurs de Golog. Cette nouvelle sémantique est compatible avec celle de Golog : ConGolog redéfinit le prédicat *Do* et pour tout programme δ contenant uniquement des constructions Golog (et pas de constructions introduites par ConGolog) :

$$Do_{ConGolog}(\delta, s, s') \equiv Do_{Golog}(\delta, s, s') \tag{2.1}$$

ConGolog introduit deux types de constructions au-dessus de Golog : des constructions parallèles et des constructions synchronisées. Les constructions parallèles permettent l'exécution de plusieurs programmes en même temps, chacun des programmes en cours d'exécution pouvant effectuer l'action suivante. Dans un programme concurrent, le test ϕ? permet de bloquer un programme si ϕ est faux ; l'exécution du programme reprend une fois qu'un autre programme a modifié la situation de manière à ce que ϕ soit vrai. Les constructions synchronisées permettent d'exprimer les «if» et «while» classique, avec la garantie que le sous-programme commence dans la situation dans laquelle la condition a été évaluée (dans un programme concurrent, aucun autre sous-programme ne peut agir entre l'évaluation de la condition du «if» et le début de l'exécution du corps du «if»).

— **if** ϕ **then** δ_1 **else** δ_2, expression conditionnelle synchronisée ;
— **while** ϕ *do* δ, boucle while synchronisée ;
— $(\delta_1 \,||\, \delta_2)$, exécution concurrente de δ_1 et δ_2 : l'un des deux peut être bloqué s'il attend une action primitive dont les préconditions ne sont pas remplies, ou un test dont la condition est fausse, dans ce cas l'exécution du programme continue avec l'autre processus jusqu'à ce que l'exécution du processus bloqué puisse reprendre (si δ_1 et δ_2 sont bloqués, ou que l'un est bloqué et l'autre terminé, alors $\delta_1 \,||\, \delta_2$ est bloqué) ;
— $(\delta_1 \rangle\rangle \delta_2)$, exécution concurrente avec priorité différente : similaire à la construction précédente, mais δ_2 ne s'exécute que si δ_1 est bloqué ou terminé ; si δ_1 et δ_2 sont tous deux exécutables, il est garanti que c'est δ_1 qui sera exécuté.
— $\delta^{||}$, itération concurrente : comme δ^*, δ est exécuté un nombre non-déterministe de fois, mais les instances de δ sont exécutées en concurrence plutôt qu'en séquence.

Au-delà de ces nouvelles constructions, ConGolog introduit une nouvelle sémantique pour Golog. La sémantique classique de Golog, dite d'évaluation, est basée sur la définition pour chaque construction de l'évaluation complète du programme : $Do(\delta, s, s')$ indique que le programme δ exécuté dans la situation s se terminera dans la situation s'. Cette sémantique ne permet pas de modéliser la concurrence, car elle considère que l'exécution d'un programme δ est toujours atomique (et qu'il ne peut donc pas être exécuté parallèlement à une autre programme).

Pour cela, ConGolog propose une sémantique de transition : on s'intéresse à l'exécution d'une étape élémentaire (un élément atomique de l'exécution, qui ne peut être interrompu lors de la concurrence), c'est-à-dire une action primitive. La sémantique est définie par deux prédicats. Le prédicat *Trans* exprime comment exécuter une étape d'un programme : $Trans(\delta, s, \delta', s')$ exprime que l'exécution d'une étape de δ dans la situation s mène à la situation s', avec δ' restant à être exécuté. Évidemment, *Trans* ne suffit pas, car il ne permet pas d'exprimer qu'un programme est terminé. Cela est fait par le prédicat *Final* : $Final(\delta, s)$ exprime que le programme δ peut être considéré comme terminé dans la situation s (plus rien à exécuter).

Par exemple, la sémantique de l'opérateur de séquence ; est la suivante :

$$\begin{aligned} Trans(\delta_1;\delta_2, s, \delta', s') \equiv \exists\gamma.\delta' = (\gamma;\delta_2) \wedge Trans(\delta_1, s, \gamma, s') \\ \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s') \end{aligned} \tag{2.2}$$

Exécuter une étape élémentaire de $\delta_1;\delta_2$ consiste à exécuter une étape élémentaire de δ_1, (c'est la partie $Trans(\delta_1, s, \gamma, s')$), puis continuer avec le reste de δ_1 (γ) en séquence avec δ_2, ou alors, si δ_1 est terminé ($Final(\delta_1, s)$), à exécuter une étape de δ_2 et continuer avec le reste de δ_2 ($Trans(\delta_2, s, \delta', s')$).

$$Final(\delta_1;\delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \tag{2.3}$$

Le programme $\delta_1;\delta_2$ est terminé si et seulement si δ_1 et δ_2 sont tous deux terminés.

La sémantique complète est donnée en annexe C.

Comme pour Golog, le problème de l'exécution d'un programme δ dans une situation s consiste à trouver une situation s' [13] telle que $Do(\delta, s_0, s')$. ConGolog donne une nouvelle définition de Do basée sur cette sémantique, présentée dans l'équation (2.4).

$$Do(\delta, s, s') = \exists\delta'.Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s') \tag{2.4}$$

Le prédicat $Trans^*$ est la fermeture transitive de $Trans$: $Trans^*(\delta, s, \delta', s)$ signifie qu'en exécutant le programme δ dans s, on peut arriver en un certain nombre d'étapes à la situation s', avec δ' restant à exécuter. L'équation (2.4) exprime que trouver une exécution d'un programme consiste simplement à trouver une séquence d'actions élémentaires menant à une situation finale.

Cette nouvelle sémantique est importante, car elle permet l'introduction d'actions exogènes. Les actions exogènes sont des actions qui ne sont pas présentes dans le programme, mais qui peuvent apparaître spontanément lors de l'exécution. Elles offrent une manière de représenter les changements du monde qui échappent à la volonté de l'agent, comme une porte qui claque à cause d'un courant d'air. Les actions exogènes peuvent être une manière de représenter les actions de l'utilisateur : ce sont des actions qui ne font pas partie du programme, mais qui peuvent tout de même changer la situation. Cependant, l'utilisation d'actions exogènes ne permet pas de définir finement à quel moment ces actions peuvent se produire. Or l'interaction avec l'utilisateur se produit dans un cadre précis de dialogue : l'utilisateur ne répond que quand il est interrogé. Cette sémantique fournit aussi des bases pour l'exécution en ligne.

Un autre apport intéressant de ConGolog est l'encodage des programmes. Dans Golog, les programmes ne sont pas des termes du premier ordre, mais des macros. La sémantique de ConGolog contient des quantificateurs portant sur des programmes, ce qui exige que les programmes soient présents dans la logique en tant que termes du premier ordre. Un encodage des programmes Golog en termes du premier ordre a donc été mis en place [14]. Cet encodage permet non seulement de quantifier sur les programmes, mais de manipuler les programmes comme des termes du premier ordre, et donc d'écrire des formules portant sur les programmes. Cela nous sera utile pour la transformation de programmes.

13. Trouver une situation et trouver une séquence d'actions menant à une situation sont équivalents, car les situations sont définies par les séquences d'actions qui y mènent : $s' = do(\vec{a}, s)$

14. De manière générale, les programmes ne sont pas des termes du premier ordre : les tests contiennent des formules et l'opérateur π est un quantificateur. Cependant, il est possible de contraindre l'expression des programmes pour les ramener à des termes du premier ordre. Ces contraintes (telles que l'interdiction de mentionner des programmes dans les formules contenues dans les tests) ne sont pas gênantes, car les limites ne sont de toute façon pas rencontrées lors d'une utilisation normale de Golog.

2.2.3 Gestion de l'interlocuteur humain

Golog permet d'exprimer un agent interagissant avec un environnement certain : l'agent sait tout de l'environnement, et sait quelles sont les conséquences de ses actions. L'exécution est une série de choix de la part de l'agent pour atteindre une solution. Dans notre cas, l'agent interagit avec un utilisateur qui est incertain : on ne sait jamais comment l'utilisateur va répondre aux questions que l'on pose. L'agent n'est plus seul à faire des choix.

Exemple

Dans le cadre d'un hypermédia adaptatif pédagogique, l'agent propose des ressources à l'utilisateur. Dans le premier cas, l'agent choisit les ressources parmi les définitions et les exemples comme il le souhaite (figure 2.2). Dans le second cas, l'agent propose des exercices et doit prendre en compte le résultat de l'exercice pour déterminer la ressource suivante (figure 2.3).

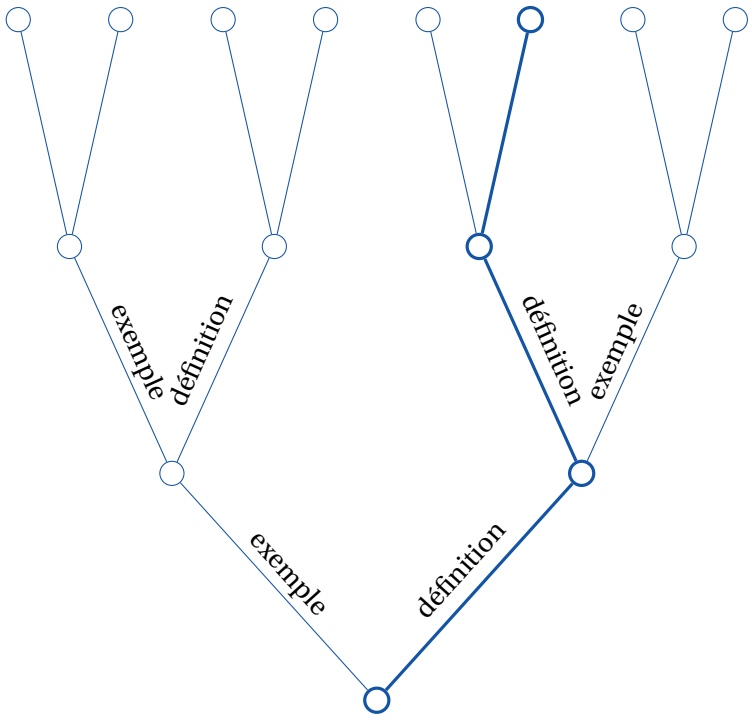

FIGURE 2.2 – Les choix d'un agent seul.

L'agent fait chaque choix, et peut décider exactement le parcours qu'il souhaite.

Une manière de contourner ce problème est de faire apparaître les choix de l'utilisateur comme des choix non-déterministes dans le programme Golog, mais cette solution brouille la frontière entre l'agent et l'utilisateur en mélangeant les choix qu'ils font. Or, il est important de distinguer les choix de l'agent, sur lesquels l'agent peut réfléchir et se projeter en avant pour obtenir le meilleur résultat, et les choix du l'utilisateur, sur lesquels l'agent n'a aucun contrôle.

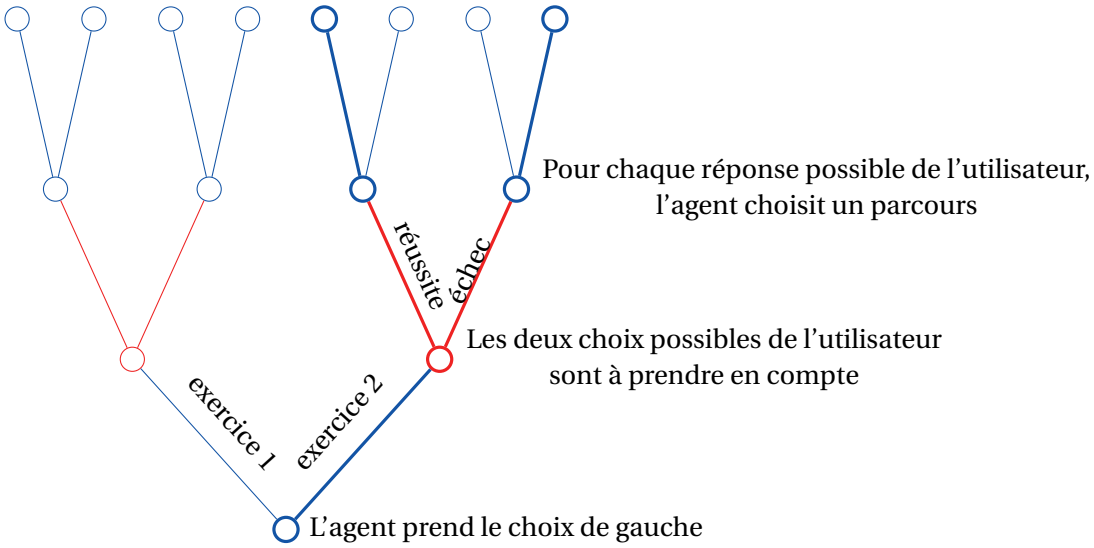

FIGURE 2.3 – Les choix d'un agent interagissant avec un utilisateur. Les choix de l'agent sont en bleu, ceux de l'utilisateur en rouge.

L'agent prend certaines des décisions, mais ne peut prévoir celles de l'utilisateur, et doit donc prendre en compte les multiples réponses possibles.

Interaction avec un environnement incertain

Il existe de nombreuses manières de modéliser des changements de l'environnement qui ne sont pas directement dus à l'agent, ou qui lui échappent d'une manière ou d'une autre.

L'utilisation d'actions exogènes [GLL00], introduites dans Golog avec ConGolog, consiste à définir des actions qui ne sont pas présentes dans le programme de l'agent, mais qui peuvent être exécutées à n'importe quel moment pendant le déroulement du programme. Formellement, un programme ConGolog δ est remplacé par $\delta \,||\, \delta_{EXO}$, où $\delta_{EXO} = (\pi a\, Exo(a)?; a)^*$ et $EXO(a)$ indique que a est une action exogène. Cette approche présente deux problèmes. D'une part, le lancement des actions exogènes est mélangé au programme, et rien ne distingue les actions exogènes des autres. D'autre part, l'interaction avec l'utilisateur sur le Web se fait de manière très contrainte : une page est envoyée à l'utilisateur, qui envoie une réponse ; rien ne peut se passer en dehors du moment précis où l'application attend la réponse de l'utilisateur.

[BF+95] propose une variante du calcul des situations comprenant des actions non-déterministes, qui ont plusieurs issues possibles, ainsi qu'une solution au problème du cadre prenant en compte ces actions non-déterministes. De telles actions pourraient être utiles pour représenter les interactions avec l'utilisateur (on demande à l'utilisateur, à un moment donné, de faire un choix), mais il n'existe pas de langage similaire à Golog basé sur ce formalisme ni de système de planification. Ce formalisme est donc moins adapté à l'écriture d'un agent que Golog.

Pour gérer les actions de sensing, dont le résultat n'est pas connu à l'avance et qui empêchent donc la planification classique, sGolog [Lak99] cherche une exécution d'un programme qui n'est pas une séquence d'actions, mais un arbre d'actions : chaque action de sensing cause un branchement de l'arbre, et une planification est lancée pour chaque résultat possible du sensing. Il est ainsi possible de générer hors-ligne un plan conditionnel qui pourra être exécuté quel que soit le résultat des actions de sensing.

Il existe également des approches s'intéressant à la quantification de l'incertitude. [BHL98] introduit la notion d'effecteurs bruités, qui sont des actions ayant un argument numérique dont l'argument effectif peut être différent de l'argument demandé (par exemple, un robot décide d'avancer de 10cm, mais l'imprécision de ses moteurs le fait avancer en fait de 10.2cm, et l'imprécision de ces capteurs le fait mesurer une avancée de 9.9cm). L'article s'intéresse à la limitation de l'imprécision dans ce contexte. [BP03] traite le résultat des actions d'un point de vue probabiliste, en donnant à chaque action des résultats probables, et en définissant un mode de planification qui permet d'obtenir un plan qui marche avec une certaine probabilité, compte tenu d'actions qui peuvent échouer ou avoir différents résultats. De la même manière, [GL00] introduit pGolog, une variante de Golog dont les actions primitives peuvent être probabilistes ; les actions de mesure de valeurs incertaines sont alors considérées comme des actions probabilistes. [GL01] s'intéresse à la mise à jour des croyances dans pGolog, c'est-à-dire à comment gérer les situations où les résultats de nouvelles mesures sont en contradiction avec ce que l'agent croit.

DTGolog [BRS+00] est une variante de Golog intégrant des processus de décision de Markov dans Golog, permettant de chercher des plans optimaux dans le cadre d'actions non déterministes. Un interpréteur en ligne de DTGolog a été proposé [Sou01], mais celui-ci ne peut calculer une exécution optimale que jusqu'à l'action de sensing suivante. Ce problème peut être résolu par le remplacement des actions de sensing explicites par des capteurs passifs actualisant continuellement le modèle de l'environnement de l'agent [FFL04]. Cette solution n'est cependant disponible que quand de tels capteurs existent, comme c'est le cas pour des robots évoluant dans le monde réel.

Solution épistémique

Une solution au problème de l'interaction avec un environnement incertain consiste à traiter explicitement la connaissance de l'agent en maintenant une représentation de ce que l'agent sait et ne sait pas de l'environnement, et en introduisant des actions dites de *sensing* pouvant mettre à jour la connaissance de l'agent. Pour cela, [SL93, SL03], à la suite de [Moo79], adaptent le modèle classique de connaissance à base de mondes possibles au calcul des situations. Dans ce modèle, une relation binaire d'accessibilité entre les situations est introduite, où une situation s' est accessible depuis s si, d'après ce que l'agent sait, il n'est pas possible de distinguer s et s'. En conséquence, un fait est connu si ce fait est vrai dans toute situation s' accessible à partir de s. La figure 2.4 illustre cette relation entre les situations. La relation $K(s', s)$ (s' est accessible à partir de s) est introduite comme un fluent, et la notation $Knows(P, s)$ (P est connu dans la situation s) est définie à partir de K.

Exemple

La définition du fluent *Knows* pour le fluent $broken(y)$ est :

$$Knows(broken(y), s) \stackrel{def}{=} \forall s'\, K(s', s) \implies broken(y, s')$$

On distingue alors deux types d'actions : les actions ordinaires, et les actions qui produisent de la connaissance. Pour chaque action, un axiome de résultat de sensing de la forme

$$SR(\alpha(\vec{x}), s) = r \equiv \phi_\alpha(\vec{x}, r, s)$$

est défini, liant les résultats d'une opération de sensing à une information sur les fluents dans la situation s. La fonction *SR* (sensing result) est une fonction qui caractérise les résultats de sensing.

Exemple

Pour une action $SENSE_Q$ déterminant la valeur d'un fluent Q, la fonction *SR* pourra renvoyer "*YES*" si Q est vrai "*NO*" si Q est faux :

$$SR(SENSE_Q) = r \equiv (r = \text{"}YES\text{"} \wedge Q(s)) \vee (r = \text{"}NO\text{"} \wedge \neg Q(s))$$

L'introduction du fluent épistémique K et du sensing permet l'écriture d'agents qui ne connaissent pas tout de l'environnement. Cette ignorance initiale de la part de l'agent est utile pour modéliser une interaction avec un utilisateur dont l'agent ne connaît pas a priori les décisions. Cependant, ce formalisme ne permet de représenter que des faits statiques qui sont inconnus de l'agent, mais pas des faits changeants : toute modification de l'environnement se fait toujours par l'intermédiaire des actions de l'agent. Cela signifie que le sensing peut être utilisé pour représenter un agent qui souhaite

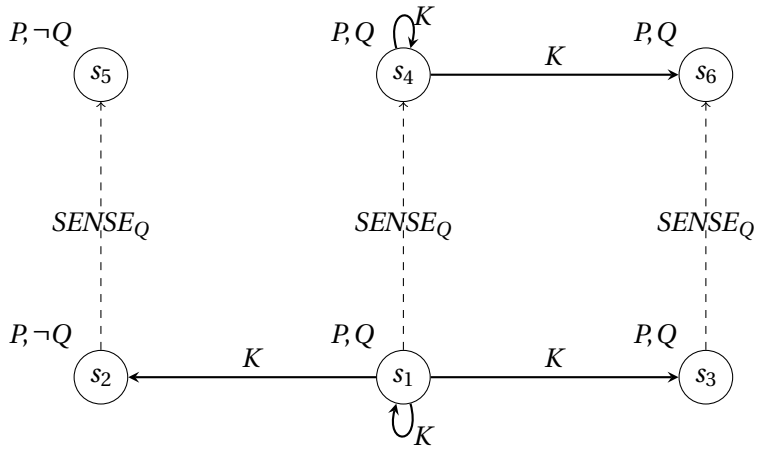

FIGURE 2.4 – Relation d'accessibilité K entre les situations

L'action $SENSE_Q$ ne change pas la valeur de P et Q, mais change le fluent K. La formule $K(s_1, s_2)$ signifie que l'agent ne peut pas distinguer s_1 et s_2, c'est-à-dire qu'il ne connaît pas la valeur de K. Après l'action de sensing $SENSE_Q$, l'agent peut distinguer s_4 et s_5, car il connaît la valeur de Q.

apprendre des caractéristiques fixes de l'utilisateur (son âge), mais pas des caractéristiques changeant pendant l'interaction (un agent pédagogique va mesurer plusieurs fois la connaissance d'un sujet chez l'utilisateur ; si tout se passe bien, le niveau de connaissance devrait changer au fur et à mesure que l'utilisateur apprend).

Le sensing a aussi l'inconvénient de rendre inadaptées les techniques classiques de planification. Pour remédier ce problème, [Lev96] étudie la création de plans conditionnels (plans ayant des branchements en fonction des résultats de sensing) dans le cadre du calcul des situations. [Rei01] formalise une représentation de la connaissance issue du sensing permettant le raisonnement et la preuve, ce qui permet la mise au point d'un interpréteur Golog en ligne (un interpréteur relié au monde réel [15] dans lequel les actions exécutées dans le formalisme sont aussi exécutées dans le monde réel, ce qui est nécessaire pour le sensing, et où le retour en arrière n'est pas possible). [BM06] étend ce travail en proposant une sémantique d'exécution hors-ligne pour les programmes comprenant des actions de sensing, permettant donc de trouver des plans malgré la présence des actions de sensing dont le résultat est inconnu, offrant donc des possibilités de planification à ces programmes. [DGL99b] propose une évolution de l'axiome des états successeurs permettant de spécifier quand la valeur d'un fluent peut être déduite des états précédents, et quand elle doit être obtenue par sensing. [DGL99a] propose l'ajout d'un opérateur de recherche Σ à Golog permettant de spécifier des morceaux de programmes à exécuter hors-ligne par planification, au sein d'un programme exécuté en ligne avec sensing. Ces deux idées réunies ont amené à la création du langage IndiGolog [DGLS01, GLLS09]. [SPLL00] propose une extension de la notion de connaissance en croyance, qui peut être révisée si l'agent acquiert de nouvelles informations qui contredisent ce qu'il croyait précédemment.

Parmi les extensions intéressantes de la théorie du sensing, [MS$^+$00] propose de formaliser la notion de test, c'est-à-dire la création d'un plan incluant des opérations de sensing dans le but de répondre à une question qui n'est pas forcément une conséquence directe de l'opération de sensing, comme le fait de brancher une bouilloire électrique et de mesurer la température de l'eau dans le but de savoir s'il y a du courant dans la prise.

[BGMP01] se penche sur le même problème dans le cadre de logiques modales, et propose une solution similaire (une relation d'accessibilité pour représenter la connaissance).

IndiGolog

Contrairement à ConGolog sur lequel il est basé et qui repose sur des théories des actions basiques, IndiGolog repose sur des théories des actions gardées, où les axiomes de l'état successeur et les axiomes de résultat de sensing sont remplacés par des équivalents gardés, c'est à dire possédant une formule contenant des fluents et des fonctions de sensing appelée garde qui indique quand l'axiome est applicable. La fonction *SR* disparaît pour laisser place à des fonctions de sensing, des fonctions unaires dont le seul argument est la situation, et qui renvoient des valeurs dans un domaine quelconque (par exemple, *thermometer*(s) renvoie la température dans la situation s). L'axiome gardé de l'état successeur est de la forme

$$\alpha(\vec{x}, a, s) \implies [F(\vec{x}, do(a, s)) \equiv \gamma(\vec{x}, a, s)]$$

15. Par exemple, un interpréteur commandant un robot.

et l'axiome gardé de fluent mesuré est de la forme

$$\alpha(\vec{x}, s) \implies [F(\vec{x}, s) \equiv \rho(\vec{x}, s)]$$

où α est une formule contenant des fluents et des fonctions de sensing et appelée la garde, F est un fluent, γ est une formule contenant des fluents, et ρ est une formule contenant des fonctions de sensing.

Avec une théorie de l'action gardée, il est donc possible de spécifier des fluents dont les valeurs sont déduites des situations précédentes dans certains cas, mesurées dans d'autres cas, ou inaccessibles. Par exemple, un robot dans une maison peut déduire l'état courant de la porte de son état précédent (la porte reste ouverte ou fermée si l'agent ne la ferme ou ne l'ouvre pas), mais seulement dans la situation où il est seul ; il peut mesurer l'état de la porte quand elle est dans son champ de vision. Dans la situation où l'agent n'est pas seul et où il ne voit pas la porte, il ne sait pas quel est l'état de la porte, et n'a aucun moyen de le savoir. L'agent peut ne pas savoir un fait qu'il a su par le passé, et des choses peuvent se produire sans que l'agent le sache. Cette ignorance est nécessaire à la modélisation d'interactions avec l'utilisateur. Un agent pédagogique ne peut pas savoir si la leçon qu'il vient de prodiguer a eu un effet sur le niveau de connaissance de l'utilisateur sans l'évaluer. Ce formalisme gère donc autant la mise à jour de connaissances que l'extension de connaissances : il est possible d'invalider des connaissances passées pour les mettre à jour.

IndiGolog est une extension de ConGolog qui utilise une théorie de l'action gardée. IndiGolog introduit un nouvel opérateur : l'opérateur de recherche Σ. Lors d'une exécution en ligne d'un programme, $\Sigma(\delta)$ exécute δ hors-ligne de manière à en trouver une exécution complète qui soit possible, puis déroulera cette exécution en ligne [16]. De cette manière, il est possible d'entremêler une exécution en ligne avec du sensing et de la planification sur les morceaux de programmes n'utilisant pas de sensing.

2.2.4 Logiques modales

DyLOG [BGMP01] est un langage de programmation logique basé sur une théorie de l'action en logiques modales. Le cadre logique sur lequel se base DyLOG représente les états comme des mondes possibles et les actions comme des modalités. Le problème du sensing est résolu par l'utilisation d'un fluent épistémique : les actions dont le résultat est incertain retirent de la connaissance, tandis que les actions de sensing ajoutent de la connaissance.

DyLOG a été utilisé pour la création d'agents interagissant avec des services sur le Web [BBP04]. Un agent assiste des étudiants dans la création d'un parcours pédagogique valide. Les cours sont modélisés par des actions qui font passer l'étudiant d'un état de connaissance à un autre. Il a également été utilisé pour créer des agents d'assistance à l'achat d'ordinateurs en pièces détachées [BBCP01]. Là, ce sont les ajouts de composants au panier qui sont des actions. L'agent utilise le raisonnement de DyLOG pour guider l'utilisateur et utilise du sensing pour demander son avis à l'utilisateur quand un choix se présente. L'agent parcourt ainsi les états en avançant avec l'utilisateur jusqu'à atteindre un état final où l'objet à composer (cursus ou ordinateur) est conforme aux requêtes de l'utilisateur.

16. Sans Σ, dans une exécution en ligne, le programme est exécuté sans prévision, en faisant les choix de manière aléatoire plutôt qu'en cherchant une exécution valide.

2.2.5 Conclusion

Le raisonnement sur des actions a beaucoup été utilisé pour exprimer et implémenter des agents intelligents évoluant dans des environnements changeants et partiellement inconnus. Cette technique a été utilisée pour doter des robots de systèmes de raisonnements capables d'évaluer les effets de leurs actions et de planifier une série d'actions pour atteindre leur but, mais également dans le contexte d'interactions avec des humains, notamment sur le Web.

Cependant, bien souvent, le raisonnement sur les actions n'est qu'un composant de l'application, une modélisation d'une partie du domaine que traite l'application, le reste étant écrit dans un langage de programmation plus traditionnel. Dans ces formalismes, l'agent n'est pas présent dans le cadre d'un modèle clair de l'interaction entre l'agent et un utilisateur, mais sous la forme d'un composant ad hoc d'une application. Il n'existe pas de travaux modélisant l'interaction entre un utilisateur et une application Web sous la forme d'un agent.

En outre, la personnalisation n'est pas abordée : les travaux existants sur les agents interagissant avec des humains ne répondent pas à l'aspect altération de comportement que nous étudions.

2.3 Altération d'agents

Nous nous intéressons à la personnalisation sur le Web dans le cadre d'agents rationnels. Cependant, la personnalisation n'est qu'un cas particulier d'altération du comportement en fonction d'un contexte, il est donc nécessaire d'étudier toutes les possibilités d'altération, qui pourraient également servir pour la personnalisation.

Dans cette section, nous faisons un tour d'horizon des travaux liés à la personnalisation d'agents rationnels, ainsi que d'autres techniques d'altération d'agents, comme la personnification ou l'expression de buts ou de préférences pouvant influencer les choix de l'agent.

2.3.1 Préférences globales sur l'exécution d'agents

On retrouve souvent dans la littérature la notion de préférence : un but ou une contrainte optionnelle dans l'exécution d'un agent, qui sera satisfait si possible, mais abandonné sinon. Une priorité peut être donnée entre plusieurs buts, de manière à indiquer quels buts choisir si certains buts entrent en conflit. Ces préférences permettent de modifier la manière dont l'agent remplit sa mission, sans pour autant remettre en cause la mission. On peut penser la personnalisation en termes de préférences dépendant de l'utilisateur, poussant l'agent à accomplir sa mission de manière adaptée.

Les préférences sont de deux ordres : quantitatives et qualitatives. Une préférence quantitative est une valeur à maximiser ou à minimiser pour l'agent, comme un trajet à rendre le plus court possible. Une préférence qualitative est un but, une condition à rendre vraie ou à maintenir vraie, comme le fait qu'un trajet ne doit pas emprunter d'autoroutes. Les préférences permettent de juger un état préférable à un autre, et donc de privilégier une exécution préférable à une autre.

Exemple

Par exemple, un robot doit livrer le courrier à différentes personnes travaillant dans un laboratoire : il doit mettre au point un plan consistant en une route à suivre pour passer par chaque ►

Exemple (cont)

bureau où il doit livrer du courrier. Son but est donc d'avoir visité chaque bureau à visiter. On ajoute une préférence qualitative : le robot ne doit pas passer deux fois par le même couloir (pour ne pas déranger les personnes travaillant autour). Le robot cherchera à satisfaire cette préférence s'il le peut, mais l'abandonnera si cela n'est pas possible (par exemple, si le bureau est construit en suivant le plan des ponts de Königsberg). On ajoute également une préférence quantitative : le robot doit utiliser le moins d'énergie possible. De toutes les routes que le robot peut envisager, il choisira celle qui lui fera consommer le moins d'énergie.

DTGolog [BRS+00] introduit une variante de Golog permettant de définir des actions stochastiques, c'est à dire dont le résultat n'est pas certain, mais probabiliste, les probabilités étant connues par l'agent. L'exécution d'un programme DTGolog demande la définition d'une fonction de récompense qui attribue une valeur numérique à chaque état, et consiste à rechercher un déroulement maximisant un compromis entre la récompense (la valeur de la fonction de récompense en cas de réussite) et le risque (la probabilité que le déroulement échoue et ne rapporte pas la récompense). L'utilisation d'une fonction de récompense permet de spécifier des préférences quantitatives.

DTGolog a été étendu [FM06] pour prendre en compte des préférences qualitatives. Les contraintes sont exprimées dans une logique temporelle, ce qui permet d'exprimer aussi bien des conditions sur l'état final que des conditions sur les états intermédiaires. Les contraintes sont ensuite compilées dans un programme DTGolog particulier dont les exécutions possibles sont exactement les exécutions satisfaisant les contraintes. Par exemple, dans un domaine contenant deux actions a et b et un fluent ϕ, la contrainte «ϕ ne doit jamais être faux» s'exprime **always**(ϕ) et génère le programme $(a \mid b; \phi?)^*$ dans lequel a et b peuvent être exécutés un nombre quelconque de fois, tant qu'après chaque exécution, ϕ est vrai. L'application de la contrainte à un programme consiste à exécuter ce programme en synchronisation avec le programme généré à partir de la contrainte. L'exécution synchronisée (**sync**), définie dans le même article, permet de garantir que l'exécution du premier programme est aussi une exécution valide du second, et donc que l'exécution du programme est conforme aux contraintes.

[FM06] définit enfin un opérateur de préférence **lex**(δ_1, δ_2) permettant d'exprimer «exécuter le programme δ_1 ; si celui-ci n'a pas d'exécution valide, exécuter δ_2». Cet opérateur permet d'exprimer qu'une contrainte est une préférence, mais n'est pas obligatoire : l'exécution d'un programme δ en cherchant à satisfaire de préférence la contrainte compilée dans le programme $\delta_{contrainte}$ est exprimée par **lex**(**sync**($\delta, \delta_{contrainte}$), δ) (chercher une exécution satisfaisant la contrainte, s'il n'en existe pas, ignorer la contrainte).

En combinant les travaux de [BRS+00] et de [FM06], il est possible d'exprimer des préférences quantitatives (valeur à maximiser) et qualitatives (propriété à maintenir vraie) sur l'exécution de programmes Golog. Ces préférences sont globales et concernent l'intégralité du programme. Cela rend difficile l'application de préférences sur un programme complexe composé de plusieurs parties plus ou moins liées, car il n'est pas possible d'isoler une partie du programme dans l'expression des préférences.

Une approche similaire a été envisagée dans IndiGolog [SS03] : un opérateur dit de recherche rationnelle Δ_{rat} est introduit. L'exécution du programme $\Delta_{rat}(\delta : \phi_1 > \ldots > \phi_n)$ cherche à exécuter le programme δ de manière à ce que la séquence de buts $\phi_1, \ldots, \phi_i$ $(1 \leq i \leq n)$ la plus grande possible

soit satisfaite dans la situation finale de δ. Ceci permet d'ajouter des objectifs optionnels qu'un agent cherchera à satisfaire tout en pouvant continuer son exécution si ce n'est pas possible. Cette construction n'exprime qu'un sous-ensemble de ce qui est exprimable dans les travaux de [FM06], mais, en tant que construction Golog, peut être utilisée à l'intérieur d'un programme pour exprimer la préférence sur une partie de l'exécution et pas sur le programme entier.

On retrouve aussi la notion de priorité entre des buts dans [KL10], qui introduit formellement la notion de but dans le cadre du calcul des situations et permet de gérer dynamiquement les buts et les priorités entre les buts.

La notion de buts avec priorité est utile pour l'expression d'agents personnalisés ou personnifiés, car elle permet d'introduire des variations dans le comportement d'un agent à partir d'une même base et d'un même fonctionnement. Si un programme accomplit un seul service, en permettant de faire varier les buts, on peut changer la manière dont le programme accomplit ce service, et ainsi offrir des possibilités de personnalisation. Pour deux utilisateurs avec des buts différents, le programme fournira donc le même service de manière adaptée à chaque utilisateur, personnalisant ainsi le service en question.

Il peut être utile d'exprimer des objectifs ou des contraintes qui ne sont pas absolument nécessaires pour la réussite un service, mais qui sont tout de même préférables. Ces préférences permettent de prendre en compte des contraintes supplémentaires sans remettre en cause la réussite : si elles ne sont pas atteignables, elles seront simplement ignorées. Ces préférences sont pertinentes dans le contexte de la personnalisation, car l'adaptation du comportement à l'utilisateur passe toujours après la garantie de fonctionnement du système. L'expression de préférences peut être elle-même utile dans l'application : un utilisateur cherchant un hôtel pourra spécifier qu'il préférerait un hôtel ayant du WiFi, mais que le prix ne doit absolument pas dépasser 60 euros la nuitée.

La gestion des priorités dans les buts de DTGolog permet d'exprimer des préférences en donnant à l'agent des buts secondaires dont l'exécution est moins importante que l'accomplissement de la mission principale, et qui peuvent donc être abandonnés s'ils sont incompatibles avec la mission.

2.3.2 Choix des buts dans le modèle BDI

BDI (Belief-Desire-Intention) [RG+95] est un modèle d'architecture d'agents intelligents basé sur une distinction claire de trois éléments de base :

- les croyances représentent ce que l'agent sait (ou pense savoir) de son environnement ;
- les désirs représentent ce que l'agent souhaite accomplir ;
- les intentions représentent ce que l'agent a décidé de faire en fonction de ses désirs et de ses croyances.

Le langage CAN [WPHT02], basé sur le formalisme BDI, introduit une représentation explicite des buts et un système de notation basé sur cette représentation. Les buts sont représentés sous forme déclarative, et les plans comme des procédures ayant des préconditions et éventuellement des sous-plans. L'idée est d'exécuter un plan dans un but (une formule logique décrivant l'état à atteindre) : le plan sera exécuté jusqu'à ce que le but soit atteint, et si le plan se termine sans que le but soit atteint, ou que la condition d'échec (déclarative, comme le but) devienne vraie, le plan sera considéré comme ayant échoué et un autre plan sera choisi pour atteindre le but.

CANPlan [SP07, SP11] étend CAN en rajoutant un système de planification par prévision hors-ligne [17], c'est à dire un système évaluant les exécutions des plans avant de s'engager dans leur exécution, ce qui permet d'éviter de gaspiller des ressources dans l'exécution de plans vains, ou même de se bloquer dans un état où le but devient impossible à atteindre. Ce langage permet donc l'expression et l'implémentation d'agents cherchant à atteindre des buts déclaratifs à l'aide de plans définis lors de la conception de l'agent dans une librairie de plans. L'agent choisit des plans différents en fonction de son environnement pendant l'exécution, et en fonction du but à atteindre.

Cette manière d'exprimer les agents offre la possibilité de commander un agent en lui donnant simplement un but déclaratif, pour peu qu'on lui ait donné les plans nécessaires pour atteindre le but, permettant effectivement un contrôle de très haut niveau de l'agent. Cependant, elle n'offre pas un contrôle fin sur le comportement bas niveau de l'agent ni sur les choix qu'il fait pour atteindre le but. Si plusieurs plans sont disponibles pour atteindre le but, il n'est pas possible d'indiquer qu'un des plans est préférable à l'autre. Il est possible d'ordonner à un agent «emmène-moi à l'aéroport», mais pas «évite les autoroutes, si possible».

En comparaison avec DTGolog, CANPlan est centré sur les buts et cherche parmi sa bibliothèque de plans comment les atteindre, tandis que DTGolog est centré sur son programme et n'utilise les buts que comme limitations des exécutions possibles du programme. Les buts de DTGolog sont également plus larges, car ils permettent d'exprimer des contraintes sur le déroulement de l'exécution, et pas uniquement sur l'état à atteindre.

Un pont a été construit entre BDI et Golog, sous la forme d'une implémentation de CANPlan réalisée en IndiGolog [SL10], ce qui offre un cadre pour l'expression de systèmes agents en IndiGolog, et offre à CANPlan une sémantique ancrée dans la logique classique ouvrant la voie à des capacités de raisonnement habituellement absentes de BDI.

2.3.3 Personnification d'agents

Les techniques classiques de personnification des agents sont très différentes des techniques de personnalisation. Cela est principalement dû au fait que, dans les domaines qui s'intéressent à la personnification, les agents sont basés sur des architectures précises et la personnification est intrinsèquement liée à ces architectures. La famille d'architectures BDI [RG+95] (présentée plus haut) est couramment utilisée, de même que l'architecture cognitive SOAR [LNR87].

Les travaux sur l'implémentation de profils psychologiques [JRP08] se sont récemment concentrés sur des théories psychologiques incluant des taxonomies de traits, telles que le Five Factor Model (FFM) [Gol90] et ses itérations enrichies avec des facettes, ou la taxonomie NEO PI-R [CM92].

[GM04] ont implémenté un modèle psychologique, principalement dédié aux émotions, basé sur l'architecture SOAR, mais la plupart des auteurs ont plutôt proposé des améliorations des architectures BDI incluant à la fois des modules de raisonnement rationnel et psychologique [LDAP08]. Par exemple, le modèle eBDI [JVH07] implémente des émotions dans un framework BDI. Basé sur la plate-forme BDI JACK [HRHL01], CoJACK [ERBP08] fournit des couches supplémentaires qui cherchent à simuler des contraintes physiologiques humaines telles que le temps pris par la réflexion, les limitations de la mémoire à court terme (oubli d'une croyance ou de l'étape suivante d'une procédure), souvenirs flous, attention limitée, ou utilisation de modérateurs pour altérer la cognition. Les architectures BDI offrent

17. *off-line lookahead*

en effet un moteur ouvert et flexible (le cycle de délibération), ce qui a conduit la plupart des études à les utiliser pour supporter les caractéristiques cognitives ou psychologiques, en utilisant par exemple des implémentations du paradigme BDI telles que 2APL [Das08] ou CANPlan [SP07]. Des travaux ont par exemple étudié comment l'architecture cognitive elle-même est influencée par différents traits de personnalité [BS11].

La psychologie de l'agent est habituellement basée sur des états mentaux dynamiques (humeurs) qui influencent l'expression corporelle (faciale et gestuelle) des émotions, mais qui n'ont que peu ou pas d'influence sur les processus de prise de décision de l'agent, notamment dans le contrôle des stratégies de conversation. [MCRK07] cherche à modifier l'expression des émotions (se basant sur la théorie OCC des émotions [OCC88], un modèle et une théorie des émotions issue de la recherche en psychologie) d'un agent en fonction de traits de personnalité. Ils se concentrent principalement sur la manière dont les agents évaluent le résultat de leurs actions et des événements extérieurs et la manière dont celui-ci influence leurs émotions et leurs expressions, sans que cela modifie la prise de décision de l'agent. De la même manière, [RVMC97] a montré que les buts et les plans peuvent être efficacement utilisés pour représenter la personnalité d'un personnage. Les traits de personnalité sont utilisés pour choisir entre les multiples buts d'un agent BDI (les traits influencent les désirs). Une fois choisis, les buts sont planifiés et exécutés directement, les traits psychologiques n'ont pas d'autre influence sur l'agent que le choix des buts. Il en va de même pour [MSd+09], basé sur l'architecture d'agent conversationnel GRETA [Pel00], qui utilise des modèles de personnalité pour l'expression des émotions.

La plupart de ces études se concentrent sur une capacité particulière ou un trait particulier et ne cherchent pas à couvrir l'ensemble des possibilités.

2.3.4 Conclusion

La plupart des techniques d'altération des comportements d'agents permettent d'altérer globalement le comportement, en fournissant un but ou en indiquant que des conditions sont préférables à d'autres. Cela permet à un même agent d'accomplir différentes variantes de la même mission, et d'offrir différentes solutions à un même problème en fonction de ce qui est considéré comme optimal.

Il arrive cependant que l'on souhaite emprunter un autre chemin pour arriver au même endroit : si deux solutions sont égales sous tout rapport, on peut vouloir s'intéresser aux étapes qui y ont mené. Dans le cas de la personnification, la prise en compte d'un profil psychologique peut nécessiter d'agir différemment à chaque étape de l'exécution, sans pour autant arriver à une solution différente. Un agent conversationnel poli et attentionné fournira le même service qu'une autre, mais choisira de s'adresser gentiment à l'utilisateur à chaque fois qu'il en aura l'occasion. De la même manière, pour la personnalisation, on veut autant arriver à la solution optimale pour l'utilisateur que lui plaire à chaque étape. Dans un hypermédia adaptatif, en plus de prendre en compte la connaissance que l'utilisateur souhaite acquérir, on prendra en compte ses habitudes d'apprentissage pour choisir chaque ressource à lui présenter.

Les techniques d'altération globales ne permettent pas de changer les choix bas niveau de l'agent autrement qu'en considérant leurs effets sur le résultat global de l'agent. Il serait pourtant intéressant d'avoir un contrôle fin sur les choix auxquels l'agent est confronté, notamment les choix qui n'ont aucune influence sur le fonctionnement global de l'agent, mais qui ont une importance en tant que

chemin parcouru.

2.4 Conclusion de l'état de l'art

Notre objectif est l'expression de la personnalisation dans des applications web dans le cadre d'agents rationnels. Nous avons montré que l'expression d'applications web en tant qu'agents rationnels nécessite l'utilisation de sensing.

Le sensing est un frein au raisonnement pour les agents, car il introduit un environnement incertain. Pour pouvoir personnaliser un agent, il est nécessaire de pouvoir raisonner sur son comportement au-delà du sensing : il faut prendre en compte ce qui se passera après la prochaine interaction avec l'utilisateur, et ne pas se contenter de ce qui se passe avant. Aucun des formalismes agents disposant des capacités d'altération de comportement qui nous intéressent ne conserve ses propriétés en présence de sensing. Il est donc nécessaire de trouver une autre approche pour garder un contrôle fin sur le comportement de l'agent en présence de sensing.

Deuxième partie

Contributions

Introduction des contributions

Les techniques d'altération du comportement d'un agent permettent d'offrir un comportement adapté à un certain contexte, que ce soient des contraintes de fonctionnement, les contraintes d'un utilisateur (personnalisation), ou des contraintes visant à donner une psychologie apparente à l'agent (personnification). Elles permettent d'utiliser un agent pour différentes variantes d'une tâche tout en ne l'exprimant qu'une seule fois.

La majorité des techniques existantes d'altération du comportement reposent sur la modification de l'exécution de l'agent, notamment en modifiant l'interpréteur pour donner une sémantique différente à un même programme agent. Ces techniques ont cependant l'inconvénient de n'offrir au concepteur d'agent aucune possibilité d'étudier ou de vérifier l'adaptation.

Nous avons choisi d'orienter nos travaux dans une autre direction : la modification des programmes eux-mêmes. Nous transformons un programme agent pour le doter d'un comportement personnalisé, puis le programme transformé est exécuté par l'interpréteur classique. Cette manière de procéder permet au concepteur d'application de voir l'objet intermédiaire, le programme transformé et de vérifier que la transformation est bien conforme à ce qui est souhaité. Cette technique rend envisageable un processus de transformation guidée, permettant à un concepteur d'application de superviser une transformation semi-automatique à travers des outils d'aide à l'adaptation.

Cette partie est organisée de la façon suivante. Au chapitre 3, nous introduisons deux contributions préliminaires : nous commençons par définir les notions de personnalisation forte et faible pour identifier précisément le domaine dans lequel nous travaillons, puis nous présentons WAIG (web applications in Golog), un cadre adapté à la modélisation d'applications Web sous forme d'agents Golog. Les chapitres suivants sont consacrés à l'étude de la transformation de programmes. Au chapitre 4, nous étudions les altérations possibles et la manière dont elles s'expriment dans un programme agent. Nous en tirons un ensemble de transformations qui offrent des garanties pour le programme transformé par rapport au programme original. Nous étudions également la possibilité d'augmenter les programmes, pour les cas où l'altération de l'existant n'est pas suffisante pour l'adaptation. Les transformations définies dans ce chapitre constituent la première partie du cadre PAGE framework (Parametric AGEnts). Au chapitre 5, nous définissons une méthode permettant de comparer les différentes variantes d'un même programme de manière à choisir les transformations les plus judicieuses à appliquer. Cette méthode de comparaison constitue la seconde partie de PAGE framework. Enfin, au chapitre 6, nous définissons PAGE process, un processus semi-automatique de transformation de programmes agents.

Les contributions de cette partie sont illustrées dans la partie suivante par les scénarios d'interactions personnalisés et personnifiés.

Chapitre 3

Contributions préliminaires

Dans ce chapitre, nous introduisons deux contributions servant de bases aux reste de nos travaux : une définition formelle de la personnalisation, et un formalisme agent pour l'expression d'applications web.

3.1 Personnalisation faible et forte

La personnalisation est présente dans plusieurs domaines et communautés, et le terme désigne des choses différentes pour des communautés différentes. Le dénominateur commun est de faire en sorte qu'un système se comporte différemment en fonction de l'utilisateur avec lequel il interagit. Cependant, cette caractérisation est trop large : il suffit qu'un système prenne en compte une donnée de l'utilisateur lors de son fonctionnement ou de la présentation de ses résultats pour pouvoir être qualifié de système personnalisé. En poussant l'idée à l'extrême, une page Web qui montre à l'utilisateur son adresse IP est une application Web personnalisée.

Il est évident que ce dernier exemple n'est pas intéressant dans le cadre de notre étude. Il est donc nécessaire de définir précisément la classe des applications que nous souhaitons étudier, en distinguant les applications utilisant simplement l'historique des interactions avec l'utilisateur et celles qui raffinent un modèle de l'utilisateur et l'utilisent pour adapter le service qu'elles fournissent.

3.1.1 Définitions

Par service, nous entendons l'assistance que le système fournit à l'utilisateur, que l'utilisateur est venu chercher auprès du système et qui est source d'utilité pour l'utilisateur. Le service rendu par un moteur de recherche est de trouver des documents, celui rendu par le Webmail est de visionner et d'envoyer des mails. Nous dirons que le service est adapté à l'utilisateur quand il est fait différemment pour un utilisateur donné de manière à augmenter l'utilité fournie.

Définition 1 (Système non personnalisé)**.** Un système non personnalisé est un système dont le comportement ne varie pas en fonction de l'utilisateur avec lequel il interagit.

Définition 2 (Personnalisation faible)**.** Un système faiblement personnalisé est un système qui fait usage d'informations sur l'utilisateur ou de l'historique de ses interactions de manière ad hoc, sans les organiser ni en extraire un profil.

Par exemple, un Webmail simple est une application faiblement personnalisée, car il montre à chaque utilisateur des mails différents (ses propres mails), mais ne fait rien d'autre qu'aller chercher les mails correspondant à l'identifiant de l'utilisateur.

Les systèmes faiblement personnalisés exploitent les informations de l'utilisateur de manière ad hoc, il est donc impossible d'en tirer des généralités. En conséquence, la personnalisation faible n'est pas celle qui nous intéresse dans cette étude.

Définition 3 (Personnalisation forte)**.** Un système fortement personnalisé est un système qui utilise explicitement les informations qu'il a sur l'utilisateur, raffinées en un modèle, pour adapter le service qui est fourni.

3.1.2 Illustrations

Pour illustrer les différents niveaux de personnalisation, nous détaillons trois scénarios de la vie réelle mettant en jeu des services non personnalisés, faiblement personnalisés et fortement personnalisés. Dans chaque exemple, deux personnes différentes arrivent avec les deux mêmes requêtes.

Absence de personnalisation

Dans un centre commercial, un guichet renseigne les clients. La réponse n'est pas du tout personnalisée, et différents clients posant la même question obtiennent la même réponse.

Première interaction :

Premier client	Second client
— Bonjour, pourriez-vous m'indiquer où est la boulangerie la plus proche ?	— Bonjour, pourriez-vous m'indiquer où est la boulangerie la plus proche ?
— À gauche au fond de ce couloir.	— À gauche au fond de ce couloir.

Deuxième interaction :

— Bonjour, pourriez-vous m'indiquer où est la pharmacie la plus proche ?	— Bonjour, pourriez-vous m'indiquer où est la pharmacie la plus proche ?
— Au second étage.	— Au second étage.

Personnalisation faible

Dans un aéroport, un guichet permet aux voyageurs de retirer leurs cartes d'embarquement. Chaque voyageur reçoit une carte d'embarquement différente, et les personnes au guichet tiennent compte de leurs noms, mais non pas de profil de l'utilisateur : ils ne savent rien d'eux, à part le numéro de leur billet.

Première interaction :

— Bonjour, je souhaite retirer ma carte d'embarquement.
— Quel est votre nom ?
— Je suis Mme Beatrice.
— Voilà votre carte d'embarquement numéro 1684. Bon voyage, Mme Beatrice.

— Bonjour, je souhaite retirer ma carte d'embarquement.
— Quel est votre nom ?
— Je suis M. Benedict.
— Voilà votre carte d'embarquement numéro 9751. Bon voyage, M. Benedict.

Deuxième interaction, le traitement sera différent selon que la réservation existe ou non :

— Bonjour, je souhaite retirer ma carte d'embarquement.
— Quel est votre nom ?
— Je suis Mme Beatrice.
— Voilà votre carte d'embarquement. Bon voyage pour Messina, Mme Beatrice.

— Bonjour, je souhaite retirer ma carte d'embarquement.
— Quel est votre nom ?
— Je suis M. Benedict.
— Je suis désolé, je n'ai aucun billet à ce nom.

Personnalisation forte

Dans une boulangerie, les employés reconnaissent les clients et leur proposent leurs commandes habituelles. Les employés disposent d'un profil des clients (leurs commandes habituelles), et sont capables d'adapter l'échange en fonction de ce profil.

Première interaction :

— Bonjour.
— Bonjour M. Claudio. Une baguette et un pain au chocolat comme d'habitude ?

— Bonjour.
— Bonjour Mme Hero. Un pain de campagne, comme d'habitude ?

Deuxième interaction :

— Bonjour, je voudrais un pain complet.
— Bonjour M. Claudio. Est-ce que vous prendrez un pain au chocolat avec ceci ?

— Bonjour, je voudrais un pain complet.
— Bonjour Mme Hero. Avec ceci ?

3.1.3 Conclusion

La distinction entre la personnalisation forte et la personnalisation faible est nécessaire, car la diversité des systèmes faiblement personnalisés est telle qu'il est impossible de les étudier dans leur ensemble, et qu'un grand nombre de ces applications utilisent des techniques de personnalisation qui ne sont pas intéressantes à étudier [1]. Notre définition de la personnalisation forte correspond à ce qui est appelé "personnalisation» dans la plupart des communautés. Cette définition nous servira de base pour définir formellement une application personnalisée.

1. Par exemple, les techniques de personnalisation impliquées dans la réalisation d'une page affichant l'heure locale de l'utilisateur en fonction de son adresse IP ou ses emails en fonction de son identifiant.

L'objectif de notre travail est d'introduire de la personnalisation forte dans un programme faiblement personnalisé, c'est-à-dire de modifier son comportement pour lui faire prendre en compte un profil.

3.2 Modélisation d'applications Web sous forme d'agents logiques

D'après notre étude de l'état de l'art, il n'existe pas de formalismes logiques adaptés à la modélisation d'applications Web. L'interaction sur le Web avec un utilisateur est pourtant un problème spécifique, bien plus qu'un simple cas particulier de l'interaction avec un utilisateur, car les technologies sur lesquelles cette interaction est fondée imposent une forme précise d'interaction : un échange de requêtes et de réponses, l'utilisateur (ou du moins, le navigateur le représentant) initiant toujours la communication par une requête.

Cette forme d'interaction n'est pas directement compatible avec les formes d'interactions classiques des agents (des agents effectuant des actions et percevant leur environnement) : il est donc nécessaire de mettre en place une adaptation permettant d'exprimer des applications Web comme des agents logiques.

Dans cette section, nous présentons notre première contribution : WAIG (web applications in Golog), un formalisme agent adapté aux applications Web. Ce formalisme est basé sur IndiGolog et comprend un certain nombre de contraintes destinées à prendre en compte les spécificités du problème.

L'expression d'une application Web sous forme d'agent présente les spécificités suivantes :

— Les réponses et réactions de l'utilisateur constituent un environnement incertain [2] (l'effet de la majorité des actions de l'agent est inconnu, le raisonnement sur ces actions est un problème complexe) ;

— L'interaction avec l'utilisateur est contrainte par le cycle requête-réponse du Web : l'application envoie une page et l'utilisateur effectue une action dessus (clique sur un bouton, un lien, ou remplit un formulaire). L'agent ne peut envoyer qu'une page à la fois, et ne peut effectuer du sensing que sur ce qui a été explicitement demandé à l'utilisateur.

3.2.1 Choix d'IndiGolog

La tâche de personnalisation a parfois été considérée comme une opération de planification, particulièrement dans les hypermédias adaptatifs. Dans [BBP04], la planification est utilisée pour fournir à des étudiants un parcours pédagogique correspondant à leurs attentes ; la personnalisation est alors présente sous la forme de contraintes sur la planification. Dans [JBP+06], l'utilisation d'un hypermédia adaptatif est exprimée en termes d'actions élémentaires, et la planification est utilisée pour proposer un parcours à l'utilisateur. Golog offre la même possibilité de recherche de plan, mais permet l'écriture d'une structure de programme ayant un certain degré de rigidité, laissant au concepteur la possibilité de choisir sur quels choix la planification se fait. Golog nous permet ainsi de choisir la granularité avec laquelle faire de la planification, plutôt que de forcément planifier à l'échelle des actions élémentaires. Cela permet d'utiliser la planification dans des programmes complexes où une

2. Au sens de Russel et Norvig [RN10]

étape unique en terme de planification est en fait constituée de plusieurs actions élémentaires au niveau du programme, ce qui nous donne une plus grande latitude dans l'expression des programmes sans pour autant perdre la planification. Golog a été intégré avec des planificateurs de pointe tels qu'ADL [CELN07], offrant ainsi l'efficacité des dernières recherches en planification pour l'exécution de programmes Golog.

Prenons par exemple le cas d'un système qui dispose d'un grand nombre de ressources, dont certaines sont les prérequis d'autres, et qui doit fournir à l'utilisateur toutes les ressources nécessaires à la maîtrise d'un sujet donné, ainsi que tous les prérequis. On peut tout d'abord résoudre ce problème à l'aide du seul calcul des situations, en définissant la présentation des ressources comme une action, et en cherchant, à l'aide d'outils de planification, une suite d'action menant à une situation où le sujet voulu est connu. On introduit le fluent $known(x,s)$ qui signifie qu'une ressource x est connue par l'utilisateur dans la situation s, et une action $present(x)$ qui montre une ressource à l'utilisateur, ayant pour effet de la rendre connue à l'utilisateur, et qui n'est possible que si l'utilisateur connaît tous les prérequis de la ressource. Formellement, cette théorie de l'action est définie par les équations (3.1) et (3.2).

$$Poss(present(x),s) \equiv \forall r.prereq(r,x) \implies known(x,s), \tag{3.1}$$

$$known(x,do(a,s)) \equiv a = present(x) \vee known(x,s). \tag{3.2}$$

Résoudre le problème consiste à chercher un plan menant à une situation dans laquelle $known(goal)$ est vrai.

Ce problème de planification peut également être exprimé identiquement en Golog ; il s'agit de chercher une exécution du programme suivant :

$$(\pi x\ present(x))^*; known(goal)?.$$

Golog permet cependant de garder la même capacité de planification tout en construisant des programmes plus complexes, où l'étape de planification n'est plus une action primitive, mais un programme Golog quelconque. Dans cet exemple, on souhaite maintenant que pour chaque ressource présentée, un exercice soit également présenté, et que le résultat soit récupéré. Une étape de parcours n'est plus $present(x)$ pour un x donné, mais $present(x); presentExercice(x); checkResult$. Le programme dont on cherche une exécution est alors :

$$(\pi x\ present(x); presentExercice(x); checkResult)^*; known(goal)?.$$

L'exécution du programme consistera à trouver une séquence d'action sous forme de répétition de $present(x), presentExercice(x), checkResult$ pour différents x. Cette recherche est identique à la précédente, bien que le programme soit plus complexe : Golog permet cette complexité sans perdre les avantages de la planification. Cela permet d'écrire l'intégralité du comportement d'un agent dans un programme Golog plutôt que de cantonner la planification à un sous-composant de l'application qui serait utilisé par d'autres composants non logiques.

Golog est un langage logique, construit sur la logique de premier ordre. Cela permet d'accéder à la logique du premier ordre lors de l'écriture de programmes en Golog, ce qui offre une grande puissance de raisonnement et permet notamment l'écriture de programmes basés sur des modèles complexes en utilisant le raisonnement pour faire apparaître des relations non explicitées dans ces modèles.

Par ailleurs, Golog a déjà été employé pour écrire des agents interagissant sur le Web [MS01, MS02] ou exprimer des systèmes hypertextes [Sch99]. Il a également été employé pour exprimer la personnalisation d'hypermédias adaptatifs [JBP+06]. Golog est un choix approprié pour notre étude.

IndiGolog est un langage de la famille de Golog qui permet l'utilisation du sensing tout en gardant les avantages de Golog. Le sensing est nécessaire pour l'interaction avec l'utilisateur sur le web. Le choix d'IndiGolog pour l'expression de la personnalisation dans le cadre du web a fait l'objet d'une communication [DPB11b].

3.2.2 Structure de l'agent interagissant sur le Web

Quand un utilisateur interagit avec une application sur le Web, il le fait à travers un navigateur qui dialogue avec un serveur HTTP. Dans la majorité des cas, le serveur fournit une abstraction, une interface, sur laquelle l'application est construite. De la même manière, nous souhaitons masquer à l'agent les détails de la communication avec l'utilisateur pour lui fournir une interface simple, permettant de considérer l'interaction avec l'utilisateur sous la forme d'actions et de sensing.

L'étendue de cette abstraction est une question intéressante. Il semble évident qu'il n'est pas de la responsabilité de l'agent de construire les documents HTML renvoyés à l'utilisateur, mais l'agent doit-il être pour autant exclu de toutes les considérations de présentation ? De même, les applications Web sont construites autour d'adresses de documents (les URL) : l'agent doit-il traiter explicitement ce concept, ou celui-ci doit-il être masqué ?

Dans tous les cas, cette abstraction exige la présente d'un composant intermédiaire qui fasse office de serveur HTTP ou communique avec un serveur HTTP, et fournisse à l'agent une interface sous forme d'actions. Nous appellerons ce composant l'intermédiaire.

Toutes les actions de l'agent ne sont pas intéressantes dans ce cas. Certaines actions n'ont pour but que de modifier l'état interne de l'agent sans chercher à interagir avec l'utilisateur. D'autres actions ont au contraire pour objet d'envoyer des informations à l'utilisateur, ou d'en recevoir. Nous distinguons ces deux types d'actions des autres.

Définition 4 (Action d'envoi)**.** Une action d'envoi est une action par laquelle l'agent cherche à envoyer des informations ou des contenus à l'utilisateur, sous la forme d'une page Web.

Définition 5 (Action de réception)**.** Une action de réception est une action par laquelle l'agent cherche à obtenir des informations de l'utilisateur, notamment celles entrées dans un formulaire ou résultant de l'interaction de l'utilisateur.

Communication entre l'agent et l'intermédiaire

Le navigateur web émet des requêtes HTTP à l'intermédiaire et reçoit en retour des réponses HTTP. L'agent émet des actions d'envoi et de réception. Ces échanges sont illustrés par la figure (3.1).

Pour masquer à l'agent les détails de la communication avec l'utilisateur, l'intermédiaire doit effectuer une correspondance entre les actions d'envoi et les actions de réception d'une part, et les requêtes et réponses d'autre part. Grossièrement, les actions d'envoi sont des envois d'information et correspondent à une réponse, tandis que les actions de réception sont des réceptions d'information et correspondent à des requêtes de l'utilisateur. Un problème se pose : dans le modèle de l'agent, toutes les actions (d'envoi comme de réception) sont à l'initiative de l'agent, tandis que, dans le protocole

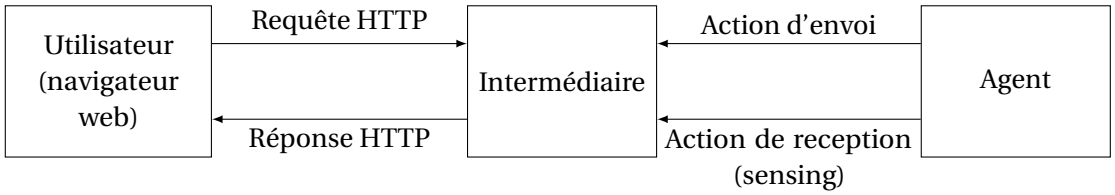

FIGURE 3.1 – Échanges entre les différents composants.

Les flèches représentent les messages échangés. Le sens des flèches indique quel composant est à l'initiative du message.

HTTP, les requêtes sont envoyées par l'utilisateur. Il est donc du ressort de l'intermédiaire de procéder à une adaptation de la séquence des évènements pour rendre ces composants compatibles.

L'intermédiaire doit donc recevoir les requêtes et actions et répondre quand c'est possible, en bloquant l'agent si nécessaire. À chaque requête correspond une ou plusieurs opérations de sensing, et à chaque action d'envoi correspond une unique réponse. La figure 3.2 illustre le détail des communications entre les l'utilisateur, l'intermédiaire et l'agent.

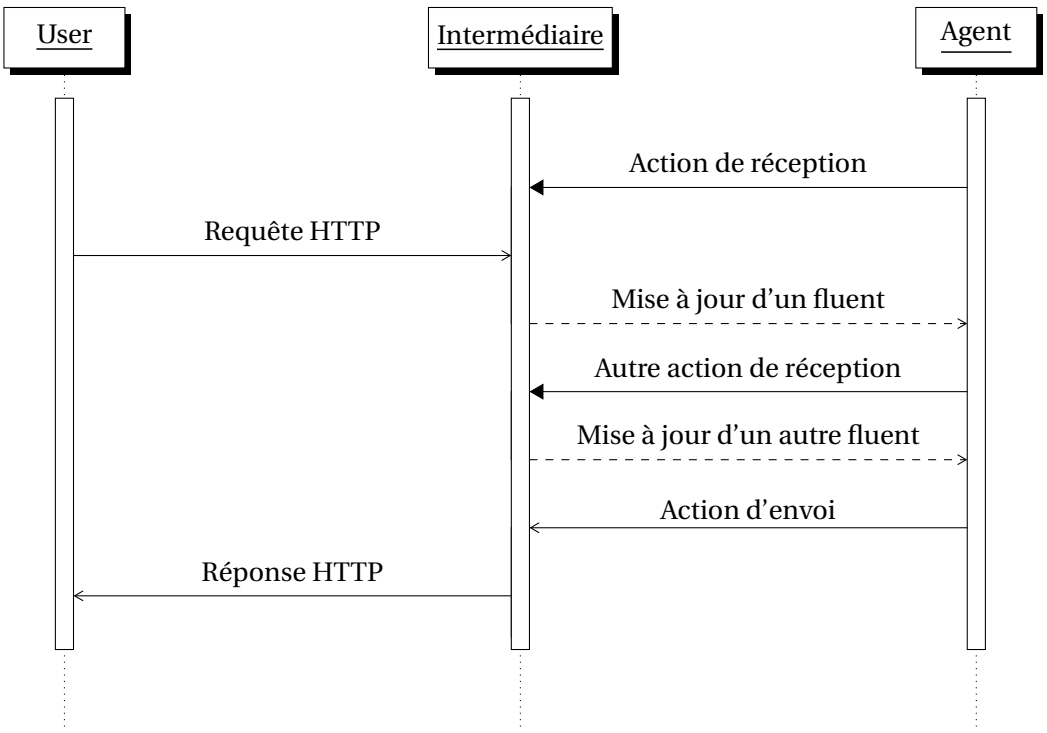

FIGURE 3.2 – Détails des communications entre l'utilisateur, l'intermédiaire, et l'agent

L'action de réception est bloquée par l'intermédiaire jusqu'à ce que l'utilisateur émette une requête HTTP. La réponse HTTP n'est renvoyée par l'intermédiaire, qu'après l'action d'envoi suivant.

Définition de l'intermédiaire d'une application

Pour que l'application puisse interagir avec l'intermédiaire, il est nécessaire de fixer les actions de l'agent que l'intermédiaire devra gérer, et comment il les gérera.

Actions d'envoi Les actions d'envoi sont les actions du calcul des situations qui correspondent à l'envoi d'une page. Ces actions peuvent avoir des arguments.

Actions de réception Les actions de réception sont les actions par lesquelles l'agent récupère de l'information. Ces actions peuvent également avoir des arguments.

Pages L'intermédiaire est défini par un ensemble de pages. À chaque action d'envoi est associée une page qui sera envoyée à l'utilisateur quand l'action correspondante est émise. En outre, à chaque page sont associées une ou plusieurs actions de réception. Ces actions de réception donneront lieu à des formulaires permettant l'entrée d'informations sur la page ; l'agent peut obtenir les informations des formulaires en effectuant une ou plusieurs actions de réception après une action d'envoi.

Cette définition de l'intermédiaire comme une liste de triplets (action d'envoi, page, actions de réception) établit un contrat entre l'intermédiaire et l'agent : l'agent ne peut effectuer une action de réception qu'après l'action d'envoi correspondante. L'intermédiaire n'a ainsi qu'à collecter les informations transmises dans la dernière requête de l'utilisateur et peut les oublier dès l'action d'envoi suivante.

Le document HTML correspondant à la page est construit par l'intermédiaire au moment de l'envoi à l'utilisateur. L'implémentation de cette construction est du ressort de l'intermédiaire, et peut être faite de n'importe quelle manière. Pour la suite, nous considérerons qu'à chaque page correspond un template : la construction du document consiste à remplacer dans le template les occurrences de `{{argument}}` par l'argument correspondant de l'action d'envoi.

3.2.3 Théorie de l'action d'une application Web

Nous présentons la théorie de l'action commune à tous les agents jouant le rôle d'application Web dans notre proposition. La théorie donnée ici n'est qu'une partie de la théorie totale de l'agent, celle-ci étant complétée par un ensemble d'actions et de fluents ne concernant pas l'interaction avec l'utilisateur.

Nous utilisons le fait que les actions du calcul des situations sont réifiées, c'est-à-dire qu'elles sont représentées par des symboles de logique du premier ordre, sur lesquelles il est possible de quantifier. Cette théorie de l'action repose sur une transcription de la définition de l'intermédiaire en logique du premier ordre, spécifique à l'application en question, qui devra être fournie à l'écriture de l'agent. Cette définition est fournie sous la forme de deux prédicats : $sendAction(a)$ indique que a est une action d'envoi, et $receptionAction(r, a)$ indique que r est une action de réception rattachée à la page envoyée par l'action a.

Tout d'abord, il est toujours possible d'envoyer une page.

$$Poss(a, s) \equiv sendAction(a) \tag{3.3}$$

Une action de réception n'est possible qu'après l'envoi de la page correspondante. Un fluent fonctionnel *lastAction*(*s*) est introduit pour garder la trace de la dernière action d'envoi effectuée.

$$Poss(r,s) \equiv \exists a\ receptionAction(r,a) \wedge lastAction(s) = a \tag{3.4}$$

$$\begin{aligned} lastAction(do(a,s)) = last \equiv \\ sendAction(a) \wedge last = a \\ \vee \neg sendAction(a) \wedge last = lastAction(s) \end{aligned} \tag{3.5}$$

Chaque action de réception r permet la récupération d'une valeur. Cette valeur sera stockée dans le fluent fonctionnel $result(r,s)$. En utilisant la modélisation de [SL03] pour le sensing, nous définissons l'axiome de l'état successeur du fluent K.

$$receptionAction(r) \implies [result(r,do(a,s)) = value \equiv a = r \vee result(r,s) = value] \tag{3.6}$$

$$\begin{aligned} K(s',do(a,s)) \equiv (\exists s'').s' = do(a,s'') \wedge K(s'',s) \wedge \\ receptionAction(a) \implies result(a,s'') = result(a,s) \end{aligned} \tag{3.7}$$

Définition 6. Une théorie de l'action d'une application Web, notée $\mathscr{W}$, est composée des axiomes génériques (donnés en 3.4, 3.5, 3.6 et 3.7), et d'une définition des prédicats *sendAction* et *receptionAction* dérivée de la définition d'un intermédiaire.

L'incorporation d'une théorie de l'action d'une application Web dans une théorie de l'action permet donc l'utilisation des actions d'envoi et de réception pour interagir avec l'intermédiaire, et donc l'utilisateur.

3.2.4 Illustration : bonjour, monde

Pour illustrer ce formalisme, nous présentons tout d'abord un agent simple : il envoie une page contenant le texte «bonjour, monde», demande son nom à l'utilisateur, puis le salue directement.

Cette application est composée des éléments suivants :

— une action d'envoi, *welcome*(*name*), qui accueille l'utilisateur en l'appelant par le nom donné en argument ;

— une page *welcomePage* contenant l'affichage du message et un formulaire demandant son nom à l'utilisateur ;

— une action de réception *askName* qui récupère la valeur contenue dans le formulaire.

La définition de l'intermédiaire est une liste contenant un seul triplet, constitué de ces trois éléments :

$$[\langle welcome(name), welcomePage, [askName]\rangle]. \tag{3.8}$$

Nous laissons de côté l'implémentation de l'intermédiaire, mais donnons tout de même le template HTML pour la page *welcomePage*, pour donner une idée :

```
Bonjour, {{name}}.

<form>
```

```
  <input type="text" name="name">
  <button type="submit">Submit</button>
</form>
```

Alors, en utilisant une théorie de l'action basée uniquement sur $\mathscr{W}$, le programme IndiGolog de l'agent est :

$$welcome(\text{“monde"}); askName; welcome(result(askName)). \tag{3.9}$$

3.2.5 Illustration : agent gérant un hypertexte

Nous illustrons le formalisme avec un exemple plus complet, inspiré d'un illustration de l'usage de Golog [Sch99] : un *hypertexte*. D'après [BK94], un *hypertexte* consiste en un nombre quelconque de nœuds et de liens. Les nœuds sont des objets déclarés dans une base de données qui sont représentés comme du texte à l'écran. Les liens décrivent des relations entre des paires de nœuds (appelés source et destination), également représentés dans une base de données.

Les fluents représentant un hypertexte sont les suivants [3] :

1. $node(x, y, s)$ x est l'identifiant d'un nœud dont le contenu est y dans la situation s ;
2. $link(w, x, y, z, s)$ w est l'identifiant d'un lien entre le nœud source x et le nœud destination y et ayant pour intitulé z dans la situation s ;
3. $currentNode(x, s)$ x est l'identifiant du nœud courant dans la situation s ;
4. $homeNode(x, s)$ x est l'identifiant du nœud de départ (le nœud par lequel commence la navigation) dans la situation s (n'est modifié par aucune action).

Considérons par exemple l'hypertexte suivant, conçu pour une personne intéressée par l'histoire du Saint-Empire Romain Germanique [Wik14]. Les définitions suivantes sont déclarées être vrais dans la situation initiale S_0.

— $node$(0, “Le Saint-Empire romain germanique ou Saint-Empire romain de la nation germanique (en allemand : Heiliges römisches Reich deutscher Nation, en latin : Sacrum romanum Imperium Nationis germanicæ) ; également appelé parfois Premier Reich ou Vieil Empire, pour le différencier de l'Empire allemand, est un regroupement politique des terres d'Europe occidentale et centrale au Moyen Âge, dirigé par l'Empereur romain germanique mais aujourd'hui disparu. Il se voulait, au Xe siècle, l'héritier de l'Empire d'Occident des Carolingiens, mais également de l'Empire romain. L'adjectif Saint n'apparaît que sous le règne de Frédéric Barberousse (attesté en 1157) pour légitimer le pouvoir de manière divine.”, S_0),

— $node$(1, “Le Saint Empereur Romain, est le vocable sous lequel on désigne le souverain du Saint-Empire romain. Il était élu par un collège de princes appelés pour cette raison princes électeurs. Du XIVe siècle au XVIIe siècle, ces princes sont au nombre de sept.”, S_0),

— $node$(2, “Frédéric Ier de Hohenstaufen, dit Frédéric Barberousse (en allemand : Friedrich I., Barbarossa, 1122 – 10 juin 1190), fut empereur romain germanique, roi d'Allemagne (Rex Romanorum), roi d'Italie, duc de Souabe et duc d'Alsace, comte palatin de Bourgogne.”, S_0),

3. Pour garder l'exemple simple, ils sont simplifiés par rapport à [Sch99], délaissant les annotations sémantiques des nœuds et liens.

— $link(0,0,1,$ “Saint Empereur”$, S_0)$,

— $link(1,0,2,$ “Frédéric Barberousse”$, S_0)$,

— $link(2,2,1,$ “empereur romain germanique”$, S_0)$,

— $currentNode(0, S_0)$,

— $homeNode(0, S_0)$.

Les actions permettant de manipuler ces fluents sont les suivantes :

1. $traverse(x, y)$ déplace le nœud courant de x vers y ;
2. *goHome* définit le nœud de départ comme nœud courant ;
3. $createNode(x, y)$ ajoute le nœud x avec comme contenu y à la base de données ;
4. $deleteNode(x)$ supprime le nœud x ;
5. $createLink(w, x, y, z)$ ajoute le lien w entre les nœuds x et y avec comme intitulé z ;
6. $deleteLink(w)$ supprime le lien w.

Les interactions entre les actions et les fluents sont évidentes, et la théorie de l'action correspondante est constituée des formules suivantes [4].

Il est possible de passer d'un nœud x à un nœud y si le nœud courant est x et s'il existe un lien entre x et y.

$$Poss(traverse(x, y), s) \equiv currentNode(x, s) \wedge \exists w\ link(w, x, y, \_, s); \tag{3.10}$$

Il est toujours possible de revenir au nœud de départ.

$$Poss(goHome, s) \equiv true; \tag{3.11}$$

Il est possible de créer un nœud d'identifiant x si aucun nœud ne porte déjà l'identifiant x.

$$Poss(createNode(x, y), s) \equiv \neg\exists y'\ node(x, y', s); \tag{3.12}$$

Il est possible de supprimer un nœud x s'il existe.

$$Poss(deleteNode(x), s) \equiv \exists y\ node(x, y, s); \tag{3.13}$$

Il est possible de créer un lien d'identifiant w si aucun lien ne porte déjà l'identifiant w.

$$Poss(createLink(w, x, y, z), s) \equiv \neg\exists x', y', z'\ link(w, x', y', z', s); \tag{3.14}$$

Il est possible de supprimer un lien s'il existe.

$$Poss(deleteLink(w), s) \equiv \exists x, y, z\ link(w, x, y, z, s); \tag{3.15}$$

Le fluent *currentNode* est affecté par les actions *traverse* et *goHome*, et aucune autre.

$$\begin{aligned} currentNode(x, do(a, s)) \equiv\ & a = traverse(y, x) \\ & \vee\ a = goHome \wedge x = homeNode \\ & \vee\ (a \neq traverse(y, x) \wedge \neq goHome \wedge currentNode(x, s)); \end{aligned} \tag{3.16}$$

4. Pour simplifier les formules, nous utilisons l'underscore _ avec la même signification que la variable anonyme de Prolog. On peut revenir à une formule classique en remplaçant chaque underscore par une variable unique, quantifiée universellement sur l'ensemble de la formule

Le fluent *homeNode* ne change jamais.

$$homeNode(x, do(a, s)) \equiv homeNode(x, s); \tag{3.17}$$

Le fluent *node* est affecté par les actions de création et de suppression de nœud.

$$\begin{aligned} node(x, y, do(a, s)) \equiv\ & a = createNode(x, y) \\ & \vee (a \neq deleteNode(x) \wedge node(x, y, s)); \end{aligned} \tag{3.18}$$

Le fluent *link* est affecté par les actions de création et de suppression de lien.

$$\begin{aligned} link(w, x, y, z, do(a, s)) \equiv\ & a = createLink(w, x, y, z) \\ & \vee (a \neq deleteLink(w) \wedge link(w, x, y, z, s)). \end{aligned} \tag{3.19}$$

L'utilisateur accède à cet hypertexte à travers une interface Web. Chaque nœud est représenté par une page. En plus du contenu du nœud, la page contient la liste des liens dont le nœud est source. L'utilisateur peut éditer l'hypertexte : chaque page contient des boutons permettant de détruire le nœud correspondant et les liens partant de ce nœud, ainsi que des boutons permettant d'accéder à des formulaires de création de nœuds et de liens.

On compte donc trois pages [5] et trois actions d'envoi correspondantes.

— Une page représentant un nœud, et une action d'envoi *displayNode*(*content*, *links*), qui affiche le contenu *content* du nœud, et la liste de liens *links*, qui est une liste de paires (nœud, intitulé du lien). Les actions de réception correspondantes sont :
  — *askLinkClicked*, si l'utilisateur a cliqué sur un lien, renvoie le nœud vers lequel ce lien pointe,
  — *askDeleteNode*, qui renvoie vrai si l'utilisateur a supprimé le nœud,
  — *askDeleteLink*, si l'utilisateur a supprimé un lien, renvoie le nœud vers lequel ce lien pointe,
  — *askCreateNode* qui renvoie vrai si l'utilisateur a demandé la création d'un nœud, et
  — *askCreateLink* qui renvoie vrai si l'utilisateur a demandé la création d'un lien.
— Une page de création de nœuds, et une action d'envoi *createNodeForm*. L'action de réception correspondante est *askNodeContent*, qui renvoie le contenu du nœud.
— Une page de création de liens, et une action d'envoi *createLinkForm*(x), où x est le nœud suggéré comme nœud d'origine pour la création du lien (quand un utilisateur crée un lien, l'agent propose le nœud actuellement affiché comme nœud d'origine du lien). Les actions de réception correspondantes sont :
  — *askLinkSource* qui renvoie la source du lien,
  — *askLinkTarget* qui renvoie la destination, et
  — *askLinkTitle* qui en renvoie l'intitulé.

5. Certaines pages peuvent avoir des arguments, une page de l'intermédiaire peut donc amener à la génération d'un grand nombre de pages différentes pour l'utilisateur.

La liste de triplets définissant l'intermédiaire est donc la suivante :

$$
\begin{array}{l}
[\langle \mathit{displayNode}(\mathit{content}, \mathit{links}), \mathit{nodePage}, [\\
\quad \mathit{askLinkClicked}, \\
\quad \mathit{askDeleteNode}, \\
\quad \mathit{askDeleteLink}, \\
\quad \mathit{askCreateNode}, \\
\quad \mathit{askCreateLink}]\rangle, \\
\langle \mathit{createNodeForm}, \mathit{nodeForm}, [\mathit{askNodeContent}]\rangle, \\
\langle \mathit{createLinkForm}(x), \mathit{linkForm}, [\mathit{askLinkSource}, \mathit{askLinkTarget}, \mathit{askLinkTitle}]\rangle]. \qquad (3.20)
\end{array}
$$

Nous décrivons maintenant le programme agent utilisant la théorie de l'action de l'hypertexte et la théorie de l'action des applications Web pour fournir à l'utilisateur l'accès à l'hypertexte.

Le programme commence par appeler l'action *displayNode* pour envoyer à l'utilisateur la page correspondant au nœud en cours et contenant ses liens. Le programme appelle ensuite toutes les actions de réception correspondant à cette page pour charger les fluents correspondants, et ainsi savoir sur quel bouton l'utilisateur a cliqué. Un choix est ensuite fait entre les différentes possibilités : l'utilisateur n'ayant cliqué que sur un bouton, un seul de ces sous-programmes est exécutable. Par exemple, l'action $\mathit{traverse}(n, \mathit{result}(\mathit{askLinkClicked}))$ n'est valide que si l'utilisateur a cliqué sur un lien, et que le fluent $\mathit{result}(\mathit{askLinkClicked})$ a pour valeur un nœud lié à partir du nœud courant. De même, la séquence d'actions $\mathit{result}(\mathit{askDeleteNode})?; \mathit{deleteNode}(n); \mathit{goHome}$ n'est possible que si le fluent $\mathit{result}(\mathit{askDeleteNode})$ est vrai, c'est-à-dire si l'utilisateur a cliqué sur le bouton de suppression de page.

$$
\begin{array}{l}
(\\
\quad \pi n\, \pi l\ \mathit{currentNode}(n) \wedge \mathit{links}^{6}(n, l)?; \mathit{displayNode}(n, l); \\
\quad \mathit{askLinkClicked}; \mathit{askDeleteNode}; \mathit{askDeleteLink}; \mathit{askCreateNode}; \mathit{askCreateLink} \\
\quad (\\
\quad\quad \mathit{traverse}(n, \mathit{result}(\mathit{askLinkClicked})) \mid \\
\quad\quad \mathit{result}(\mathit{askDeleteNode})?; \mathit{deleteNode}(n); \mathit{goHome} \mid \\
\quad\quad \pi w, z\ \mathit{link}(w, n, \mathit{result}(\mathit{askDeleteLink}), z)?; \mathit{deleteLink}(w) \mid \\
\quad\quad \mathit{result}(\mathit{askCreateNode})?; \mathit{doCreateNode} \mid \\
\quad\quad \mathit{result}(\mathit{askCreateLink})?; \mathit{doCreateLink}(n) \\
\quad) \\
)* \qquad (3.21)
\end{array}
$$

La procédure suivante envoie la page de création de nœuds à l'utilisateur, puis récupère le contenu

6. $\mathit{links}(n, l)$ est vrai si l est la liste des liens partant de n. C'est un prédicat spécial, implémenté en Prolog à l'aide du prédicat `findall`.

rentré par l'utilisateur et crée un nœud à partir de celui-ci, avec un nouvel identifiant.

$$\begin{array}{l}\textbf{proc}\ \mathit{doCreateNode}\\ \quad \mathit{createNodeForm}; \mathit{askNodeContent};\\ \quad \pi x\ (\neg\exists y\ \mathit{node}(x,y))?; \mathit{createNode}(x, \mathit{result}(\mathit{askNodeContent}))\\ \textbf{endProc}\end{array} \tag{3.22}$$

De la même manière, la procédure suivante envoie la page de création de liens, lit toutes les entrées de l'utilisateur, et crée un lien avec un nouvel identifiant.

$$\begin{array}{l}\textbf{proc}\ \mathit{doCreateLink}(\mathit{defaultNode})\\ \quad \mathit{createLinkForm}(\mathit{defaultNode}); \mathit{askLinkSource}; \mathit{askLinkTarget}; \mathit{askLinkTitle}\\ \quad \pi w\ (\neg\exists x, y, z\ \mathit{link}(w,x,y,z)?;\\ \quad\quad \mathit{createLink}(w, \mathit{result}(\mathit{askLinkSource}), \mathit{result}(\mathit{askLinkTarget}), \mathit{result}(\mathit{askLinkTitle}))\\ \textbf{endProc}\end{array} \tag{3.23}$$

3.2.6 Conclusion

Nous avons défini une architecture permettant l'expression d'applications Web modélisées par des agents. Un composant intermédiaire s'occupe de la traduction pour offrir à l'agent et à l'utilisateur des interfaces appropriées : il échange des réponses et des requêtes avec le navigateur de l'utilisateur, et gère les actions et le sensing de l'agent. L'agent, écrit en IndiGolog, envoie des pages à l'utilisateur en exécutant des actions et reçoit les réponses de l'utilisateur par le sensing.

Ce formalisme permet de s'affranchir des spécificités de l'interaction sur le Web et d'exprimer un agent modélisant une application web de la même manière que n'importe quel autre agent IndiGolog. Cette généralité de l'expression nous permettra de définir aux chapitres suivants des transformations s'appliquant non seulement sur les applications Web, mais sur n'importe quel programme agent écrit en IndiGolog.

Chapitre 4

Choix d'un agent et altération des choix

Un agent rationnel est un agent ayant un objectif et utilisant les moyens à sa disposition pour l'atteindre. Pour atteindre cet objectif, il doit maximiser une fonction d'utilité de l'agent. Un agent dispose généralement de marges de manœuvre et doit choisir entre plusieurs modi operandi qui mènent tous au but. Si deux options satisfaisant le but ont la même utilité, l'agent ne dispose pas de moyens de les départager, et choisira de manière arbitraire, par exemple aléatoirement, ou en prenant toujours le premier choix dans l'ordre d'évaluation des options. Si deux options ont la même utilité, favoriser l'une ou l'autre ne change rien à l'efficacité de l'agent. Il est donc possible de doter l'agent d'un mécanisme modifiant la façon dont l'agent fait ses choix afin d'altérer son comportement sans pour autant nuire à son but.

Exemple

Prenons par exemple un robot dont la tâche est de parcourir une maison composée de nombreuses pièces. Le robot n'est pas obligé de visiter toutes les pièces, ce qui lui laisse de la marge de manœuvre dans l'accomplissement de son but. Un robot pourvu du trait psychologique «prudent» pourrait refuser de rentrer dans une pièce verrouillée, tandis qu'un robot pourvu du trait «brutal» pourrait au contraire défoncer les portes verrouillées. Les deux agents atteignent leur but en visitant la maison, mais chacun l'a fait à sa manière en faisant des choix différents. La figure 4.1 illustre cette idée en présentant tous les choix possibles pour un agent, et les chemins que des agents ayant des traits spéciaux pourraient préférer emprunter.

De tels choix apparaissent directement dans l'expression d'un programme Golog, sous la forme d'opérateurs de choix non-déterministes. Ces opérateurs créent autant de branchements dans l'exécution qu'il y a de choix possibles. L'agent choisit alors un des branchements qui mènent à une exécution complète du programme, choisissant arbitrairement si plusieurs options sont possibles. La plupart des implémentations de Golog, étant codées en Prolog, héritent du comportement de Prolog pour la résolution de tels choix : les options sont évaluées dans l'ordre dans lequel elles apparaissent dans le code, et la première option valide est utilisée. En modifiant la manière dont Golog résout les choix

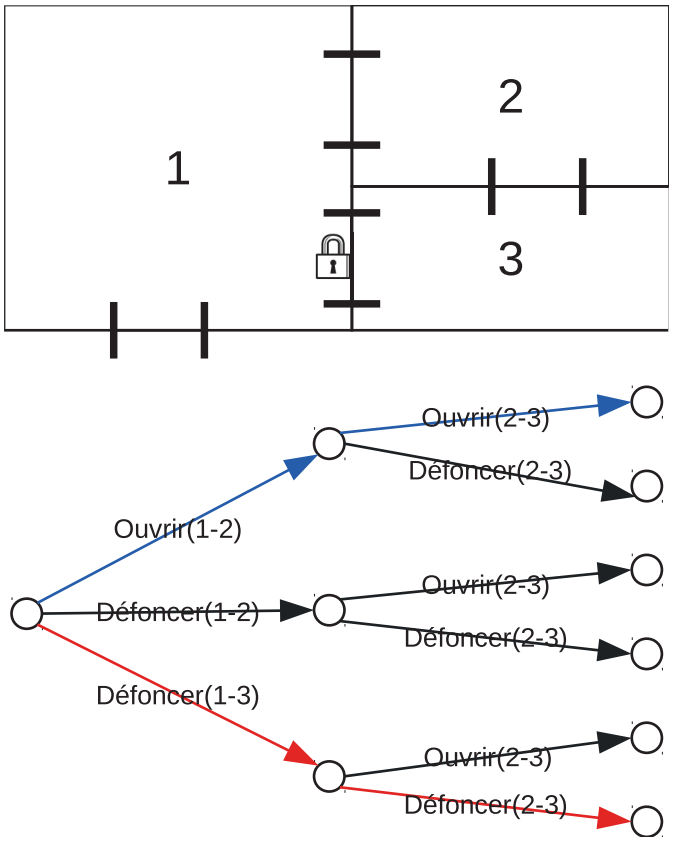

FIGURE 4.1 – Arbre des choix possibles pour le robot visitant une maison. Les chemins préférés par l'agent prudent sont en bleu. Un chemin préféré par l'agent brutal est en rouge.

non-déterministes, il est possible de modifier le comportement des programmes sans en modifier la fonctionnalité.

Pour agir sur les choix, deux possibilités se présentent : la modification du programme, ou la modification de l'interprétation du programme. La modification de l'interpréteur permet d'utiliser des informations concernant la situation courante au moment du choix, tandis qu'une modification du programme ne peut se baser que sur la structure elle-même du programme. Cependant, modifier l'interpréteur exige de valider chaque modification afin de s'assurer que l'interpréteur reste correct, alors qu'il est bien plus facile de prouver qu'un programme modifié reste conforme à l'original. Nous souhaitons étudier différentes manières d'altérer les choix, au-delà du simple réordonnancement des options, nous choisissons donc d'étudier la modification de programme qui permet de traiter plus facilement des modifications plus complexes.

Cette étude des manières d'altérer un programme pour le personnaliser a fait l'objet d'une communication [DPB11a].

4.1 Extension nécessaire de Golog

Golog dispose d'un opérateur de choix non-déterministe de programmes noté |. Le programme $\delta_1 \mid \delta_2$ exprime que soit δ_1 soit δ_2 doit être exécuté. Il n'est cependant pas possible d'exprimer une préférence sur cette exécution. Si δ_1 et δ_2 sont possibles, rien ne spécifie lequel des deux sera exécuté et le choix sera fait de manière arbitraire.

Nous introduisons un opérateur de choix de programmes avec préférence, noté $\rangle$. Le programme $\delta_1 \rangle \delta_2$ signifie «essayer δ_1, sinon δ_2» : l'un des deux programmes est exécuté, avec une préférence pour δ_1 si δ_1 et δ_2 sont tous les deux possibles. Le tableau 4.1 détaille tous les cas possibles. Il est à noter que, comme le fait remarquer [GLLS09], l'implémentation de Golog évalue les branches de gauche à droite, donc l'implémentation de | a le même comportement que $\rangle$. Cependant, il est important de distinguer les cas où le choix est indifférent et les cas où l'ordre d'évaluation des options a une importance. Ce fait pourra tout de même être utilisé pour simplifier l'implémentation en gardant l'implémentation de | plutôt que de prendre la sémantique formelle de $\rangle$ qui est beaucoup plus complexe.

δ_1 possible ?	δ_2 possible ?	exécution de $\delta_1 \mid \delta_2$	exécution de $\delta_1 \rangle \delta_2$
oui	oui	δ_1 ou δ_2	δ_1
oui	non	δ_1	δ_1
non	oui	δ_2	δ_2
non	non	impossible	impossible

TABLE 4.1 – Exécutions des opérateurs de choix

Cette construction est similaire à la construction $\mathbf{lex}(\delta_1, \delta_2)$ définie dans [FM06] (dont nous parlions page 40), mais son implémentation est différente, puisque les formalismes sont différents. L'idée est également similaire à la construction $\rhd$ de CANPlan [SP07], à ceci près que dans $\delta_1 \rangle \delta_2$, aucune partie de δ_1 n'est exécutée si δ_1 n'est pas entièrement exécutable.

La sémantique formelle de l'opérateur est donnée dans les équations (4.1) et (4.2). À titre de comparaison, la sémantique de l'opérateur | [GLL00] est rappelée en (4.3) et (4.4).

$$\begin{aligned} \mathit{Final}(\delta_1 \rangle \delta_2, s) \equiv\ & \mathit{Final}(\delta_1, s) \\ & \vee \neg \exists s".\mathit{Do}(\delta_1, s, s") \wedge \mathit{Final}(\delta_2, s) \end{aligned} \tag{4.1}$$

$$\begin{aligned} \mathit{Trans}(\delta_1 \rangle \delta_2, s, \delta', s') \equiv & \\ & \exists s''.\mathit{Do}(\delta_1, s, s'') \wedge \mathit{Trans}(\delta_1, s, \delta', s') \\ & \vee \neg \exists s''.\mathit{Do}(\delta_1, s, s'') \wedge \mathit{Trans}(\delta_2, s, \delta', s') \end{aligned} \tag{4.2}$$

$$\mathit{Final}(\delta_1 \mid \delta_2, s) \equiv \mathit{Final}(\delta_1, s) \vee \mathit{Final}(\delta_2, s) \tag{4.3}$$

$$\mathit{Trans}(\delta_1 \mid \delta_2, s, \delta', s') \equiv \mathit{Trans}(\delta_1, s, \delta', s') \vee \mathit{Trans}(\delta_2, s, \delta', s')) \tag{4.4}$$

Nous rappelons que $\mathit{Do}(\delta, s, s')$, dont la définition est donnée dans la formule (2.4) page 32, signifie que l'exécution complète du programme δ dans la situation s est possible et se termine dans la situation s'. Le terme $\mathit{Final}(\delta, s)$ signifie que le programme δ peut être considéré comme terminé dans la situation s. La définition de $\mathit{Final}(\delta_1 \rangle \delta_2, s)$ donnée en (4.1) déclare que le programme $\delta_1 \rangle \delta_2$ est terminé si δ_1 est terminé, ou si δ_1 n'est pas exécutable et que δ_2 est terminé. Le terme $\mathit{Trans}(\delta, s, \delta', s')$ signifie que l'exécution d'une étape élémentaire de δ dans s résulte dans la situation s', avec le programme δ' restant à être exécuté. La définition de $\mathit{Trans}(\delta_1 \rangle \delta_2, s, \delta', s')$ donnée en (4.2) déclare que si δ_1 est intégralement exécutable dans s, une étape de δ_1 sera exécutée, sinon, c'est une étape de δ_2

qui le sera. Ces deux définitions peuvent se lire «exécuter $\delta_1 \rangle \delta_2$ dans la situation s est équivalent à exécuter δ_1 si δ_1 peut être entièrement exécuté dans s, sinon, à exécuter δ_2».

4.2 Transformations

Comme expliqué précédemment, il est possible de modifier le comportement d'un agent en modifiant la manière dont il fait ses choix, sans pour autant altérer sa fonctionnalité. Nous décrivons ici les transformations de programmes Golog que nous avons mises au point pour modifier la manière dont les choix sont faits dans le programme.

Ces transformations sont de la forme $a \rightarrow b$; l'application d'une transformation consiste à remplacer un programme de la forme a par la forme b équivalente. Cette transformation peut se faire sur le programme lui-même, ou sur une partie du programme. Ainsi, une transformation de la forme $a \rightarrow b$ permet de transformer le programme $\delta_1 ; (a \mid \delta_2)$ en $\delta_1 ; (b \mid \delta_2)$.

Avant tout, pour modifier la manière dont les choix sont faits, il faut que l'agent ait des choix : un programme doit contenir des opérateurs de choix non-déterministes pour pouvoir être modifié. Si ce n'est pas le cas, le programme n'est de toute façon qu'une séquence d'ordres avec une seule exécution possible, et on ne pourrait modifier le programme qu'en introduisant de nouveaux éléments, ce qui ne permettrait pas de garantir que l'exécution du programme modifié ne s'éloigne pas excessivement des exécutions permises par le programme original [1].

4.2.1 Transformations simples

Les choix en Golog apparaissent sous deux formes possibles : les deux opérateurs de choix non-déterministes $\mid$ et π. Le programme $\delta_1 \mid \delta_2$ demande à l'agent de choisir entre l'exécution de δ_1 et celle de δ_2. Le programme $\pi x \, \delta(x)$ demande à l'agent de choisir une valeur de x pour laquelle exécuter $\delta(x)$. Nous introduisons deux transformations pour altérer les choix de ces deux constructions.

Tout d'abord, il est possible de remplacer un choix non-déterministe de programme par une préférence. Comme expliqué dans la section 4.1, cette transformation T1 n'empêche pas le programme de fonctionner : si l'option préférée n'est pas possible, l'autre sera exécutée. La transformation T1 a pour unique effet de forcer un ordre d'évaluation entre les deux choix.

Transformation 1 (T1)**.** $\delta_1 \mid \delta_2 \rightarrow \delta_i \rangle \delta_j \qquad i, j \in \{1,2\}, i \neq j$

On remarquera que si $\delta_1 \mid \delta_2$ peut s'exécuter, alors $\delta_1 \rangle \delta_2$ le peut aussi. C'est une propriété intéressante, car elle garantit que le remplacement d'un choix non-déterministe par un choix avec préférence dans un programme n'empêche pas le programme de s'exécuter. Intuitivement, si $\delta_1 \mid \delta_2$ a une exécution possible, alors soit δ_1 soit δ_2 ont une exécution possible, donc $\delta_1 \rangle \delta_2$ en a une aussi. Formellement, «δ peut s'exécuter dans une situation s» correspond à la propriété $\exists s'.Do(\delta, s, s')$.

Théorème 1. *Si $\delta_1 \mid \delta_2$ peut s'exécuter, alors $\delta_1 \rangle \delta_2$ le peut aussi :*

$$\exists s'.Do(\delta_1 \mid \delta_2, s, s') \implies \exists s''.Do(\delta_1 \rangle \delta_2, s, s''). \tag{4.5}$$

Démonstration. En annexe B.1. □

1. Cela est vrai dans le cas général. Cependant, nous étudions au chapitre 4.3 sous quelles conditions il est possible de modifier un programme en ajoutant de nouveaux éléments sans en altérer la fonctionnalité

La seconde transformation T2 change l'ordre d'évaluation des arguments lors d'un choix non-déterministe d'arguments, de manière à ce que les arguments x vérifiant $\phi(x)$ soient évalués en premier.

Il n'est pas nécessaire d'introduire d'équivalent personnalisé de l'opérateur de choix non-déterministe d'argument π : la préférence peut s'exprimer à l'aide de l'opérateur $\rangle$. Ainsi, l'exécution d'un programme $\delta(v)$ pour une certaine valeur de l'argument v, en préférant les cas où $\phi(v)$ est satisfait peut s'exprimer ainsi :

$$\begin{array}{l} \pi\, v\, \phi(v)?;\delta(v) \\ \quad \rangle \\ \pi\, v\, \delta(v). \end{array} \tag{4.6}$$

Deux sous-programmes sont possibles : le premier cherche un v ($\pi\, v$) tel que $\phi(v)$ et exécute $\delta(v)$ pour celui-ci, ce sous-programme est préféré au second qui cherche un v quelconque et exécute $\delta(v)$. Cela signifie que s'il existe un v tel que $\phi(v)$ pour lequel l'exécution de $\delta(v)$ est possible, alors c'est un tel v qui sera choisi, sinon, c'est un v quelconque. Cette construction permet donc d'établir une préférence sur le choix de v. La transformation suivante remplace donc $\pi\, v\, \delta(v)$ par une cette construction. Dans le pire des cas, c'est $\pi\, v\, \delta(v)$ qui est exécuté, et le programme revient à son comportement original.

Transformation 2 (T2)**.** $\pi x.\delta(x) \rightarrow \pi x.\phi(x)?;\delta(x)\rangle\pi x.\delta(x)$

On peut remarquer qu'en cas de rabattement sur la partie droite, l'exécution est à nouveau évaluée pour les arguments x tels que $\phi(x)$, bien qu'à ce stade on soit sûr qu'ils ne marcheront pas. On pourrait rajouter la condition $\neg\phi(x)$ à la partie droite pour éviter la répétition, mais il n'est pas certain que cela représente un réel gain de temps, ϕ et δ pouvant être quelconques. Nous préférons la formulation actuelle qui explicite le fait qu'en cas d'échec de la partie gauche, l'exécution revient sur le programme original ($\pi x.\delta(x)$).

4.2.2 Transformations destructrices

Le théorème 1 garantit que si un programme est exécutable, le résultat de sa transformation par T1 ou T2 sera également exécutable. Cependant, il peut être désirable de restreindre l'exécution en interdisant certaines options, ce qui conduit à perdre cette propriété. Par exemple, dans le cas du robot visitant une maison (décrit en annexe A.1), on souhaite pouvoir modifier le programme pour empêcher le robot de défoncer des portes. Nous introduisons les transformations T3 et T4, qui sont similaires aux transformations T1 et T2, à ceci près que la partie droite a été supprimée. Ainsi, si l'option préférée n'est pas envisageable, alors l'exécution du programme n'est pas possible.

Transformation 3 (T3)**.** $\delta_1 \mid \delta_2 \rightarrow \delta_i$ avec $i \in \{1,2\}$

Transformation 4 (T4)**.** $\pi x.\delta(x) \rightarrow \pi x.\phi(x)?;\delta(x)$

Dans T3, plutôt que de spécifier une préférence sur le programme à exécuter, on spécifie une obligation, sous la forme du retrait pur et simple de l'autre option du programme. De la même manière, le programme transformé par T4 ne permet de choisir l'argument que parmi les x tels que $\phi(x)$, plutôt que prendre x quelconque.

Ces transformations suppriment des possibilités de l'agent, soit en retirant du code, soit en imposant des contraintes limitant les choix possibles. Appliquées à tort, elles peuvent casser un programme en retirant des parties cruciales à son exécution. De ce fait, leur application ne doit se faire qu'avec soin. Dans la suite, nous veillerons à ce que ces transformations ne soient appliquées que dans des cas précis [2], quand cela est demandé explicitement, afin d'éviter de générer des programmes inopérants.

4.2.3 Report du choix à l'exécution

Ces transformations permettent de modifier un programme pour modifier les choix de l'agent. Cette transformation est faite sur le programme avant son exécution. Il est arrivé cependant qu'il ne soit pas possible de décider avant l'exécution quel choix faire, quand le choix dépend de la situation. On veut par exemple transformer un programme $\delta_1 \mid \delta_2$ de manière à ce que δ_1 soit préféré à δ_2 seulement quand une formule-fluent ϕ est vraie. Nous introduisons pour cela une transformation utilitaire permettant de faire apparaître explicitement dans le programme les différents cas possibles de manière à les traiter différemment avec les autres transformations.

Transformation 5 (T5)**.** $\delta \rightarrow$ **if** ϕ **then** δ **else** δ

Cette transformation permet de transformer un programme $\delta_1 \mid \delta_2$ en **if** ϕ **then** $\delta_1 \mid \delta_2$ **else** $\delta_1 \mid \delta_2$ dans un premier temps, l'application de la transformation T1 à ce résultat permet d'obtenir le programme **if** ϕ **then** $\delta_1 \rangle \delta_2$ **else** $\delta_2 \rangle \delta_1$, exprimant ainsi «si ϕ est vrai, préférer δ_1, sinon préférer δ_2».

4.2.4 Validité des transformations

Il est important que les transformations ne fassent que préciser les choix et n'ajoutent pas de fonctionnalités. Cela signifie que toute exécution d'un programme transformé doit être une exécution valide pour le programme original. Nous définissons formellement cette propriété avant de prouver que les transformations proposées en sont bien pourvues.

Définition 7. Soient δ_1 et δ_2 des programmes Golog. Le programme δ_2 est une restriction du programme δ_1, notée $restriction(\delta_2, \delta_1)$, si toute exécution de δ_2 est aussi une exécution valide de δ_1, c'est à dire :

$$Final(\delta_2, s) \implies Final(\delta_1, s),$$

$$Trans(\delta_2, s, \delta_2', s') \implies \exists \delta_1' \; Trans(\delta_1, s, \delta_1', s') \wedge restriction(\delta_2', \delta_1').$$

Cette définition se lit ainsi :

— si δ_2 est dans un état final, alors δ_1 est aussi dans un état final ;

— si l'exécution d'une étape élémentaire de δ_2 dans la situation s peut mener à la situation s' avec δ_2' restant à être exécuté, alors l'exécution d'une étape élémentaire de δ_1 dans s peut mener à la même situation s' (c'est donc la même étape élémentaire), avec δ_1' restant à exécuter, et δ_2' est lui-même une restriction de δ_1'.

Théorème 2. *Si δ_2 est une restriction de δ_1, alors pour tout programme δ, $\delta^{\delta_1}_{\delta_2}$ est une restriction de δ, $\delta^{\delta_1}_{\delta_2}$ étant obtenu en remplaçant chaque occurrence de δ_1 par δ_2 dans δ.*

2. Dans notre approche, un processus sélectionne les transformations à appliquer. Cette contrainte ne sera qu'une des règles permettant de sélectionner des transformations.

Démonstration. La démonstration se fait par induction structurelle sur les différentes constructions de Golog, et est triviale dans chaque cas. □

Ce théorème a pour conséquence que l'application d'une transformation peut se faire au sein d'un programme (en transformant une partie du programme).

Théorème 3. *Soit δ_t le programme obtenu par transformation de δ avec la transformation T1, T2, T3, T4 ou T5. Alors δ_t est une restriction de δ :*

Démonstration. Démonstration en annexe B.2 □

Ces transformations permettent de transformer un programme et peuvent être appliquées plusieurs fois, c'est-à-dire en appliquant une transformation au résultat d'une autre transformation. Il est ainsi possible d'utiliser ces transformations pour opérer des changements complexes des programmes, ce que nous ferons dans les chapitres suivants.

4.2.5 Conclusion

Nous avons défini un ensemble de transformations qui, appliquées à un programme IndiGolog, permettent de modifier la manière dont sont faits les choix d'un agent. Ces transformations font partie de PAGE process.

Dans le cadre d'un agent, la personnalisation consiste à faire des choix différents pour des utilisateurs différents, de manière à augmenter l'utilité fournie à l'utilisateur. Les modifications de l'agent permises par ces transformations peuvent donc permettre de personnaliser un programme agent.

4.3 Ajout de choix

Les transformations précédentes créent des variations de programmes en jouant sur la manière dont les choix sont faits dans les programmes. Cela suppose que le programme à transformer contienne suffisamment de constructions non-déterministes pour que les choix faits lors de l'exécution permettent de dévier de manière significative de l'exécution par défaut. Cette approche a l'avantage de garantir que les exécutions des programmes transformés sont des exécutions potentielles des programmes originaux. Cependant, si le programme ne contient pas suffisamment de choix, les possibilités de personnalisation sont beaucoup plus restreintes et la méthode perd une grande partie de son intérêt. Dans cette section, nous introduisons une nouvelle transformation permettant d'introduire de nouveaux choix en se basant sur des actions interchangeables, puis nous étudions comment caractériser de telles actions.

4.3.1 Transformation d'actions

Il est possible d'ajouter de nouveaux choix, de nouvelles possibilités, dans un programme, à condition que ces nouveaux choix n'empêchent pas le fonctionnement du programme. Nous devons veiller à éviter de définir une transformation qui permettrait de transformer le programme de manière quelconque, n'offrant ainsi plus aucune garantie sur l'exécution résultante.

Jusque là, nous avons étudié comment transformer la structure des programmes ; nous nous intéressons maintenant à un élément atomique du programme : l'action primitive. Réduire le champ

d'études aux seules actions permet en outre de limiter les risques d'introduire des transformations pouvant casser les programmes. Nous souhaitons introduire les variantes d'une action pour permettre de choisir parmi ces variantes.

Intuitivement, un agent a mille et une manières d'exécuter une même tâche élémentaire, qui sont toutes des variations d'une même action. Un agent qui doit franchir une porte d'une pièce à l'autre peut le faire discrètement, en ouvrant la porte le moins possible et en se faufilant dans l'entrebâillement ; il peut le faire poliment, en toquant à la porte avant d'entrer ; il peut le faire violemment, en la faisant claquer ; il peut le faire fièrement, ouvrant grand la porte et s'annonçant aux occupants de la pièce. Un agent gérant une application web qui envoie un article à un utilisateur peut le faire en gardant la mise en page originale de l'article, en le reformatant avec une typographie optimisée pour la lecture sur écran ou une mise en page plus appropriée à la taille de l'écran d'un téléphone portable. Dans tous les cas, la même action est effectuée, avec à des effets à des préconditions similaires. Nous cherchons à introduire ce type de choix entre les variantes d'une même action.

Une transformation qui remplacerait une action par un choix entre ses variantes exige de regrouper ensemble les actions similaires. Nous appelons *famille d'actions* un ensemble d'actions exprimant la même opération exécutée de manières différentes. La définition d'une famille d'actions est délicate, car un mauvais regroupement permettrait d'introduire à des endroits du programme des actions n'ayant rien à y faire et pouvant nuire à son bon fonctionnement. Nous considérons donc initialement que la définition des familles d'actions est effectuée par le concepteur d'application conjointement à la définition d'une théorie de l'action. Nous étudions ensuite dans la section 4.3.2 les caractéristiques des familles d'actions dans le but d'aider à leur création.

Une famille d'actions regroupe des actions effectuant la même opération. Si une action fait partie de deux familles d'actions, cela signifie que ces deux familles d'actions sont toutes constituées d'actions effectuant la même opération, et qu'elles pourraient être rassemblées pour ne former qu'une seule famille d'action. De fait, nous introduisons la contrainte qu'une action ne peut faire partie que d'une seule famille d'action.

La définition des familles d'actions suppose l'existence d'une théorie de l'action décrivant les actions en question. Cette définition se fait sous la forme d'un prédicat *family*.

Définition 8. *family* est un prédicat défini par le concepteur de l'application. La formule $family(F)$ est vraie si l'ensemble F est une famille d'actions, c'est-à-dire un ensemble d'actions effectuant la même opération.

De plus, une action ne peut faire partie que d'une seule famille d'action :

$$\forall a, F_1, F_2 \ (family(F_1) \wedge family(F_2) \wedge a \in F_1 \implies a \notin F_2). \tag{4.7}$$

Exemple

Prenons un exemple similaire à celui de la section 4.2.2 : un agent devant parcourir un donjon pour trouver des trésors (la définition complète est donnée en annexe A.1). Ses actions contiennent du pourfendage de monstres, de la récupération de trésors, mais surtout de l'ouverture de porte. De base, c'est l'action *openDoorWithKey* qui est utilisée pour ouvrir les portes. Cette action présuppose [3] que la porte soit fermée ($\neg open(d, s)$) et que l'agent possède la clé de la porte ($inInventory(key(d), s)$), et a pour effet de l'ouvrir ($open(d, s)$). On introduit deux variantes ►

Exemple (cont)

de cette action. L'action *breakDoorOpen* ne requiert pas de clé et exige uniquement que la porte soit fermée ($\neg open(d,s)$), et casse la porte (de manière à ce quelle ne puisse plus être fermée, par exemple) en plus de l'ouvrir ($open(d,s) \wedge broken(d,s)$). L'action *lockPickDoor* crochète la porte, ce qui ne requiert pas une clé, mais un rossignol ($\neg open(d,s) \wedge inInventory(lockPick,s)$). Ces trois actions sont regroupées dans une famille avec le prédicat *family* :

$$family(\{openDoorWithKey, breakDoorOpen, lockPickDoor\}).$$

Nous introduisons une transformation permettant de remplacer une action par un choix entre toutes les actions d'une famille dont elle fait partie.

Transformation 6 (T6)**.** Si $F = \{a_1, \ldots, a_n\}$ est une famille d'action,

$$\forall a_i \in F \qquad a_i \rightarrow a_1 \mid \ldots \mid a_n.$$

Cette transformation T6 est conçue pour être utilisée avec les transformations T1 et T3 pour contrôler plus finement les actions que l'agent va essayer. La transformation T3 permet de retirer certaines actions du résultat, de manière à ne garder que celles qui sont intéressantes pour un agent donné. La transformation T1 permet d'ordonner les autres de manière à forcer l'ordre d'évaluation des actions, pour introduire une préférence sur la manière d'exécuter l'opération en question.

Exemple

Pour reprendre l'exemple précédent, le remplacement de

$$openDoorWithKey$$

par

$$openDoorWithKey\rangle\, lockpickDoor\rangle\, breakDoorOpen$$

permet de définir un agent pragmatique et compétent, qui ouvrira la porte s'il a la clé (il est pragmatique, c'est le choix le plus simple), qui la crochètera sinon (car il est compétent) et qui la défoncera en dernier recours. Le replacement par

$$breakDoorOpen\rangle\, openDoorWithKey$$

définit un agent brutal qui préfère défoncer les portes dès qu'il en a l'occasion, et qui est incapable de les crocheter.

4.3.2 Caractérisation des familles d'actions

Nous n'avons pour l'instant considéré que des familles d'actions définies manuellement par le concepteur d'application. Cette définition est à la charge du concepteur d'application et s'ajoute

3. Nous listons dans cet exemple les préconditions et les effets des actions sans ordre, mais dans une véritable théorie de l'action, les préconditions se trouveraient dans la définition de *Poss* et les effets dans les axiomes de l'état successeur pour chaque fluent.

aux autres définitions nécessaires pour l'altération d'un programme ; dans la mesure du possible, il serait préférable de l'automatiser, ou du moins d'aider à sa réalisation. De plus, rien n'empêche de définir des familles d'actions risquant de modifier le fonctionnement d'un programme, voire de rendre un programme non fonctionnel : il serait également utile de fournir des contraintes ou des guides pour éviter cela. Nous nous intéressons à l'étude de propriétés qui permettent de définir, de manière automatique ou semi-automatique, des familles d'actions qui ne présentent pas de risque, afin de guider le concepteur d'application.

Pour être exacts, nous cherchons comment définir une famille d'action de manière à ce qu'un programme transformé par T6 soit une restriction (voir définition 7, page 68) du programme original. Nous introduisons la notion de famille sûre d'actions, qui offre un moyen pour un concepteur d'application de s'assurer qu'une famille d'action ne peut mener à des transformations posant problème à l'exécution d'un programme.

Définition 9 (Famille sûre d'actions)**.** Une famille d'actions F est sûre vis-à-vis d'un programme δ si, pour toute paire d'actions $a_1 \in F, a_2 \in F, a_1 \neq a_2$, $\delta[a_1/a_2]$ est une restriction de δ.

$\delta[a_1/a_2]$ est obtenu par remplacement de chaque occurrence de a_1 par a_2 dans δ

Actions sans différences dans le formalisme

Les actions sont définies dans le calcul des situations par leurs préconditions et par leurs effets. Souvent, les actions du formalisme correspondent à des actions effectuées dans le monde réel, qui sont pilotées par l'agent (commandes d'un robot, requêtes d'un agent sur le web). Les actions du calcul des situations ne sont qu'une modélisation des actions réelles, et ne prennent en compte que ce qui est important pour l'agent : il est tout à fait raisonnable d'ignorer l'échauffement de l'air produit par une commande à un moteur. De fait, il est possible de vouloir distinguer deux actions réelles différentes, mais qui sont modélisées de la même manière.

Il est donc possible d'avoir plusieurs actions ayant la même définition dans le formalisme, mais représentant des actions différentes hors formalisme. Il est évident que le remplacement d'une action par une autre action ayant la même définition n'a aucun effet sur le programme et est donc complètement sûr. Ce changement peut cependant être important quand on considère le programme non plus simplement dans le formalisme, mais également dans ses effets non formalisés.

Dès lors, une famille d'actions constituées d'actions ayant toutes les mêmes préconditions et les mêmes effets est une famille sûre d'actions.

Exemple

Un agent dont la mission est d'envoyer des articles à un utilisateur peut ne pas de soucier de la forme qu'ont les articles envoyés. Les actions *sendPlainTextArticle*, qui envoie l'article sans aucune modification de format, *sendFormattedArticle*, qui utilise une belle typographie, et *sendMobileArticle*, qui présente l'article de manière appropriée pour un téléphone mobile, ont alors toutes la même définition, et forment ensemble une famille sûre d'actions.

Théorème 4. *Une famille d'actions sans différences dans le formalisme est une famille d'actions sûre.*

Démonstration. Soit F une famille d'actions ayant toutes la même définition dans le calcul des situations. Alors pour toute paire a_1, a_2 d'actions de F, a_1 est une restriction de a_2 (lemme 3). Donc F est une famille sûre d'actions (théorème 2. □

Enrichissement du domaine

Nous avons cité dans la section précédente l'exemple de théories de l'action où les effets secondaires de certaines actions ne sont pas représentés. Étudions maintenant ce qui se produit quand on cherche à rajouter des fluents à la théorie de l'action pour représenter ces détails qui ne sont initialement pas modélisés.

Considérons une théorie de l'action $\mathscr{D}$ définissant un ensemble d'actions $\mathscr{A}$ et un ensemble de fluents $\mathscr{F}$. L'ajout d'un fluent f à $\mathscr{F}$ permet de définir un ensemble de variations des actions de $\mathscr{A}$ agissant différemment sur f sans rien changer aux autres fluents de $\mathscr{F}$. Nous nous limitons au cas où f est un fluent relationnel (pouvant être vrai ou faux), mais le raisonnement est le même pour des fluents fonctionnels.

Tout d'abord, chaque action peut avoir différents effets sur f. Pour chaque action $a \in \mathscr{A}$, il y a trois possibilités d'effets sur f : soit l'exécution de a passe f à vrai, soit l'exécution de a passe f à faux, soit l'exécution de a ne change pas f. Appelons a^+ l'action ayant la même définition que a, mais qui passe f à vrai, a^- celle qui passe f à faux, et a celle qui ne fait rien à f (qui a exactement la même définition de a).

Ensuite, f peut apparaître dans les prérequis de a de trois manières différentes : soit f est nécessaire pour exécuter a, soit $\neg f$ est nécessaire, soit f n'apparaît pas. L'ajout des variantes de a par rapport aux préconditions sur f donne trois actions que nous appelons respectivement a_+, a_- et a.

La combinaison des variantes sur les préconditions et les effets de a donne huit nouvelles actions qui viennent s'ajouter à a : a^+, a^-, a_+, a_+^+, a_-^+, a_-, a_+^-, a_-^-. Appelons $\mathscr{D}'$ la théorie de l'action issue de l'enrichissement de $\mathscr{D}$ avec le fluent f et les huit variantes de chaque action.

Dans ce cas, pour tout programme écrit avec les actions de $\mathscr{D}$, mais exécuté dans $\mathscr{D}'$, les préconditions et effets des actions sur f n'ont aucune importance sur l'exécution. Pour chaque action $a \in \mathscr{A}$, l'ensemble $\{a, a^+, a^-, a_+, a_+^+, a_-^+, a_-, a_+^-, a_-^-\}$ est une famille sûre d'actions.

Exemple

Pour reprendre l'exemple de l'envoi d'article, en partant de l'action *sendArticle*, on peut introduire le fluent *formatted* qui indique si l'affichage de l'article doit être adapté ou non. On peut introduire les actions *sendPlainTextArticle* et *sendFormattedArticle*, qui aient les mêmes préconditions et effets que *sendArticle*, à ceci près qu'elles passent le fluent *formatted* respectivement à faux et vrai. Dans ce cas {*sendArticle*, *sendPlainTextArticle*, *sendFormattedArticle*} est une famille sûre d'actions.

Cette méthode de construction de familles sûres d'actions permet à un concepteur d'application de prendre une application existante dans un certain domaine et d'ajouter de nouvelles actions et de nouveaux choix en enrichissant le domaine sans prendre le risque de perturber le fonctionnement de l'application. Le concepteur d'application introduit de nouveaux fluents, choisit parmi les actions construites par cette méthode celles qui ont du sens dans le cadre de l'application, et les nomme.

Nous n'avons pour l'instant traité que le cas où le fluent introduit a pour seul argument la situation. Il est également possible que le fluent ait un ou plusieurs autres arguments. Dès lors, les possibilités de définitions d'actions sont beaucoup plus grandes. Les nouvelles actions peuvent prendre une partie ou la totalité des arguments, ou au contraire ne pas avoir d'arguments et dépendre de la valeur du fluent pour une constante donnée ou utiliser des quantificateurs. Par exemple, à partir d'une action $a(x)$, on pourra dériver autant d'actions que précédemment en ajoutant $f(x)$ ou $\neg f(x)$ aux préconditions et aux effets. On pourra également, pour une action dépendant ou non de x, rajouter aux préconditions ou aux effets $f(X)$ ou sa négation, où X est remplacé par une valeur constante du domaine de X : le nombre de possibilités devient immensément grand. Il sera aussi possible de rajouter aux préconditions [4] une formule utilisant un quantificateur sur x, avec éventuellement une restriction sur x, de la forme $\forall x\, \phi(x) \implies f(x)$ ou $\exists x\, \phi(x) \wedge f(x)$. Le concepteur peut construire de nouvelles actions en utilisant les méthodes décrites ici et s'assurer que la famille d'actions ainsi construite ne présente pas de risque pour l'application.

Le raisonnement est le même pour les fluents fonctionnels. La principale différence est qu'il est possible d'écrire des formules complexes utilisant f dans les préconditions [5] d'une action, pour requérir une valeur de $f(s)$ parmi plusieurs, ou une condition sur $f(s)$. Par exemple, on pourra ajouter aux préconditions d'une action $f(s) = X \vee f(s) = Y$ ou bien $\phi(f(s))$.

4.3.3 Généralisation aux procédures

L'utilité de l'introduction des variations d'une même action ne se limite pas aux actions primitives. Les procédures permettent de définir des opérations complexes [6] qui sont composées de plusieurs actions primitives assemblées par des constructions Golog. Il est tout à fait envisageable de définir plusieurs procédures effectuant des variantes de la même opération et de vouloir remplacer l'une par l'autre dans en programme en fonction du profil. De fait, nous définissons la notion de famille de procédures, similaires à celle de famille d'actions, et introduisons la transformation associée.

Définition 10. Le prédicat *familyP* est un prédicat défini par le concepteur de l'application. La formule *familyP*(F) est vrai si F est une famille de procédures, c'est-à-dire un ensemble de procédures effectuant la même opération.

De plus, une procédure ne peut faire partie que d'une seule famille de procédures :

$$\forall P, F_1, F_2\ (\mathit{family}(F_1) \wedge \mathit{family}(F_2) \wedge P \in F_1 \implies P \notin F_2). \tag{4.8}$$

Transformation 7 (T7)**.** Si $F = \{P_1, \ldots, P_n\}$ est une famille de procédures,

$$\forall P_i \in F \qquad P_i \rightarrow P_1 \mid \ldots \mid P_n.$$

L'introduction de ce concept permet d'étendre la possibilité de transformation offerte par la famille d'actions à n'importe quel niveau du programme : la transformation n'est plus cantonnée au niveau le plus bas (les actions primitives). Il est possible de modéliser l'exemple précédent plus finement, en

4. Mais pas aux effets : une action ne peut que changer la valeur de fluents donnés, pas changer la valeur de vérité d'une formule quelconque. Un quantificateur existentiel dans les effets d'une action reviendrait à définir une action ayant des effets non-déterministes, ce qu'IndiGolog ne permet pas.

5. De même, pas dans les effets.

6. Les procédures sont également appelées «actions complexes» dans la publication introduisant Golog [LRL+97].

définissant *openDoorWithKey*, *breakDoorOpen* et *lockPickDoor* comme des procédures utilisant des actions plus précises plutôt que comme des actions primitives.

$$\begin{array}{l}\textbf{proc}\ openDoorWithKey\\ \quad take(key); move(key, lock); turn(key); putAway(key); push(door)\\ \textbf{endProc}\\ \textbf{proc}\ breakDoorOpen\\ \quad take(axe); \textbf{while}\ \neg destroyed(door)\ \textbf{do}\ strike(door, axe)\\ \textbf{endProc}\end{array} \tag{4.9}$$

4.3.4 Conclusion

Nous avons défini une nouvelle transformation permettant de remplacer une action par une autre effectuant la même opération. Cela offre la possibilité au concepteur d'application d'introduire des possibilités d'altération dans un programme qui n'en aurait pas assez.

Nous avons également étudié sous quelles conditions cette transformation offre la garantie de ne pas entraver le fonctionnement du programme et comment générer des familles d'actions offrant cette garantie.

Cette transformation s'applique aussi bien aux actions primitives du calcul des situations qu'aux procédures complexes écrites en Golog. Cela permet au concepteur d'application de remplacer n'importe quelle procédure d'un programme par une procédure similaire, mais différente, permettant ainsi d'introduire des possibilités de personnalisation ou de personnification n'importe où dans un programme.

En l'état, ces familles de procédures doivent être définies par le concepteur de l'application. Il serait possible d'étudier comment générer ces variantes d'une procédure : les différentes procédures obtenues par l'application de transformation sur une procédure pourraient être introduites comme des familles d'actions lors de la transformation du programme les contenant.

Chapitre 5

Affinités et critères de choix

Dans un programme Golog quelconque, il n'y a pas d'ordre parmi les exécutions possibles : toutes les exécutions valables sont égales, et l'interpréteur en choisit une arbitrairement. Il n'existe à la base aucun critère qui permette de distinguer deux exécutions ou de préférer une exécution à une autre. Pour permettre de faire cette distinction, nous introduisons la notion d'*affinité* d'une action, qui indique à quel point une action est désirable. L'affinité est définie relativement à un *profil* contenant des *attributs*. Nous introduisons également une comparaison de programme, basée sur l'affinité des actions, qui permet de déterminer si un programme est plus désirable qu'un autre programme vis-à-vis d'un profil (ou qu'une variante de programme est plus désirable que l'original).

Cette approche est inspirée de [BS11], qui met en relation des actions et des traits psychologiques dans le cadre d'agents BDI. Cette contribution a fait l'objet d'une communication [DPBS13].

5.1 Profils et attributs

Pour procéder à l'altération d'un programme, il est nécessaire de définir les facteurs de cette altération. La réponse dépend du type d'application, et surtout du type d'altération. Dans le cadre d'un système de recommandation d'articles de presse que l'on souhaite personnaliser, la personnalisation consistera à prendre en compte le fait qu'un utilisateur *n'aime pas le sport* ou *préfère lire des articles courts*, alors que dans le cadre d'un robot personnifié évoluant dans un bâtiment [1], on voudra personnifier l'agent en prenant le compte le fait qu'il soit *brutal*, *prudent* ou *roublard*. Nous réunissons tous ces facteurs sous la notion d'*attributs*, définis comme des termes clos de la logique des prédicats. Le domaine des attributs dépend du domaine de l'application, et est défini par le concepteur de l'application.

Définition 11 (attribut)**.** Soit un ensemble de symboles Σ d'arité quelconque, défini par le concepteur d'application.

Un attribut est un terme clos de Σ. L'ensemble des attributs est noté $\mathscr{A}$.

Intuitivement, un attribut est un terme auquel un agent associe une certaine valeur. Selon le domaine, cette valeur peut signifier «l'utilisateur avec lequel l'agent interagit aime ou n'aime pas cet attribut» ou encore «l'agent incarne ou n'incarne pas cet attribut». Les attributs correspondant

1. Annexe A.1.

aux exemples précédents seront *likesSubject*(*sport*) (l'utilisateur aime les articles traitant de sport), *shortArticles* (l'utilisateur préfère les articles courts), *brutal, cautious* et *outlaw* (l'agent doit montrer une personnalité brutale, prudente ou roublarde). Les attributs sont liés aux actions (voir section suivante) et permettent de déterminer quelles actions sont préférables pour un agent donné.

Le programme est altéré en fonction de l'attitude de l'agent vis-à-vis d'un attribut et donc des actions qui y sont liées. Les attitudes d'un agent vis-à-vis d'un ensemble d'attributs sont regroupées dans un *profil*. En nous inspirant de [BS11], nous définissons cinq valeurs possibles pour un attribut donné :

hostile l'agent tient à éviter à tout prix ce qui est lié à cet attribut ;

défavorable l'agent préfère éviter ce qui est lié à cet attribut ;

neutre l'agent n'a pas d'avis sur cet attribut, n'y est ni favorable ni défavorable ;

favorable l'agent préfère ce qui est lié à cet attribut ;

inconditionnel l'agent veut à tout prix ce qui est lié à cet attribut.

On positionne ces valeurs selon deux axes :

— polarité : l'agent est soit favorable à l'attribut, soit défavorable ;

— intensité : l'attitude vis-à-vis d'un attribut peut-être une simple préférence, ou une contrainte absolue.

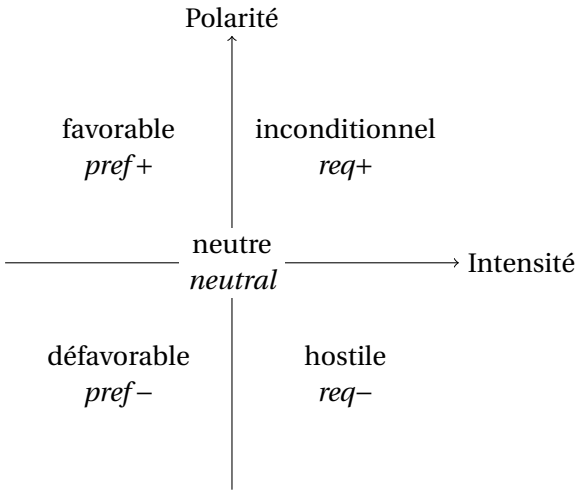

FIGURE 5.1 – Les valeurs possibles du profil peuvent se répartir selon deux axes.

De fait, ces cinq positions sont respectivement représentées par les symboles suivants : *req*−, *pref*−, *neutral*, *pref*+ et *req*+. Les deux positions extrêmes *req*− et *req*+ indiquent des avis forts qui peuvent aller au-delà de la rationalité, et nuire au bon fonctionnement de l'agent. De fait, on s'autorisera des transformations pouvant être destructrices en réponse à ces valeurs. Le concepteur d'application devra garder ceci en tête en définissant les profils.

Définition 12 (profil)**.** Un profil p est une fonction de l'ensemble des attributs $\mathscr{A}$ vers l'ensemble

$$\{req-, pref-, neutral, pref+, req+\}.$$

Le profil peut être fourni directement par le concepteur de l'application, ou calculé à partir d'une autre source. Dans le cas de la personnalisation, le profil de l'agent peut par exemple être calculé à partir d'un profil utilisateur recueilli précédemment sous la forme de couples clé-valeur.

La fonction de profil est définie en donnant sa valeur pour chacun des attributs existants. Un profil p dans un domaine ayant pour attributs a_1 et a_2 sera défini ainsi : $p(a_1) = pref+$, $p(a_2) = req-$. Cette formulation étant un peu longue, nous utiliserons l'abréviation suivante, avec le même sens : $p = \{a_1 : pref+, a_2 : req-\}$.

Cette classification des valeurs du profil selon deux axes est un choix que nous faisons. Ce choix offre la possibilité d'exprimer des contraintes plus fortes que la simple préférence dans le profil. Il serait possible de ne représenter ces valeurs que sur un axe, ce qui mènerait à un formalisme similaire offrant moins de possibilités pour le profil. De la même manière, nous avons choisi de l'utiliser que des valeurs discrètes pour simplifier les équations. Il serait tout à fait possible de choisir de représenter le profil par des valeurs continues (entre -1 et 1, par exemple), ce qui amènerait à un formalisme plus complexe, mais permettant plus de variation dans les profils.

5.2 Affinités d'actions

Pour altérer automatiquement le programme, il faut pouvoir déterminer si une action donnée est préférable à une autre action, et si un sous-programme donné est préférable à un autre sous-programme. Cela n'est pas possible en utilisant uniquement les informations fournies dans la théorie de l'action : soit une action a les préconditions et les effets qui permettent de l'utiliser pour atteindre le but, soit ce n'est pas le cas et l'action ne peut pas être utilisée. La théorie de l'action seule ne permet pas de choisir une action plutôt qu'une autre. Il est donc nécessaire d'apporter des informations supplémentaires, en dehors du calcul des situations, pour permettre ce choix. Pour ce faire, nous attachons des attributs aux actions.

Attacher des attributs aux actions plutôt que de donner directement des valeurs aux actions dans le profil offre une indirection qui permet de donner aux attributs des définitions complexes, dépendant de la situation, sans pour autant augmenter la complexité du profil. C'est aussi une séparation sémantique : les attributs font partie de la définition de l'action, il est donc intéressant de les faire apparaître en tant que tels, plutôt que de les cacher dans le profil.

Pour les raisons expliquées à la section 4.3.3, cette section porte autant sur les actions primitives (définies dans le calcul des situations) que sur les actions complexes (procédures définies en Golog).

5.2.1 Attributs d'une action

Nous souhaitons attacher des attributs aux actions. Le fait qu'une action possède un attribut conduit à ce qu'elle soit favorisée à l'exécution lorsque l'attribut a une valeur positive dans le profil, et défavorisée lorsqu'il a une valeur négative. Pour l'instant, nous pouvons exprimer qu'une action est liée, ou non, à un attribut ; nous souhaitons également pouvoir exprimer qu'une action est liée à l'opposé d'un attribut. Nous introduisons pour cela le symbole fonctionnel *neg*.

Exemple

Lors d'une mission discrète, le fait d'ouvrir une porte normalement n'est pas particulièrement discret, mais n'est pas tapageur non plus. Par contre, le fait de chanter très fort dans les couloirs, en plus de ne pas être discret, est particulièrement tapageur. Cette action aura l'attribut *neg*(*cautious*) et sera particulièrement évitée par un agent discret.

Définition 13 (*neg*)**.** *neg* est un symbole fonctionnel d'arité 1.

Définition 14 (attribut d'action)**.** Soit un ensemble de symboles Σ d'arité quelconque, défini par le concepteur d'application.

Un attribut d'action est un terme clos de $\Sigma \cup neg$. L'ensemble des attributs d'actions est noté $\mathscr{A}^*$.

Par la suite, on utilisera le terme attribut pour désigner des membres de $\mathscr{A}^*$ dans le contexte d'attributs liées à des actions, et pour désigner des membres de $\mathscr{A}$ dans le contexte d'attributs du profil.

5.2.2 Actions à attributs fixes

Dans le cas le plus simple, une action a un ensemble fixe d'attributs. Nous définissons une fonction *attributes*, dont la value devra être donnée par le concepteur de l'application.

Définition 15 (attributs d'une action)**.** *attributes* est une fonction qui fait correspondre à chaque action un ensemble d'attributs d'actions.

Pour chaque action a, $attributes(a)$ est l'ensemble des attributs attachés à a. Cette fonction est définie par le concepteur d'application.

Exemple

Pour l'agent-robot (décrit en annexe A.1), l'action de défoncer des portes est toujours brutale, et l'action de crocheter une porte en surveillant les environs est à la fois prudent et roublard :

$$attributes(bashDoorOpen(d)) = \{brutal, neg(cautious)\},$$

$$attributes(lockpickDoor(d)) = \{cautious, outlaw\}.$$

5.2.3 Actions à attributs variables

Les attributs d'une action peuvent également dépendre de la situation. On peut par exemple vouloir définir que crocheter une serrure n'est prudent que pendant la nuit. Il faut donc un mécanisme pour définir des attributs variables. Nous introduisons une seconde fonction qui donne, pour chaque action, une liste d'attributs accompagnés de conditions sous la forme de formules-fluents.

Dans le cas d'actions avec arguments, les attributs peuvent également dépendre des arguments : quand un verbe a un objet, la signification du groupe verbal dépend autant du verbe que de l'objet. Par exemple, la lecture d'un ouvrage scientifique complexe a l'attribut *difficile*, tandis que la lecture

d'un journal de presse a l'attribut *facile*. De fait, la formule gardant les attributs peut dépendre des arguments de l'action ainsi que de la situation.

Définition 16 (attributs variables d'une action)**.** La fonction *variableAttributes* prend en entrée une action $a(\vec{x})$ prenant un argument et renvoie une liste de paires $\langle \phi(\vec{x}, s), attribut \rangle$, où ϕ est une formule pouvant contenir comme variables libres les arguments de a et la situation s. Pour chaque paire, $a(\vec{x})$ possède l'attribut si $\phi(\vec{x}, s)$ est vrai. Cette fonction est fournie par le concepteur d'application.

$$\forall \vec{x}\ variableAttributes(a(\vec{x})) = \{\langle \phi_1, att_1 \rangle, \ldots, \langle \phi_n, att_n \rangle\}$$

Exemple

Pour l'exemple précédent, on définira *variableAttributes* tel que :

$$\forall a\ variableAttributes(read(a)) = \{\langle scientificArticle(a), hard \rangle, \langle pressArticle(a), easy \rangle\}.$$

Une considération est à garder en tête : nous travaillons sur l'analyse et la transformation structurelle des programmes. Quand nous parlons de la valeur d'une formule ϕ, il se s'agit pas de déterminer si ϕ est vrai dans une situation à l'exécution, mais si on peut déterminer par la lecture du programme Golog que ϕ est vraie en un point donné. Dès lors, les possibilités ne sont plus simplement ϕ et $\neg\phi$, mais ϕ, sa négation, ou l'impossibilité de déterminer ϕ ou sa négation à partir de l'analyse de la structure du programme.

L'interprétation est donc la suivante : une action ayant parmi ses attributs variables $\langle \phi, att \rangle$ a pour attribut *att* si on peut déduire ϕ de l'analyse du programme qui l'entoure (voir plus bas). Une question se pose : que faire quand l'analyse permet de déduire $\neg\phi$? Est-ce que l'action n'a simplement pas l'attribut, où faut-il un autre comportement ? Nous répondons à cette question par analogie avec les actions à attributs fixes. Lorsqu'un profil a une affinité négative vis-à-vis d'un attribut, cela signifie que les actions liées à cet attribut doivent être évitées. Dans le cas d'actions à attributs fixes, la transformation du programme préférera d'autres branches, ou fera disparaître l'action de manière à ce qu'elle ne soit pas exécutée. Dans le cas d'une action à attributs variables, la transformation va également éviter que l'action soit exécutée *dans le cas où ϕ est vraie*, ce qui passe par l'introduction de constructions s'assurant que ϕ est fausse. Il est donc nécessaire que $\neg\phi$ soit traité particulièrement. Dans la suite, nous considérerons que la négation de la formule ϕ induit une négation de la valeur de l'affinité liée à l'attribut : ne pas vouloir exécuter l'action pour ϕ, c'est vouloir l'exécuter pour $\neg\phi$.

5.2.4 Attributs d'une action dans un programme

De fait, les attributs de certaines actions dépendent de la situation. La situation n'est cependant présente qu'à l'exécution, et n'apparaît pas dans le programme lui-même, ce qui rend plus difficile le raisonnement sur les attributs en vue de transformer un programme. Pourtant, il est possible en regardant un programme de déterminer une partie de ce que sera la situation à l'exécution : dans le programme $\phi?; a$, on sait que quand a est exécuté, ϕ est vraie.

En IndiGolog, les tests et les conditions permettent de contraindre l'exécution en choisissant des branches où les tests sont vrais, et peuvent donc être utilisés lors d'une analyse statique comme des

garanties de la valeur d'une formule en un point du programme. Il est donc possible, lors de la descente récursive d'un programme, d'accumuler les formules des tests et conditions pour obtenir un ensemble de formules qu'on sait vraies en un point du programme.

Lors de l'analyse du programme, on assignera un attribut variable à une action si l'on peut déduire la formule correspondante de l'ensemble des formules vraies au niveau de l'action. On attribuera la négation de cet attribut (*neg*(*attribut*), qui amènera à une négation de la valeur du profil) si l'on peut déduire la négation de la formule correspondante, prouvant ainsi que l'action n'a jamais cet attribut. Les négations des valeurs du profil sont définies ainsi :

Définition 17 (Opposé d'une valeur du profil)**.** Nous introduisons une relation d'opposition *neg* sur les valeurs du profil.

$$\begin{aligned} neg(req-) &= req+ \\ neg(req+) &= req- \\ neg(pref-) &= pref+ \\ neg(pref+) &= pref- \\ neg(neutral) &= neutral \end{aligned}$$

Définition 18 (Attributs d'une action au sein d'un programme)**.** L'ensemble des attributs d'une action $a(\vec{x})$ connaissant un ensemble de formules $G = \{\psi_1, \ldots, \psi_n\}$ est donné par :

$$\begin{aligned} &attributesGiven(a(\vec{x}), G) = \\ &\qquad \{att_i \text{ pour } \langle \phi_i, att_i \rangle \in variableAttributes(a(\vec{x})) \text{ si } G \models \phi_i\} \\ &\qquad \cup \{neg(att_i) \text{ pour } \langle \phi_i, att_i \rangle \in variableAttributes(a(\vec{x})) \text{ si } G \models \neg\phi_i\} \end{aligned} \tag{5.1}$$

où $G \models \phi_i$ signifie que l'on peut déduire ϕ_i de la conjonction des formules de G :

$$\{\psi_1, \ldots, \psi_n\} \models \phi \equiv \psi_1 \wedge \ldots \wedge \psi_n \models \phi. \tag{5.2}$$

Exemple

Pour reprendre l'exemple des articles scientifiques qui sont difficiles à lire, on considère une action *read*(*a*) ayant des attributs variables et un programme choisissant un article scientifique et le lisant.

$$variableAttributes(read(a)) = \{\langle scientificArticle(a), hard\rangle, \langle pressArticle(a), easy\rangle\}.$$

$$\pi a.scientificArticle(a)?; read(a)$$

On peut déterminer qu'au niveau de l'action *read*(*a*), *scientificArticle*(*a*) et donc que l'action *read*(*a*) a pour attribut *hard*, ce qu'il est impossible de déterminer en évaluant le terme *read*(*a*) seul.

La construction de G se fait par accumulation lors de la descente du programme. Elle est définie récursivement sur les constructions Golog. On distingue deux cas : les constructions unaires, telles que δ^*, et les constructions binaires, telles que $\delta_1; \delta_2$. Pour les constructions unaires, nous introduisons

la forme G_{δ^*}, définie ainsi : considérer δ^* sachant G, amène à considérer δ sachant G_{δ^*}. Pour les constructions binaires, on distingue les ensembles de formules connus pour les parties gauches et droites : ainsi, si l'on considère $\delta_1;\delta_2$ sachant G, on sera amené à considérer δ_1 sachant $G^{left}_{\delta_1;\delta_2}$ et δ_2 sachant $G^{right}_{\delta_1;\delta_2}$.

L'accumulation de formules se fait pour les constructions suivantes : le test, le if, et le while. Pour les autres constructions, G est passé sans modifications.

Dans le cas du while, on considère que la formule du while est vraie dans le corps du while.

$$G_{\textbf{while } \phi \textbf{ do } \delta} \equiv G \cup \{\phi\} \tag{5.3}$$

Le if présente un choix entre deux branches. Dans le programme **if** ϕ **then** δ_1 **else** δ_2, on peut déterminer que si δ_1 est exécuté, alors ϕ y sera vrai, et que si δ_2 est exécuté, alors $\neg\phi$ est vraie. De fait, ϕ et $\neg\phi$ sont respectivement connus dans les parties gauches et droites.

$$G^{left}_{\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2} \equiv G \cup \{\phi\} \tag{5.4}$$

$$G^{right}_{\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2} \equiv G \cup \{\neg\phi\} \tag{5.5}$$

Le test apparaît dans le cadre d'une séquence. Si une séquence a pour partie gauche un test, alors la formule du test est connue dans la partie droite. S'il n'y a pas de test, la séquence propage les formules connues sans modifier.

$$G^{right}_{\phi?;\delta} \equiv G \cup \{\phi\} \tag{5.6}$$

$$G^{left}_{\delta_1;\delta_2} \equiv G \tag{5.7}$$

$$G^{right}_{\delta_1;\delta_2} \equiv G \tag{5.8}$$

Les autres éléments ne modifient pas les formules connues.

$$G^{left}_{\delta_1|\delta_2} \equiv G \tag{5.9}$$

$$G^{right}_{\delta_1|\delta_2} \equiv G \tag{5.10}$$

$$G_{\pi x.\delta} \equiv G \tag{5.11}$$

$$G_{\delta^*} \equiv G \tag{5.12}$$

Les hypothèses sur lesquelles cette analyse se fonde ne sont pas vérifiées lors d'une exécution concurrente. Par exemple, dans le programme

$$(\phi?; a) \;||\; \mathit{setPhiToFalse},$$

l'action *setPhiToFalse* (qui rend ϕ faux) peut être exécutée entre ϕ? et a, ce qui interdit toute déduction sur la valeur de ϕ au moment où a est exécuté. Il en va de même pour les autres constructions : la présence de concurrence invalide notre analyse statique du programme. En conséquence, nous n'effectuons cette analyse que sur des programmes ne présentant pas de concurrence, et nous ne définissons pas G pour les constructions concurrentes.

5.2.5 Calcul de l'affinité

Nous utilisons les attributs d'une action pour déterminer comment elle est perçue par un profil donné. Cela est représenté par un multi-ensembles de valeurs du profil que nous nommons *affinité*. L'affinité est l'ensemble des valeurs correspondant pour un profil à tous les attributs de l'action. L'utilisation d'un multi-ensemble permet la comparaison fine de deux actions en prenant en compte toute la complexité des positions vis-à-vis des attributs d'une action : une action ayant un attribut positif pour le profil et un attribut négatif pour le profil n'est pas équivalente à une action ayant un attribut neutre.

L'affinité est calculée en réunissant tous les attributs de l'action sachant un ensemble de formules, en recueillant toutes les valeurs de profil correspondantes et en les réunissant dans un multi-ensemble. La valeur *neutral* est ignorée, car elle signifie l'absence d'opinion vis-à-vis d'un attribut, et est donc équivalente à l'absence d'élément dans le multi-ensemble.

Définition 19 (Affinité d'une action)**.** L'affinité d'une action $a(\vec{x})$ sachant G vis-à-vis d'un profil p est définie par :

$$aff_p(a(\vec{x}), G) = \{p(attr) \text{ pour } attr \in attributes(a(\vec{x})) \cup variableAttributes(a(\vec{x}), G) \text{ si } attr \neq neutral\}.$$

La comparaison d'une action à une autre (pour déterminer laquelle des deux est préférable pour un profil donné) se fait alors par comparaison de l'affinité. La comparaison de ces multi-ensemble est définie dans la section suivante.

5.2.6 Classe d'affinité

Par la suite, il sera utile de considérer la valeur de l'affinité d'une action de manière absolue, et pas simplement en comparaison à un autre programme, de manière à pouvoir parler des actions positives (qui sont préférables à une action ayant une affinité neutre), et des actions négatives (qui sont préférables à une action ayant une affinité neutre). Il sera ainsi possible d'extraire d'un ensemble d'action celles qui sont de manière évidente favorisées ou défavorisées par un profil, ce qui sera utile pour manipuler les familles d'actions.

Définition 20 (Classes d'actions)**.** Les classes d'actions C_{req+}, C_{pref+}, C_{req-} et C_{pref+} sont des prédicats définis ainsi :

$$\begin{aligned}
C_{req+}(a, G) &\equiv aff(a, G) \geq \{req+\} \\
C_{pref+}(a, G) &\equiv aff(a, G) \geq \{pref+\} \wedge \neg aff(a, G) \geq \{req+\} \\
C_{req-}(a, G) &\equiv aff(a, G) \leq \{req-\} \\
C_{pref-}(a, G) &\equiv aff(a, G) \leq \{pref-\} \wedge \neg aff(a, G) \leq \{req-\}.
\end{aligned} \tag{5.13}$$

Les classes d'action C_+ et C_- sont définies ainsi, à partir des classes précédentes :

$$\begin{aligned}
C_+(a, G) &\equiv C_{req+}(a, G) \vee C_{pref+}(a, G) \\
C_-(a, G) &\equiv C_{req-}(a, G) \vee C_{pref-}(a, G).
\end{aligned} \tag{5.14}$$

Définition 21 (Affinité positive et négative)**.** Une action a, sachant un ensemble de formules G a une affinité positive si et seulement si $C_+(a, G)$. Elle a une affinité négative si et seulement si $C_-(a, G)$.

5.3 Comparaison de programmes

La transformation de programmes demande de pouvoir évaluer quel programme est le plus approprié pour un profil donné. Pour juger de cela, nous introduisons une comparaison de programmes, définie récursivement sur les termes d'IndiGolog. Cette comparaison définit un ordre partiel : elle permet parfois de déterminer qu'un programme est supérieur à un autre, mais peut également être incapable de prouver la supériorité d'un programme sur l'autre.

Comme expliqué à la section 5.2.4, la comparaison ne se fait pas simplement sur un programme, mais sur un programme sachant vraies des formules. La comparaison est donc définie en termes de couples programme-formules (δ, G) plutôt que simplement de programmes. L'ensemble des formules connues comme vraies à la racine d'un programme étant vide, on définit la comparaison de programmes simples ainsi [2] :

$$\frac{(\delta_1, \emptyset) \geq (\delta_2, \emptyset)}{\delta_1 \geq \delta_2}. \tag{5.15}$$

5.3.1 Comparaison de multi-ensembles

Nous avons vu précédemment qu'une action est ramenée à un multi-ensemble de valeurs représentant les positions d'un profil vis-à-vis des différents attributs. Pour permettre de comparer ces actions, nous introduisons un ordre de comparaison sur les multi-ensembles. Cet ordre de comparaison nous permettra aussi de comparer des ensembles d'actions en choix non-déterministe, ou en séquence.

Cette comparaison repose sur les idées suivantes.

— L'ensemble des éléments supérieurs est supérieur à l'ensemble des éléments inférieurs : si $A > B$ et $C > D$ alors $A \cup C > B \cup D$.

— Un ensemble est supérieur ou égal à un autre si chacun de ses éléments est supérieur ou égal à chaque élément de l'autre (fortement supérieur).

Prouver qu'un ensemble est supérieur à un autre consiste donc à partitionner chaque ensemble de manière à ce qu'à chaque sous-ensemble du premier corresponde un sous-ensemble du second, duquel il est fortement supérieur. Cette comparaison permet de prouver la supériorité dans les cas où tous les éléments de l'un sont supérieurs à tous les éléments de l'autre, mais également dans le cas où les ensembles sont proches, mais diffèrent de quelques éléments.

Les règles qui définissent cette comparaison sont :

$$\frac{M_1 \neq \emptyset \wedge M_2 \neq \emptyset \wedge \forall a \in M_1, b \in M_2 \; a \geq b}{M_1 \geq M_2}, \tag{5.16}$$

$$\frac{M_1 \geq M_2 \wedge M_3 \geq M_4}{M_1 \cup M_3 \geq M_2 \cup M_4}. \tag{5.17}$$

5.3.2 Comparaison d'actions

La comparaison d'action se fait par comparaison du multi-ensemble des valeurs de l'affinité. Pour permettre cette comparaison, nous introduisons un ordre sur les valeurs d'affinités :

$$req+ > pref+ > neutral > pref- > req-. \tag{5.18}$$

2. Nous utilisons la notation $\frac{a}{b}$ pour signifier $a \models b$ (si a, on peut déduire b).

Les valeurs positives sont préférables au neutre, qui est préférable aux valeurs négatives. Les valeurs «requis» sont plus fortes que les valeurs «préféré», et sont donc situées à l'extrémité dans la comparaison.

Le neutre est un cas particulier : nous l'avons retiré de l'affinité, puisque celui-ci indique une absence de position. Ainsi, il faut tenir compte du fait que l'absence de position est équivalente à *neutral* en considérant l'ensemble vide équivalent à $\{neutral\}$ pour la comparaison. Ceci se fait par les règles suivantes.

$$\frac{M \geq \{neutral\}}{M \geq \emptyset} \tag{5.19}$$

$$\frac{\{neutral\} \geq M}{\emptyset \geq M} \tag{5.20}$$

Selon la formule (5.16), les conditions de ces règles sont donc équivalentes respectivement à $M \neq \emptyset \wedge \forall a \in M\ a \geq neutral$ et $M \neq \emptyset \wedge \forall a \in M\ neutral \geq a$, c'est-à-dire que ces ensembles sont uniquement constitués $req+$ et $pref+$ (respectivement $req-$ et $pref-$).

La comparaison d'actions se fait donc par la règle suivante.

$$\frac{aff_p(a, G_a) \geq aff_p(b, G_b)}{(a, G_a) \geq (b, G_b)} \tag{5.21}$$

5.3.3 Résolution des conditions

Le if présente un choix entre deux branches. Dans le programme $\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2$, on peut déterminer que si δ_1 est exécuté, alors ϕ y sera vrai, et que si δ_2 est exécuté, alors $\neg\phi$ est vraie. Si ϕ ou $\neg\phi$ font partie des formules connues, alors il est possible de réduire le if à un de ses sous-programmes. S'il est impossible de savoir si ϕ est effectivement vrai, alors on considérera le if de la même manière qu'un choix non-déterministes.

$$\frac{G \models \phi \wedge (\delta_1, G^{left}_{\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2}) \geq (\delta', G')}{(\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2, G) \geq (\delta', G')} \tag{5.22}$$

$$\frac{G \models \neg\phi \wedge (\delta_2, G^{right}_{\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2}) \geq (\delta', G')}{(\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2, G) \geq (\delta', G')} \tag{5.23}$$

$$\frac{G \models \phi \wedge (\delta', G') \geq (\delta_1, G^{left}_{\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2})}{(\delta', G') \geq (\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2, G)} \tag{5.24}$$

$$\frac{G \models \neg\phi \wedge (\delta', G') \geq (\delta_2, G^{right}_{\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2})}{(\delta', G') \geq (\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2, G)} \tag{5.25}$$

5.3.4 Comparaison de séquences

L'opérateur de séquence (;) est un opérateur binaire. Cependant, plusieurs séquences sont souvent utilisées les unes à la suite des autres, et cet opérateur est intuitivement associatif $((a;b);c)$ est équivalent à $a;(b;c)$. Nous souhaitons donc considérer les chaînes de séquences dans leur intégralité.Pour cela, nous transformons la chaîne de séquences en multi-ensemble et définissons la comparaison en terme de comparaison de multi-ensemble.

$$\frac{(\delta_1 = \_;\_ \vee \delta_2 = \_;\_) \wedge bag_{seq}(\delta_1, G_1) \geq bag_{seq}(\delta_2, G_2)}{(\delta_1, G_1) \geq (\delta_2, G_2)} \tag{5.26}$$

La fonction bag_{seq} est la fonction qui convertit une séquence en multi-ensemble. La définition suivante dépend de l'associativité à gauche de l'opérateur de séquence, c'est-à-dire que $a;b;c$ soit parsé comme $(a;b);c$.

$$bag_{seq}(\delta, G) = \begin{cases} \text{si } \delta = \phi_1?;\phi_2? & \emptyset \\ \text{si } \delta = \phi?;\delta' & bag_{seq}(\delta', G^{right}_{\phi?;\delta'}) \\ \text{si } \delta = \delta'?;\phi? & bag_{seq}(\delta', G^{left}_{\phi?;\delta'}) \\ \text{si } \delta = \delta_1;\delta_2 & bag_{seq}(\delta_1, G^{left}_{\phi?;\delta'}) \cup bag_{seq}(\delta_2, G^{right}_{\phi?;\delta'}) \\ \text{sinon} & \{(\delta, G)\}. \end{cases} \tag{5.27}$$

5.3.5 Comparaison des choix

De la même manière que pour l'opérateur séquence, l'opérateur de choix non-déterministe est binaire et associatif. La comparaison des choix non-déterministes de programmes (|) se fait donc de la même manière que pour les séquences : en prenant ensemble tous les programmes du choix. La définition est plus simple, puisque les tests ne sont pas pris en compte : un test dans une branche de choix n'a pas d'influence sur les autres branches. Le if est pris en compte ici : **if** ϕ **then** δ_1 **else** δ_2 est considéré équivalent à $\phi?;\delta_1 \mid \neg\phi?;\delta_2$ [3].

$$\frac{(\delta_1 = \_ \mid \_ \vee \delta_2 = \_ \mid \_) \wedge bag_{choice}(\delta_1, G_1) \geq bag_{choice}(\delta_2, G_2)}{(\delta_1, G_1) \geq (\delta_2, G_2)} \tag{5.28}$$

$$bag_{choice}(\delta, G) = \begin{cases} \text{si } \delta = \delta_1 \mid \delta_2 & bag_{choice}(\delta_1, G^{left}_{\delta_1|\delta_2}) \cup bag_{choice}(\delta_2, G^{right}_{\delta_1|\delta_2}) \\ \text{si } \delta = \textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 & bag_{choice}(\delta_1, G^{left}_{\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2}) \\ & \cup bag_{choice}(\delta_2, G^{right}_{\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2}) \\ \text{sinon} & \{(\delta, G)\}. \end{cases} \tag{5.29}$$

5.3.6 Éléments transparents à la comparaison

Les autres constructions sont transparentes vis-à-vis de la comparaison : **while** ϕ **do** δ, $\pi x.\delta$ et δ^* sont équivalents à δ, moyennant une éventuellement accumulation à G.

$$\frac{(\delta, G_{\textbf{while } \phi \textbf{ do } \delta}) \geq (\delta', G')}{(\textbf{while } \phi \textbf{ do } \delta, G) \geq (\delta', G')} \tag{5.30}$$

$$\frac{(\delta, G) \geq (\delta', G'_{\textbf{while } \phi \textbf{ do } \delta'})}{(\delta, G) \geq (\textbf{while } \phi \textbf{ do } \delta', G')} \tag{5.31}$$

$$\frac{(\delta, G_{\pi x.\delta}) \geq (\delta', G')}{(\pi x.\delta, G) \geq (\delta', G')} \tag{5.32}$$

$$\frac{(\delta, G) \geq (\delta', G'_{\pi x.\delta'})}{(\delta, G) \geq (\pi x.\delta', G')} \tag{5.33}$$

$$\frac{(\delta, G_{\delta^*}) \geq (\delta', G')}{(\delta^*, G) \geq (\delta', G')} \tag{5.34}$$

3. Cette équivalence est vraie dans IndiGolog en l'absence de concurrence

$$\frac{(\delta, G) \geq (\delta', G'_{\delta^*})}{(\delta, G) \geq (\delta'^*, G')} \tag{5.35}$$

5.4 Programmes requis et programmes bannis

La comparaison de programmes n'est cependant pas suffisante pour exprimer les opinions fortes que sont *req*+ et *req*−. Ces deux valeurs de profils indiquent respectivement que l'agent veut à tout prix ce qui y est lié, et veut éviter à tout prix ce qui y est lié. Les comparaisons ne permettent pas de distinguer une simple préférence d'une nécessite.

Pour prendre en compte cette contrainte forte, nous introduisons deux prédicats : *req* et *ban*. Le but de ces prédicats est de permettre de savoir quand préférer une partie du programme à une autre. Par exemple, prenons l'agent-robot (décrit en annexe A.1) qui exécute le programme

$$bashDoorOpen(d) \mid openDoor(d),$$

sachant que l'action $bashDoorOpen(d)$ a pour attribut *brutal*. Nous souhaitons distinguer l'agent ayant une préférence pour la brutalité ($p(brutal) = pref+$), et l'agent refusant les solutions non brutales ($p(brutal) = req+$). Pour le premier, le programme transformé sera

$$bashDoorOpen(d) \rangle openDoor(d)$$

(défoncer les portes si possible, sinon les ouvrir). Pour le second, le programme transformé sera

$$bashDoorOpen(d)$$

(défoncer la porte, sans aucune autre possibilité).

Définition 22. $req(\delta, G)$ est vrai si l'agent cherche à tout prix à exécuter δ sachant G.

Définition 23. $ban(\delta, G)$ est vrai si l'agent cherche à tout prix à éviter d'exécuter δ sachant G.

Les définitions de *req* et *ban* sont similaires, et se rapprochent de la définition de la comparaison, à savoir récursivement sur les constructions Golog.

$$req(a, G) \equiv C_{req+}(a, G) \tag{5.36}$$

$$ban(a, G) \equiv C_{req-}(a, G) \tag{5.37}$$

Le reste des définitions étant identiques pour *req* et *ban*, nous les donnons en utilisant le symbole P avec, pour chaque formule, $P \in \{req, ban\}$ [4].

$$\begin{aligned}
P(\phi?, G) &\equiv false \\
P(\delta_1; \delta_2, G) &\equiv P(\delta_1, G^{left}_{\delta_1;\delta_2}) \vee P(\delta_2, G^{right}_{\delta_1;\delta_2}) \\
P(\delta_1 \mid \delta_2, G) &\equiv P(\delta_1, G^{left}_{\delta_1|\delta_2}) \wedge P(\delta_2, G^{right}_{\delta_1|\delta_2}) \\
P(\pi x.\delta, G) &\equiv P(\delta, G_{\pi x.\delta}) \\
P(\delta^*, G) &\equiv P(\delta, G_{\delta^*}) \\
P(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, G) &\equiv P(\delta_1, G^{left}_{\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2}) \wedge P(\delta_2, G^{right}_{\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2}) \\
P(\textbf{while } \phi \textbf{ do } \delta, G) &\equiv P(\delta, G_{\textbf{while } \phi \textbf{ do } \delta})
\end{aligned} \tag{5.38}$$

4. L'espace des valeurs de P étant trivial, l'utilisation de second ordre pour cette définition ne pose pas de problèmes.

req et *ban* étant des contraintes très fortes, il est nécessaire de limiter leur propagation : un opérateur de choix ne porte *req* et *ban* que si ses deux membres le portent. Sans cela, une seule action *req* amènerait à des transformations majeures du programme. Par exemple, dans le programme $((a \mid b); c) \mid d$, $req(a)$ engendrerait $req(a \mid b)$, $req((a \mid b); c)$ et $req(a; c)$, ce qui amènerait à transformer le programme en $a; c$, ce qui est beaucoup plus radical que ce que nous souhaitons.

5.5 Conclusion

Dans ce chapitre, nous avons défini les notions de profils et d'affinités, qui permettent de définir sur quels critères un programme agent peut être transformé, et une comparaison de programme qui découle des affinités et qui permet de déterminer si un programme est préférable à un autre vis-à-vis d'un profil donné. Nous avons choisi de définir les valeurs du profil sur deux axes : la polarité, qui est utilisée lors de la comparaison, et l'intensité, qui permet de différencier la préférence de la contrainte. Les prédicats *req* et *ban* découlent de ce second axe, et permettent de savoir si une branche de programme est simplement préférée vis-à-vis d'un programme, ou si elle doit être absolument choisie.

La comparaison, ainsi que les prédicats *req* et *ban*, se joignent aux transformations des chapitres précédents pour former PAGE framework, un cadre de manipulation de programmes Golog. Ce cadre est la base du processus automatique de transformation (PAGE process) présenté au chapitre suivant : c'est sur cette base que le processus s'appuie pour déterminer quelles transformations appliquer, parmi celles définies au chapitre 4.

Chapitre 6

Processus automatique de transformation

L'espace des programmes qu'il est possible d'obtenir par le biais des transformations définies au chapitre 4 pour un programme donné est potentiellement très large [1] et contient un grand nombre de programmes qui n'ont aucun intérêt pour un concepteur d'application. Dans ce chapitre, nous proposons une approche constructive pour choisir un programme de cet ensemble, sous la forme de PAGE process, un processus semi-automatique qui applique ces transformations selon certaines règles pour obtenir un programme intéressant. Ce processus utilise la notion d'affinité pour choisir quelles transformations appliquer, dans le but d'obtenir un programme convenant à un profil donné.

PAGE process est défini en terme de constructions de Golog, et est constitué de trois transformations, chacune étant spécifique à une construction Golog : les transformations d'actions en un choix entre les actions d'une même famille, les transformations de choix de programmes pour en favoriser un, et les transformations de choix d'arguments pour favoriser les arguments correspondant à certains critères.

Un programme Golog peut être vu comme une structure récursive, dans laquelle les programmes sont des constructions composées de sous-programmes. Un choix non-déterministe de programme est ainsi un nœud à deux fils. De ce point de vue, le processus agit de manière récursive en parcourant l'arbre de bas en haut : les sous-programmes sont traités avant la construction qui les englobe. Ainsi, pour la passe qui transforme les choix de programme appliquée à un programme $a \mid b$, on appliquera d'abord cette passe à a, puis à b, avant de l'appliquer au programme entier.

Formellement, on notera $T(\delta, \delta', G)$ pour dire que le résultat de la transformation du programme δ à un point du programme où G est connu est δ'. On utilisera de la même manière T_a pour désigner la transformation spécifique aux actions, $T_|$ pour la transformation spécifique aux choix de programmes, et T_π pour la transformation spécifique aux choix d'arguments. Les règles de parcours d'un programme pour la transformation dans le cas de constructions à transformer sont les suivantes :

1. Chaque transformation peut être appliquée en plusieurs points du programme, et peut également être appliquée sur les programmes qui sont le résultat d'autres transformations : selon le programme de départ, le nombre de programmes résultant peut être très grand ou même infini.

$$T(a,\delta,G) \equiv T_a(a,\delta,G), \tag{6.1}$$

$$T(\delta_1 \mid \delta_2,\delta,G) \equiv T(\delta_1,\delta_1',G^{left}_{\delta_1|\delta_2}) \wedge T(\delta_2,\delta_2',G^{right}_{\delta_1|\delta_2}) \wedge T_|(\delta_1' \mid \delta_2',\delta',G), \tag{6.2}$$

$$T(\pi x.\delta(x),\delta') \equiv T(\delta(x),\delta''(x),G_{\pi x.\delta(x)}) \wedge T_\pi(\pi x.\delta''(x),\delta',G). \tag{6.3}$$

Pour les autres cas, le programme est simplement parcouru pour trouver des constructions à transformer.

$$T(\phi?,\phi?,\_) \tag{6.4}$$

$$T(\delta_1;\delta_2,\delta_1';\delta_2') \equiv T(\delta_1,\delta_1',G^{left}_{\delta_1;\delta_2}) \wedge T(\delta_2,\delta_2',G^{right}_{\delta_1;\delta_2}) \tag{6.5}$$

$$T(\delta^*,\delta'^*) \equiv T(\delta,\delta',G_{\delta^*}) \tag{6.6}$$

$$\begin{aligned} &T(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, \textbf{if } \phi \textbf{ then } \delta_1' \textbf{ else } \delta_2') \equiv \\ &\quad T(\delta_1,\delta_1',G^{left}_{\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2}) \wedge T(\delta_2,\delta_2',G^{right}_{\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2}) \end{aligned} \tag{6.7}$$

$$T(\textbf{while } \phi \textbf{ do } \delta, \textbf{while } \phi \textbf{ do } \delta') \equiv T(\delta,\delta',G_{\textbf{while } \phi \textbf{ do } \delta}) \tag{6.8}$$

$$T(\delta_1 \mid\mid \delta_2,\delta_1' \mid\mid \delta_2') \equiv T(\delta_1,\delta_1',G^{left}_{\delta_1||\delta_2}) \wedge T(\delta_2,\delta_2',G^{right}_{\delta_1||\delta_2}) \tag{6.9}$$

$$T(\delta_1 \gg \delta_2,\delta_1' \gg \delta_2') \equiv T(\delta_1,\delta_1',G^{left}_{\delta_1 \gg \delta_2}) \wedge T(\delta_2,\delta_2',G^{right}_{\delta_1 \gg \delta_2}) \tag{6.10}$$

$$T(\delta^{||},\delta'^{||}) \equiv T(\delta,\delta',G_{\delta^{||}}) \tag{6.11}$$

$$T(\langle \phi \rightarrow \delta \rangle, \langle \phi \rightarrow \delta' \rangle) \equiv T(\delta,\delta',G_{\langle \phi \rightarrow \delta \rangle}) \tag{6.12}$$

$$T(\delta_1 \rangle \delta_2,\delta_1' \rangle \delta_2') \equiv T(\delta_1,\delta_1',G^{left}_{\delta_1 \rangle \delta_2}) \wedge T(\delta_2,\delta_2',G^{right}_{\delta_1 \rangle \delta_2}) \tag{6.13}$$

$$\tag{6.14}$$

Le processus de transformation utilise les familles d'actions et la comparaison de programmes basée sur l'affinité. De fait, en plus de la théorie de l'action nécessaire pour définir un programme Golog, le processus nécessite des définitions supplémentaires :

— *attributes*, la fonction qui donne la liste des attributs de chaque action (définition 15, page 80) ;
— *variablesAttributes*, la fonction qui permet de déterminer les attributs des actions prenant des arguments (définition 16, page 81) ;
— *family*, le prédicat permettant d'obtenir les familles d'actions (définition 8, page 70) ;
— p, le profil vis-à-vis duquel on souhaite transformer le programme (définition 12, page 78).

6.1 Actions

La première transformation consiste à remplacer chaque action faisant partie d'une famille d'actions par un choix entre les membres de cette famille ayant une affinité positive, permettant ainsi à l'agent de remplacer une action par une autre qui serait préférable, sans pour autant le bloquer si cette dernière n'était pas possible.

Pour toute action (a_0), cette transformation se déclenche s'il existe une famille F telle que $a_0 \in F$. De par la définition des familles d'actions (définition 8), nous savons qu'une action ne peut pas faire partie de plus d'une famille d'actions.

Cette transformation se base sur la transformation T6 pour faire apparaître les alternatives à l'action originale, sur la transformation T3 pour retirer les actions indésirables (ayant une affinité négative) et la transformation T1 est utilisée pour ordonner les actions, de manière à ce que les actions ayant la plus forte affinité soient préférées à celles ayant une utilité plus faible. Le résultat est une combinaison de choix préférentiels et de choix non-déterministes de la forme

$$(a_1 \mid a_2) \rangle (a_3 \mid a_4) \rangle (a_5 \mid a_6)$$

où a_1 et a_2 ont une affinité supérieure à a_3 et a_4, qui ont eux-mêmes une affinité supérieure à a_5 et a_6.

Dans les détails, nous avons choisi de ne conserver que les actions ayant une affinité positive, ainsi que l'action originale : nous n'introduisons pas d'actions ayant une affinité nulle. Nous ne retirons l'action originale que si elle est explicitement rejetée par une affinité négative. Le résultat de cette transformation pour une action $a_0 \in F$, sachant un ensemble de formules G, est donc :

$$siblings(F, C_{req+}) \rangle siblings(F, C_{pref+}) \rangle (a_0 \text{ si } \neg C_{-}(a_0, G)) \tag{6.15}$$

où *siblings* est un raccourci désignant un choix entre les actions de la famille ayant une certaine affinité.

$$siblings(F, C) = |_{a_i \in F | C(a_i, G)}\ a_i \tag{6.16}$$

6.2 Choix de programmes

La seconde transformation consiste à remplacer un choix non-déterministe de programme par un choix préférentiel de manière à orienter l'ordre dans lequel l'agent évalue les possibilités. Un choix arbitraire est alors remplacé par une préférence, c'est à dire la sélection d'une possibilité en se rabattant sur l'autre si la première n'est pas possible.

L'application de la transformation à un programme $\delta_1 \mid \delta_2$ se fait en calculant l'affinité de δ_1 et de δ_2, et en remplaçant le programme par un choix préférentiel avec la transformation T1, favorisant à l'exécution celui des deux qui doit être préféré. Ainsi, si l'on prouve que δ_1 est supérieur à δ_2, δ_1 sera évalué avant δ_2. Si la comparaison n'est pas possible, le programme est laissé inchangé.

Nous souhaitons également pouvoir rejeter complètement une option en la faisant disparaître du programme.

Si l'un des deux programmes est noté comme devant être exécuté (*req*) ou devant être ignoré (*ban*) et pas l'autre, alors la transformation T3 est utilisée pour ne garder qu'un des deux : un agent hostile à une possibilité ne la considérera pas même si c'est l'unique possibilité, et un agent inconditionnel vis-à-vis d'une possibilité refusera tout autre. Si une option est marquée *req* et pas l'autre, alors seule cette option sera conservée. De la même manière, si une option est marquée *ban* et pas l'autre, alors cette option est supprimée

Cette étape s'exprime ainsi :

$$T_{|}(\delta_1 \mid \delta_2, \delta', G) = \begin{cases} \delta_1 & \text{si } (req(\delta_1, G) \wedge \neg req(\delta_2, G)) \vee (ban(\delta_2, G) \wedge \neg ban(\delta_1, G)) \\ \delta_2 & \text{si } (req(\delta_2, G) \wedge \neg req(\delta_1, G)) \vee (ban(\delta_1, G) \wedge \neg ban(\delta_2, G)) \\ \delta_1 \rangle \delta_2 & \text{si } (\delta_1, G) > (\delta_2, G) \\ \delta_2 \rangle \delta_1 & \text{si } (\delta_2, G) > (\delta_1, G) \\ \delta_1 \mid \delta_2 & \text{sinon.} \end{cases} \tag{6.17}$$

6.3 Choix d'arguments

Cette transformation agit sur les choix non-déterministes d'arguments. La fonction *variableAttributes* définit des paires prédicat-attribut. Le principe est de déterminer les prédicats liés aux attributs favorables et de les introduire dans le programme avec les transformations T2 et T4 pour favoriser certaines valeurs d'arguments.

La transformation concerne un programme $\pi x.\delta(x)$, où x est une métavariable, c'est-à-dire une variable dont la valeur est une variable de Golog, et $\delta(x)$ est un programme x dans lequel x est une variable libre. Cette transformation suppose l'existence d'une définition de $variableAttributes(\delta(x))$.

La valeur de $variableAttributes(\delta(x))$ est un ensemble de paires prédicats attributs. La transformation consiste à introduire dans le programme un test $\phi?$ où ϕ est un de ces prédicats, sa négation, une conjonction de certains de ces prédicats et de leurs négations, ou rien ($\phi = true$, c'est à dire pas de transformation).

Exemple

Par exemple, pour une action $read(a)$ et une définition

$$variableAttributes(read(a)) = \{\langle scientificArticle(a), hard\rangle, \langle pressArticle(a), easy\rangle\},$$

les valeurs possibles pour ϕ sont :

$$true$$
$$scientificArticle(a)$$
$$\neg scientificArticle(a)$$
$$pressArticle(a)$$
$$\neg pressArticle(a)$$
$$scientificArticle(a) \wedge pressArticle(a)$$
$$\neg scientificArticle(a) \wedge pressArticle(a)$$
$$scientificArticle(a) \wedge \neg pressArticle(a)$$
$$\neg scientificArticle(a) \wedge \neg pressArticle(a).$$

Formellement, cette liste est obtenue en appliquant *conj* sur l'ensemble des parties de l'ensemble

predicates(*action*) ($\mathscr{P}$(*predicates*(*action*)) :

$$predicates(action) = \{p \text{ pour } \langle p, att\rangle \in variableAttributes(a)\}, \tag{6.18}$$

$$conj(\{p_1, \ldots, p_n\}) = p_1 \wedge \ldots \wedge p_n. \tag{6.19}$$

La procédure est la suivante : toutes les valeurs possibles pour ϕ sont évaluées, et la meilleure est conservée. Puisque la comparaison de programmes prend en compte *variableAttributes*, les prédicats liés à des attributs positifs dans le profil seront préférés aux autres.

Le meilleur est défini comme le maximum dans un ordre partiel, c'est à dire l'élément le plus grand :

$$\delta = max(D) \equiv \forall \delta' \in D\ \delta' \geq \delta \implies \delta' = \delta, \tag{6.20}$$

$$candidatePrograms(action) = \{\pi x.conj(e')?;\delta(x) \text{ pour } e' \in \mathscr{P}(predicates(action))\}, \tag{6.21}$$

$$M = \{e \text{ pour } \pi x.conj(e)?;\delta(x) = max(candidatePrograms(action))\}. \tag{6.22}$$

Ce maximum peut ne pas être unique. Si l'on se retrouve avec un ensemble de ϕ possible, correspondant à un ensemble d'ensembles de prédicats M, on considérera le sous ensemble M' de M obtenu en retirant les éléments de M en incluant d'autres (cette réduction permet d'éviter d'introduire des prédicats inutiles), et utilisera une disjonction des ϕ correspondants aux éléments de M' [2]. Par exemple, si le profil est neutre vis-à-vis de *pressArticle*(a) et positif vis-à-vis de *scientificArticle*(a), alors les résultats pour *scientificArticle*(a) et *scientificArticle*(a) $\wedge$ *pressArticle*(a) seront deux maxima, mais la réduction permet de ne considérer que *scientificArticle*(a).

$$M' = \{e \in M \text{ tel que } \forall e' \in M\ e' \subseteq e \implies e' = e\} \tag{6.23}$$

$$disj(\{p_1, \ldots, p_n\}) = p_1 \vee \ldots \vee p_n \tag{6.24}$$

$$\phi = disj(\{conj(e) \text{ pour } e \in M'\}) \tag{6.25}$$

Une fois le meilleur ϕ déterminé, un dernier test est effectué pour savoir si $req(\pi x.\phi?;\delta(x))$ est vrai ou si $ban(\pi x.\delta(x))$ est vrai. Si l'un des deux est vrai, la transformation T4 est utilisée, remplaçant le programme original par $\pi x.\phi?;\delta(x)$, sinon, c'est la transformation T2 qui est utilisée, remplaçant le programme par $\pi x.\phi?;\delta(x)\rangle\pi x.\delta(x)$. Si c'est le programme original qui est le meilleur, aucune transformation n'est effectuée.

$$T_|(\pi x.\delta(x), \delta', G) \equiv \delta' = \begin{cases} \pi x.\delta(x) & \text{si } \phi = true \\ \pi x.\phi?;\delta(x) & \text{si } req(\pi x.\phi?;\delta(x)) \wedge \neg req(\pi x.\delta(x)) \\ & \quad \vee (ban(\pi x.\delta(x)) \wedge \neg ban(\pi x.\phi?;\delta(x)) \\ \pi x.\phi?;\delta(x)\rangle\pi x.\delta(x) & \text{sinon.} \end{cases} \tag{6.26}$$

2. De nombreuses définitions sont possibles pour cette étape, et donnent différents résultats. Par exemple, on pourrait chercher l'argument satisfaisant le plus grand nombre de ϕ, ou chercher à satisfaire en priorité les ϕ correspondant à des valeurs positives du profil (choisir parmi les valeurs préférées d'abord, sinon choisir une valeur autre qui n'est pas défavorisée).

La disjonction des ϕ est un bon compromis qui donne des résultats convenables dans la plupart des cas et dont l'expression reste simple.

6.4 Conclusion

Ce processus permet la transformation de programme pour obtenir un ou des programmes appropriés pour un profil donné. Cette transformation peut s'appliquer de deux manières. Elle peut être appliquée automatiquement sur le programme tout entier. Elle peut également être appliquée semi-automatiquement sur une ou plusieurs parties du programme, ou sur certaines des procédures constituant le programme. Les parties sur lesquelles exécuter le processus sont choisies par le concepteur d'application qui a la connaissance du programme et peut prédire quelles parties peuvent bénéficier de la transformation. Cette approche a le double avantage de réduire le temps nécessaire à la transformation, puisqu'elle n'est exécutée que sur une partie du programme, et d'offrir au concepteur d'application un contrôle plus fin sur le résultat de la transformation.

Cette approche semi-automatique peut être agrémentée d'un outil graphique permettant de visualiser les différentes parties du programme et mettant en évidence celles qui sont sensibles à la transformation (les parties contenant des actions ayant des attributs). Le concepteur pourrait modifier le profil à l'aide de curseurs, avoir un aperçu en temps réel de l'effet sur le programme et s'assurer que le résultat correspond à ses attentes. Cet outil faciliterait également le choix des parties à transformer.

Troisième partie

Applications des transformations

Chapitre 7

Personnalisation

Dans ce chapitre, nous illustrons notre formalisme et notre processus de transformation par le biais de scénarios d'applications personnalisées.

7.1 Système de recommandation

Nous étudions d'abord un système de recommandation, c'est-à-dire un système dont le but est de recommander des objets aux utilisateurs.

7.1.1 Description

Ici, le système recommande des articles de presse. Les articles ont un titre, un contenu, et traitent chacun d'un sujet. Le domaine est modélisé par quatre prédicats : $article(a)$ signifie que a est un article, $related(a, a')$ signifie que a et a' sont proches (écrits par le même auteur, ou traitent d'un même évènement, par exemple), $topic(t)$ signifie que t est un sujet, et $dealsWith(a, t)$ signifie que l'article a traite du sujet t. Le contenu de l'article n'est pas représenté dans le formalisme puisque l'agent n'est pas responsable de la mise en forme, il est stocké dans une base de données accessible au composant qui prépare la page à afficher.

Nous supposons que les valeurs de ces prédicats sont données, c'est-à-dire qu'il existe déjà un ensemble d'articles traitant de sujets et ayant des liens entre eux. Ces articles ont par exemple pu être extraits des flux RSS de sites d'information, les sujets sont extraits à partir des mots-clefs des articles et le lien entre les articles est établi à partir d'une mesure de distance basée sur le texte des articles.

Pour illustrer ce programme, nous utiliserons le corpus suivant : les deux sujets sont *economy* et *sports* et chaque sujet a plusieurs articles. Pour simplifier la lecture des exemples, les articles proches (*related*) sont représentés par des symboles similaires.

— $topic(economy)$, $topic(sports)$;
— $article(euro1)$, $article(euro2)$, $article(euro3)$,
$article(bitcoin1)$, $article(bitcoin2)$, $article(bitcoin3)$,
$article(foot1)$, $article(foot2)$, $article(foot3)$,
$article(hand1)$, $article(hand2)$, $article(hand3)$;
— $dealsWith(euro1, economy)$, $dealsWith(euro2, economy)$,
$dealsWith(euro3, economy)$, $dealsWith(bitcoin1, economy)$,

dealsWith(*bitcoin2*, *economy*), *dealsWith*(*bitcoin3*, *economy*),
dealsWith(*foot1*, *sport*), *dealsWith*(*foot2*, *sport*),
dealsWith(*foot3*, *sport*), *dealsWith*(*hand1*, *sport*),
dealsWith(*hand2*, *sport*), *dealsWith*(*hand3*, *sport*) ;

— *related*(*euro1*, *euro2*), *related*(*euro1*, *euro3*), *related*(*euro2*, *euro3*),
related(*bitcoin1*, *bitcoin2*), *related*(*bitcoin1*, *bitcoin3*), *related*(*bitcoin2*, *bitcoin3*),
related(*foot1*, *foot2*), *related*(*foot1*, *foot3*), *related*(*foot2*, *foot3*),
related(*hand1*, *hand2*), *related*(*hand1*, *hand3*), *related*(*hand2*, *hand3*).

L'idée est d'adapter la recommandation d'article en fonction de l'utilisateur. Le système standard, qui n'a aucune information sur l'utilisateur, propose des articles indifféremment, de manière aléatoire.

L'agent est basé sur la modélisation d'applications web définie au chapitre 3.2. L'agent dispose d'une unique action d'envoi *display*(*article*) qui lui permet d'envoyer un article, et ne reçoit aucune information en retour. Une action de sensing lui permet cependant d'attendre que l'utilisateur ait fini la lecture. L'unique page de l'intermédiaire *displayArticlePage* met en forme et affiche l'article en question. La définition de l'intermédiaire est donc la suivante :

$$[\langle display(article), displayArticlePage, [wait]\rangle]. \tag{7.1}$$

En plus de l'action *display*(*article*) définie par l'intermédiaire, l'agent dispose d'une autre action, *markShown*(*article*), associée à un fluent *shown*(*article*) qui lui permet de noter quels articles ont déjà été montrés à l'utilisateur. Leurs définitions sont directes : *markShown*(*a*) est possible si *a* est un article qui n'a pas encore été montré, et *shown*(*a*) est vrai si *markShown*(*a*) a été exécuté par le passé.

$$Poss(markShown(a), s) \equiv article(a) \wedge \neg shown(a, s) \tag{7.2}$$

$$shown(a, do(action, s)) \equiv action = markShown(a) \vee shown(a, s) \tag{7.3}$$

Le programme agent lui-même choisit un article que l'utilisateur n'a pas encore vu, l'envoie à l'utilisateur, puis le note pour éviter de le renvoyer plus tard, et répète le tout ad libitum.

$$(\pi a.\neg shown(a)?; display(a); markShown(a); wait)^* \tag{7.4}$$

Voici par exemple une exécution possible de ce programme :

$$display(bitcoin2), markShown(bitcoin2), display(foot3), markShown(foot3).$$

Vu la simplicité du programme, nous écrirons ce résultat ainsi :

$$bitcoin2, foot3.$$

7.1.2 Personnalisation selon les sujets

La première forme d'adaptation que nous souhaitons introduire est une adaptation selon les sujets qu'un utilisateur aime ou n'aime pas. Nous souhaitons présenter en priorité des articles traitant de sujets qui intéressent l'utilisateur. Cela signifie que le profil de l'utilisateur doit contenir des sujets et la vision que l'utilisateur en a. Autrement dit, les attributs sont les sujets.

Ici, le profil est construit manuellement à partir d'informations demandées explicitement à l'utilisateur. Ce dernier devra classer les sujets disponibles en quatre catégories : «aime», «neutre», «préfère

éviter», et «à éviter absolument», qui correspondent aux valeurs *pref*+, *neutral*, *pref*− et *req*− pour le profil. La valeur *req*+ n'est pas proposée, car la choisir ferait immédiatement disparaître les articles traitant de tous les autres sujets.

Les attributs ayant été déterminés, il est nécessaire d'indiquer quels attributs sont rattachés aux actions du programme. Ici, il est évident que les attributs de $display(a)$ dépendent de a. On définit donc *variableAttributes* de la manière suivante [1], en indiquant que $display(a)$ a l'attribut s si a traite de s.

$$variableAttributes(display(a)) = [\forall s.subject(s) : \langle dealsWith(a,s), s\rangle] \tag{7.5}$$

Ainsi, si les sujets définis dans le domaine sont *sport* et *economy*, on aura :

$$variableAttributes(display(a)) = [\langle dealsWith(a,sport), sport\rangle, \langle dealsWith(a,economy), economy\rangle]. \tag{7.6}$$

On peut alors utiliser le processus automatique de transformation pour personnaliser le programme en fonction d'un profil. La seule possibilité de transformation ici est la transformation du choix d'arguments, qui est décrite dans la section 6.3. Détaillons la procédure pas à pas pour un profil $\{sport : req-, economy : neutral\}$.

La procédure consiste à choisir une condition ϕ à introduire pour contraindre le choix de la variable. Ce ϕ est un prédicat défini dans *variableAttributes*, sa négation, ou la conjonction de plusieurs d'entre eux. Ici, les candidats sont :

$$true$$
$$dealsWith(a,sports)(a)$$
$$\neg dealsWith(a,sports)(a)$$
$$dealsWith(a,economy)(a)$$
$$\neg dealsWith(a,economy)(a)$$
$$dealsWith(a,sports)(a) \wedge dealsWith(a,economy)(a)$$
$$\neg dealsWith(a,sports)(a) \wedge dealsWith(a,economy)(a)$$
$$dealsWith(a,sports)(a) \wedge \neg dealsWith(a,economy)(a)$$
$$\neg dealsWith(a,sports)(a) \wedge \neg dealsWith(a,economy)(a).$$

Tous les programmes de la forme

$$(\pi a.\phi?; \neg shown(a)?; display(a); markShown(a))^*$$

avec ϕ dans la liste précédente, sont comparés et seuls les maxima [2] sont retenus. Dans ce cas, il s'agit de :

$$\neg dealsWith(a,sports)(a)$$
$$\neg dealsWith(a,sports)(a) \wedge dealsWith(a,economy)(a)$$
$$\neg dealsWith(a,sports)(a) \wedge \neg dealsWith(a,economy)(a).$$

1. $[\forall x.\phi(x) : f(x)]$ désigne la liste constituée de tous les $f(x)$ pour chaque x tel que $\phi(x)$, où $f(x)$ est une formule incluant x. Pour l'implémentation, on utilisera le prédicat `findall` de Prolog.
2. Selon la comparaison définie à la section 5.3.

Parmi les ϕ maximaux, on ne garde que ceux qui n'incluent pas d'autres d'autres (par exemple, $\neg dealsWith(a, sports)(a) \wedge dealsWith(a, economy)(a)$ inclus $\neg dealsWith(a, sports)(a)$), dans ce cas :

$$\neg dealsWith(a, sports)(a).$$

Puisqu'il n'y a qu'un seul résultat, c'est lui qui est choisi pour la transformation.

Enfin, puisque l'on peut prouver [3]

$$ban((\pi a.\neg dealsWith(a, sports)(a)?; \neg shown(a)?; display(a); markShown(a))^*),$$

c'est la forme sans fallback qui est choisie, et le résultat est :

$$(\pi a.\neg dealsWith(a, sports)(a)?; \neg shown(a)?; display(a); markShown(a))^*.$$

Ce programme, en plus de choisir des articles qui n'ont pas encore été montrés, ne peut choisir que des articles ne traitant pas de sport. Par exemple, une exécution enverra les articles :

$$euro3, bitcoin1, bitcoin2, euro1.$$

Pour un autre profil, $\{sport : pref+, economy : pref-\}$, les ϕ maxima sont :

$$dealsWith(a, sports)(a)$$
$$\neg dealsWith(a, economy)(a)$$
$$dealsWith(a, sports)(a) \wedge \neg dealsWith(a, economy)(a),$$

que l'on ramène à

$$dealsWith(a, sports)(a)$$
$$\neg dealsWith(a, economy)(a).$$

Puisqu'il y a deux résultats, c'est la disjonction, $dealsWith(a, sports)(a) \vee \neg dealsWith(a, economy)(a)$, qui est utilisée. Enfin, on ne peut prouver ni *req* ni *ban*, le résultat est donc :

$$\begin{aligned}&(\pi a.\neg dealsWith(a, sports)(a) \vee \neg dealsWith(a, economy)(a).?; \neg shown(a)?; display(a); markShown(a)\\&\quad\rangle\\&\pi a.\neg shown(a)?; display(a); markShown(a))^*.\end{aligned}$$

Cet exemple illustre comment le processus permet l'introduction de personnalisation dans un programme qui n'en contient pas, et qui ne présente pas une architecture spécifique en vue de l'ajout de personnalisation. Dans la section suivante, nous donnons un autre exemple de personnalisation du même programme et nous montrons surtout comment ces deux ajouts interagissent quand ils sont ajoutés ensemble.

3. D'après la définition de *ban*, à la section 5.4

7.1.3 Renforcement et découverte

Nous souhaitons maintenant faire en sorte que l'article présenté dépende des articles que l'utilisateur a indiqué avoir aimés. Deux comportements sont envisageables : la présentation d'articles proches des articles que l'utilisateur a aimés, ou au contraire, la présentation d'articles qui ne sont pas proches de ceux que l'utilisateur a aimés, dans le but de faire découvrir de nouveaux contenus à l'utilisateur.

Avant tout, il est nécessaire de noter que le programme original ne dispose pas de la capacité de demander à l'utilisateur ce qu'il a pensé d'un article. Nous utilisons donc une version améliorée du programme original qui en est capable. Cela passe par une modification de l'intermédiaire : *displayArticlePage* demande à l'utilisateur s'il a aimé l'article, et une nouvelle action de réception *askLiked*(*article*) permet d'obtenir la réponse. L'action de réception *askLiked*(*article*) remplace la précédente (*wait*). Le programme est simplement modifié pour ajouter un appel à *askLiked*(*article*) :

$$(\pi a.\neg shown(a)?; display(a); markShown(a); askLiked(a))^*. \tag{7.7}$$

Bien que ce programme semble absurde, car il demande à l'utilisateur s'il a aimé les articles sans rien faire de cette information, cette modification est nécessaire, car il n'est pas possible d'introduire une fonctionnalité telle que «demander à l'utilisateur s'il a aimé un article» au cours du processus automatique de personnalisation. Puisque les réponses de l'utilisateur ne changent rien au comportement de l'agent, cet agent n'est pas personnalisé.

Pour le processus de personnalisation, nous ajoutons au profil un attribut *relatedToLiked*, qui sera rattaché à l'action *display*(*a*) quand *a* est proche d'au moins un article ayant été aimé par l'utilisateur :

$$variableAttributes(display(a)) = [\langle \exists a' \wedge liked(a') \wedge related(a, a'), relatedToLiked \rangle]. \tag{7.8}$$

Dès lors, l'application du processus de transformation avec un profil ayant une valeur positive pour *relatedToLiked* poussera le programme à proposer si possible des articles liés à ce que l'utilisateur a précédemment aimé, tandis qu'une valeur négative poussera à présenter des articles différents.

$$\begin{aligned}&(\pi a.\exists a' \wedge liked(a') \wedge related(a, a')?; \neg shown(a)?; display(a); markShown(a); askLiked(a)\\ &\quad\rangle\\ &\pi a.\neg shown(a)?; display(a); markShown(a); askLiked(a))^*\end{aligned} \tag{7.9}$$

$$\begin{aligned}&(\pi a.\neg\exists a' \wedge liked(a') \wedge related(a, a')?; \neg shown(a)?; display(a); markShown(a); askLiked(a)\\ &\quad\rangle\\ &\pi a.\neg shown(a)?; display(a); markShown(a); askLiked(a))^*\end{aligned} \tag{7.10}$$

7.1.4 Combinaison des transformations

Il est tout à fait possible d'appliquer les deux transformations en même temps. Il suffit de combiner les deux définitions de *variableAttributes* :

$$\begin{aligned}variableAttributes(display(a)) = [&\langle dealsWith(a, sport), sport\rangle,\\ &\langle dealsWith(a, economy), economy\rangle,\\ &\langle \exists a' \wedge liked(a') \wedge related(a, a'), relatedToLiked\rangle].\end{aligned} \tag{7.11}$$

La procédure est la même. Par exemple, pour un utilisateur intéresse par l'économie et préférant un comportement de renforcement $\{sport : neutral, economy : pref+, relatedToLike : pref+\}$, le programme résultant est :

$$\begin{aligned}
&(\pi a.(\neg\exists a' \wedge liked(a') \wedge related(a,a')) \vee dealsWith(a,economy)?;\\
&\quad \neg shown(a)?; display(a); markShown(a); askLiked(a)\\
&\rangle\\
&\pi a.\neg shown(a)?; display(a); markShown(a); askLiked(a))^*. \qquad (7.12)
\end{aligned}$$

Les articles remplissant au moins une condition du profil sont préférés. Le processus de transformation n'est qu'une possibilité, et il serait possible de définir des variantes du processus de transformation pour obtenir d'autres résultats, plus appropriés. Par exemple, en cas de conditions préférées multiples, on pourrait préférer l'article qui en satisfait le plus grand nombre.

7.2 Hypermédia adaptatif pédagogique

Dans cette section, nous présentons un scénario d'hypermédia adaptatif pédagogique, inspiré d'un scénario similaire [Jac06]. Dans ce scénario, l'application accompagne un utilisateur à travers l'apprentissage d'un cours.

7.2.1 Description

L'application permet à l'utilisateur de choisir l'ordre dans lequel il souhaite apprendre les différents éléments du cours, tout en fournissant des bornes pour que l'ordre de parcours du cours reste cohérent. Pour chaque partie du cours choisie, l'application présente à l'utilisateur des ressources qui peuvent être des définitions, des explications, des exemples, ou des exercices. Une fois que suffisamment de ressources ont été vues, ou qu'un exercice validant la partie a été réussi, l'application permet à l'utilisateur de choisir une autre partie.

Il se peut qu'un utilisateur survole une ressource plutôt que de la lire. Cela peut se produire si l'utilisateur devient impatient parce que le cours est trop simple, ou parce que la session est trop longue. L'application est capable de détecter quand cela se produit, et présente un exercice pour savoir si l'utilisateur a compris et s'impatiente, ou est simplement las. Si l'exercice est réussi, la partie est validée et l'application continue, sinon, l'application propose à l'utilisateur de faire une pause.

Pour permettre à l'utilisateur de parcourir le cours comme il le souhaite, le cours est constitué d'un graphe de concept ayant des relations de prérequis. Chaque concept comporte des ressources, qui sont les documents concrets à montrer à l'utilisateur. Le cours est composé par une équipe pédagogique sensibilisée à la composition de cours non linéaires.

7.2.2 Programme de base

La modélisation du domaine est la suivante. Le terme $concept(c)$ indique que c est un concept du cours. Le terme $prereq(c,c')$ indique que c est un prérequis de c', et doit être appris avant. Les termes $definition(r)$, $explaination(r)$, $example(r)$ et $exercice(r)$ indique que r est une ressource : respectivement une définition, une explication, un exemple ou un exercice. Le prédicat $about(r,c)$ indique que la ressource r concerne le concept c.

En dehors du modèle logique, chaque concept dispose d'un nom d'affichage et chaque ressource d'un contenu que l'intermédiaire peut afficher. La réalisation des exercices est prise en charge par l'intermédiaire qui signale simplement à l'agent si l'exercice a été réussi. L'application comporte quatre pages :

— Une page permettant d'afficher une liste de concept. L'agent envoie une liste de concepts avec des priorités, et l'intermédiaire renvoie le choix de l'utilisateur. Les priorités permettent de guider l'utilisateur tout en lui laissant un choix large, et amènent des affichages différents (mis en valeur, normal, estompé).
— Une page permet d'afficher une définition, une explication ou un exemple. L'intermédiaire informe l'agent si l'utilisateur a sauté la ressource plutôt que de la lire.
— Une page permet d'afficher un exercice. L'intermédiaire envoie le résultat à l'agent quand l'exercice est terminé.
— Une page suggérant à l'utilisateur de faire une pause. L'intermédiaire rend la main à l'agent une fois que la pause est terminée.

La définition de l'intermédiaire correspondante est la suivante :

$$\begin{aligned}&[\langle \mathit{showConcepts}(\mathit{list}), \mathit{conceptsPage}, [\mathit{askConcept}]\rangle,\\ &\langle \mathit{showResource}(r), \mathit{resourcePage}, [\mathit{askSkipped}]\rangle,\\ &\langle \mathit{showExercice}(r), \mathit{exercicePage}, [\mathit{askResult}]\rangle,\\ &\langle \mathit{suggestPause}, \mathit{suggestPausePage}, [\mathit{waitDuringPause}]\rangle].\end{aligned} \tag{7.13}$$

L'agent dispose d'un fluent *seen*(*resource*) permettant de savoir si l'utilisateur a déjà vu des ressources. L'action *markSeen*(*resource*) permet de modifier ce fluent. Les définitions sont triviales (similaires à *shown* et *markShown* du scénario précédent).

Le niveau de connaissance d'un concept est représenté par un nombre entre 0 et 1. Il augmente à chaque ressource lue et à chaque exercice réussi. Le fluent *level*(*concept*) contient ce niveau, et *incrLevel*(*concept*, *value*) l'incrémente. Le concept est considéré comme connu (*learned*(*concept*)) si le niveau de connaissance est 1.

$$\mathit{Poss}(\mathit{incrLevel}(\mathit{concept}, \mathit{value})) \equiv \mathit{concept}(\mathit{concept}) \wedge \mathit{number}(\mathit{value}) \tag{7.14}$$

$$\begin{aligned}\mathit{level}(\mathit{concept}, \mathit{do}(a, s)) = l \equiv\\ a = \mathit{incrLevel}(\mathit{concept}, \mathit{value})\\ \wedge\ l = \mathit{max}(\mathit{level}(\mathit{concept}, s) + \mathit{value}, 1)\\ \vee\ a \neq \mathit{incrLevel}(\mathit{concept}, \_) \wedge l = \mathit{level}(\mathit{concept}, s)\end{aligned} \tag{7.15}$$

$$\forall c\ \mathit{concept}(c) \implies \mathit{level}(c, s_0) = 0 \tag{7.16}$$

$$\mathit{learned}(\mathit{concept}, s) \equiv \mathit{level}(\mathit{concept}, s) = 1 \tag{7.17}$$

Pour construire la liste des concepts, l'agent dispose d'un fluent fonctionnel *concepts* qui a pour valeur une liste de paires concepts priorités. Les priorités possibles sont *high, medium* et *low*. L'agent peut manipuler ce fluent avec les actions *clearConcepts, addHigh*(*concept*), *addMedium*(*concept*) et *addLow*(*concept*). Leurs définitions sont les suivantes :

$$\mathit{Poss}(\mathit{clearConcepts}, s) \equiv \mathit{True}, \tag{7.18}$$

$$Poss(addHigh(c), s) \equiv Poss(addMedium(c), s) \equiv Poss(addLow(c), s)$$
$$\equiv concept(c) \wedge \neg\exists p\ (c, p) \in concepts(s), \tag{7.19}$$

$$\begin{aligned} concepts(do(a, s)) = v \equiv \\ v' = concepts(s) \wedge [\\ a = addHigh(c) \wedge v = v' + \langle c, high\rangle \\ \vee\ a = addMedium(c) \wedge v = v' + \langle c, medium\rangle \\ \vee\ a = addLow(c) \wedge v = v' + \langle c, low\rangle \\ \vee\ a \neq addHigh(\_) \wedge a \neq addMedium(\_) \wedge a \neq addLow(\_) \wedge v = v'. \end{aligned} \tag{7.20}$$

Le programme de l'agent est structuré en plusieurs procédures. Les principales procédures sont *makeConceptsList*, qui crée la liste de concepts à envoyer à l'utilisateur, et *handle*(*concept*) qui fait tout ce qui est nécessaire pour enseigner *concept* à l'utilisateur. Le programme principal est :

$$\begin{aligned} &clearConcepts; makeConceptsList; sendConcepts(concepts); \\ &askConcept; handle(result(askConcept)). \end{aligned} \tag{7.21}$$

La procédure *makeConceptsList* parcourt tous les concepts qui ne sont pas encore connus de l'utilisateur et les ajoute à la liste avec la priorité moyenne si tous les prérequis sont connus et la priorité basse si ce n'est pas le cas. Le fluent *ready*(*concept*) permet de savoir si tous les prérequis sont connus.

$$ready(c, s) \equiv \forall c'\ prereq(c', c) \implies learned(c', s) \tag{7.22}$$

```
proc makeConceptsList
  while ∃c ¬learned(c) ∧ ¬c ∈ concepts
    πc ¬learned(c)?;
    if ready(c)
      addMedium(c)
    else
      addLow(c)
    endIf
  endWhile
endProc
```
(7.23)

La procédure *handle*(*concept*) choisit des ressources et les affiche jusqu'à ce que le concept soit connu.

```
proc handle(concept)
  while ¬known(resource)
    do πconcept about(resource, concept) ∧ ¬seen(resource)?; present(resource)
endProc
```
(7.24)

La procédure *present*(*resource*) appelle simplement *presentExercice* ou *presentOther* en fonction du type de la ressource.

proc *present*(r)
  if *example*(r) **then** *presentExercice*(r) **else** *presentOther*(r) **endIf**
endProc (7.25)

La procédure *presentExercice* envoie un exercice à l'utilisateur. Si l'exercice est réussi, il est marqué comme *seen* pour ne plus être affiché, et le niveau d'apprentissage du concept est mis à jour.

proc *presentExercice*(e)
  showExercice(e); *askResult*;
  if *result*(*askResult*) **then** *markSeen*(e); *updateKnowledge*(e) **else** **endIf**
endProc (7.26)

La procédure *updateKnowledge*(*resource*) met à jour le niveau de connaissance pour le concept de *resource*.

proc *updateKnowledge*(r)
  πc *about*(r, c)?; *incrLevel*(c, 0.25)
endProc (7.27)

La procédure *presentOther*(*resource*) présente une ressource puis met à jour le niveau de connaissance. Si l'utilisateur n'a pas lu la ressource, alors un exercice est proposé pour déterminer si le concept est connu, ou si l'utilisateur est juste lassé.

proc *presentOther*(r)
  showResource(r); *askSkipped*;
  if *result*(*askSkipped*) **then**
    πc *about*(r, c)?; *boredomExercice*(c)
  else
    markSeen(r); *updateKnowledge*(r)
  endIf
endProc (7.28)

Enfin, la procédure *boredomExercice*(*concept*), déclenchée quand l'utilisateur saute une ressource sans la lire, propose un exercice : si l'exercice est réussi, le concept est considéré connu, sinon, on

propose à l'utilisateur de faire une pause.

proc *boredomExercice*(c)
  $\pi\, r$ *about*(r, c);
  showExercice(e); *askResult*;
  if *result*(*askResult*) **then**
    markSeen(e); *incrLevel*($c, 1$)
  else
    suggestPause; *waitDuringPause*
  endIf
endProc (7.29)

7.2.3 Personnalisation

Pour personnaliser cette application, nous prenons en compte les facteurs suivants.

1. Le but : l'utilisateur souhaite apprendre un concept et seuls les concepts nécessaires à ce but seront vus.
2. La vitesse d'apprentissage : certains utilisateurs ont besoin de passer beaucoup de temps sur chaque concept, d'autres moins.
3. La capacité d'abstraction : selon sa manière d'apprendre, un utilisateur peut préférer des définitions, des explications, ou des exemples.

But

Tout d'abord, nous souhaitons permettre à l'utilisateur de choisir un concept à apprendre : l'application mettra en avant les ressources permettant d'atteindre ce concept.

Nous définissons un prédicat $prereq^*$ permettant de savoir si un concept mène à un autre : il s'agit de la clôture transitive de *prereq*.

$$prereq^*(c, c') \equiv c = c' \vee \exists c''\ prereq(c, c'') \wedge prereq^*(c'', c') \tag{7.30}$$

Enfin, nous définissons les attributs de l'action *addMedium* de manière à ce qu'elle soit favorisée pour les concepts voulus. L'attribut du profil correspondant à un but est $goal(c)$.

$$variableAttributes(addMedium(c)) = \{\langle prereq^*(c, g), goal(g)\rangle \mid \forall g\ concept(g)\} \tag{7.31}$$

Un fois les attributs définis, nous pouvons appliquer le processus de transformations sur la procédure *makeConceptsList*. Par exemple, un utilisateur souhaitant apprendre l'algorithme de Dijkstra

aura pour profil {*goal*(*dijkstra*) : *req*+}. La procédure transformée sera :

proc *makeConceptsList*
  while $\exists c\ \neg learned(c) \wedge \neg c \in concepts$
    $\pi c\ \neg learned(c) \wedge \underline{prereq^*(c, dijkstra)}?;$
    if *ready*(*c*)
      addMedium(*c*)
    else
      addLow(*c*)
    endIf
  endWhile
endProc. (7.32)

Vitesse d'apprentissage

Certains utilisateurs ont besoin de passer plus de temps sur des concepts, d'autres moins. Dans le profil, la capacité d'apprendre rapidement est représentée par l'attribut *fastLearner* (et un besoin de plus de temps est représenté par son opposé, *neg*(*fastLearner*)).

Nous définissons des variantes de la procédure *updateKnowledge*(*r*) pour les différentes vitesses d'apprentissage : une personne apprenant rapidement ne verra que deux ressources par concept, une personne apprenant lentement en verra 6. Ces variantes sont réunies dans une famille, et chacune est associée à une valeur de *fastLearner*.

proc *fastUpdateKnowledge*(*r*)
  $\pi c\ about(r, c)?; incrLevel(c, 0.5)$
endProc (7.33)

proc *slowUpdateKnowledge*(*r*)
  $\pi c\ about(r, c)?; incrLevel(c, 0.17)$
endProc (7.34)

$$family(\{updateKnowledge, fastUpdateKnowledge, slowUpdateKnowledge\}) \tag{7.35}$$

$$attributes(slowUpdateKnowledge(r)) = \{neg(fastLearner)\} \tag{7.36}$$

$$attributes(fastUpdateKnowledge(r)) = \{fastLearner\} \tag{7.37}$$

Le processus de transformation se fait au sein des procédures *presentExercice* et *presentOther*. C'est l'étape de transformation des familles d'actions (section 6.1) qui opère ici. Les stéréotypes d'apprenants rapides, lents et normaux sont traduits respectivement par les valeurs de profil *pref*+, *pref*− et *neutral*.

Pour le profil neutre {*fastLearner* : *neutral*}, aucune action de la famille d'action n'a d'affinité positive : aucune transformation n'est opérée.

Pour l'apprenant rapide, *fastUpdateKnowledge* est préféré à *updateKnowledge* et la procédure *presentExercice* est transformée ainsi (il en va de même pour *presentOther*) :

proc *presentExercice*(*e*)
  showExercice(*e*); *askResult*;
  if *result*(*askResult*) **then**
    markSeen(*e*); *fastUpdateKnowledge*(*e*)⟩ *updateKnowledge*(*e*) **else endIf**
endProc. (7.38)

De la même manière, pour un apprenant lent, la procédure *slowUpdateKnowledge* est préférée à la procédure *updateKnowledge*, et la procédure *presentExercice* est transformée ainsi :

proc *presentExercice*(*e*)
  showExercice(*e*); *askResult*;
  if *result*(*askResult*) **then**
    markSeen(*e*); *slowUpdateKnowledge*(*e*)⟩ *updateKnowledge*(*e*) **else endIf**
endProc. (7.39)

Capacité d'abstraction

Selon sa capacité d'abstraction, un utilisateur préférera des définitions, des explications ou des exemples. Cela est traduit dans le profil par des valeurs *pref*+ et *pref*− pour les attributs *wantsDef*, *wantsExpl* et *wantsExam*. Ces attributs sont liés à la procédure *present*(*r*) en fonction de la valeur de *r* :

$$\begin{aligned} &\mathit{variableAttributes}(\mathit{present}(r)) = \{ \\ &\quad \langle \mathit{definition}(r), \mathit{wantsDef} \rangle, \\ &\quad \langle \mathit{explaination}(r), \mathit{wantsExpl} \rangle, \\ &\quad \langle \mathit{example}(r), \mathit{wantsExam} \rangle \}. \end{aligned} \tag{7.40}$$

La procédure *handle* sera modifiée en fonction du profil. Par exemple, pour un utilisateur préférant les explications et les exemples (profil {*wantsExpl* : *pref*+, *wantsExam* : *pref*+}), le processus de transformation générera la procédure suivante :

proc *handle*(*concept*)
  while ¬*known*(*resource*)
    do π *concept about*(*resource*, *concept*) ∧ ¬*seen*(*resource*)?;
      explaination(*resource*) ∨ *definition*(*resource*)?; *present*(*resource*)
  ⟩
    do π *concept about*(*resource*, *concept*) ∧ ¬*seen*(*resource*)?; *present*(*resource*)
endProc. (7.41)

7.3 Conclusion

Ce chapitre illustre l'utilisation de notre formalisme et de notre processus de transformation dans le cadre d'applications personnalisées. D'une part, notre formalisme agent d'application web permet l'expression de ces deux scénarios sous forme d'agent Golog. D'autre part, notre processus de transformation permet l'ajout de comportements personnalisés dans ces applications. Dans le cas de l'hypermédia adaptatif, les transformations permettent aussi bien d'introduire de l'adaptation de parcours (l'utilisateur parcourt les concepts différemment en fonction de son but) que de contenu (différentes ressources sont proposées).

Les systèmes de recommandation et les hypermédias adaptatifs sont toujours implémentés de manières totalement différentes. Ces deux exemples mettent en évidence le côté générique de notre approche : les mêmes techniques sont employées pour fournir de la personnalisation dans le cadre d'un système de recommandation et d'un hypermédia adaptatif.

Chapitre 8

Personnification

8.1 Agent conversationnel

Les agents virtuels intelligents sont des personnages animés capables de mener des conversations avec des interlocuteurs humains. Les concepteurs d'agents virtuels intelligents cherchent à améliorer ces interactions en dotant les agents de comportements humains, par exemple l'évitement du regard dans une conversation [AMG13] ou la capacité de participer à un dialogue afin d'obtenir des informations [MHvdB+12].

Dans cette section, nous présentons un scénario d'agent virtuel intelligent inspiré d'un cas d'utilisation réel [VdBBMH12]. Ce scénario nous permet d'illustre notre approche, et de montrer ses avantages en terme d'expressivité.

Le scénario en question a trait à la personnification d'un agent dans une conversation. Dans un jeu dédié à l'entraînement des compétences de communication, un joueur joue le rôle d'un agent immobilier qui doit convaincre un personnage non joueur (PNJ), joué par un agent virtuel, de visiter une maison. Lors d'une conversation, le joueur peut poser des questions pour découvrir les souhaits du PNJ et peut communiquer des informations, des interprétations et des opinions dans le but le convaincre que la maison répond à ses attentes, en mettant en valeur les qualités de la maison qui font écho aux souhaits de l'acheteur. Le PNJ peut également prendre l'initiative dans le dialogue et poser des questions pour obtenir les informations qu'il souhaite, donner des opinions, et prendre la décision de visiter la maison ou mettre fin à la conversation.

Le PNJ dispose également d'une personnalité. Les deux traits modélisés sont l'*extraversion* et l'*agréabilité* du modèle Big Five [CM92]. Ces traits sont modélisés à un niveau grossier, sans considérer les sous-traits qui composent ces traits. La personnalité a un grand nombre de conséquences. Par exemple, un personnage extraverti a tendance à donner des informations tandis qu'un personnage introverti a tendance à poser des questions. Un personnage extraverti exprime des souhaits et des opinions tandis qu'un personnage introverti préfère énoncer des faits. Un personnage agréable parlera de la maison ou de l'environnement plutôt de que de lui-même. Il cherchera à continuer les conversations alors qu'un personnage désagréable aura tendance à changer de sujet de conversation sans prévenir. Un personnage désagréable aura également tendance à rejeter les interprétations de ses interlocuteurs. [VdBBMH12] présente un grand nombre de conséquences de la personnalité dans ce scénario ; nous en avons sélectionné deux pour illustrer notre formalisme.

Dans [MHvdB+12], le PNJ est piloté par un agent BDI dont les croyances sont les connaissances

que le PNJ a sur la maison en question, dont les buts sont l'obtention d'informations et la prise de décision, et dont les plans sont des stratégies de conversation permettant d'atteindre les buts. La personnalité du PNJ est implémentée de manière simple : les comportements à adopter sont donnés sous la forme d'expressions conditionnelles dépendant du profil psychologique de l'agent (la table 8.1 illustre la manière dont cela est formalisé).

```
if (agreeable) then            /* Boolean
  if (extravert) then          /* Boolean
    if (probability .5 ) then  /* With equal chance:
      return: tell(wish)       /* Tell a wish
    else
      return: tell(opinion)    /* Tell an opinion
    end if
  else                         /* Act introvert:
    return: tell(fact)         /* Tell fact about the house
  end if
else                           /* Act non-agreeable:
  return: tell(fact)           /* Tell fact about itself
end if
```

TABLE 8.1 – Pseudocode exprimant la sélection du type d'information

Nous avons choisi d'illustrer notre formalisme avec cet exemple pour plusieurs raisons. D'une part, les publications présentant ce scénario contiennent des évaluations montrant l'utilité de la personnification de l'agent dans ce cas. D'autre part, un agent exprimé avec BDI peut être facilement traduit en un agent exprimé en Golog en gardant la même architecture [SL10].

Enfin, ce travail est un bon exemple de la complexité que notre approche cherche à résoudre : l'expression nécessaire pour prendre en compte deux traits sur une seule action (table 8.1) est complexe, et croît exponentiellement avec le nombre de traits à gérer. Notre approche permet de définir les interactions entre les traits [1] et les actions et fournit, pour un profil donné, une transformation du programme appropriée pour le profil, comme illustré par la figure 8.1.

8.2 Représentation du domaine

Une ontologie de prédicats regroupe les différents sujets pertinents pour cette conversation. Par exemple, un prédicat *KitchenSurface* représente la connaissance de la superficie de la cuisine. Ces prédicats sont organisés de façon hiérarchique, ce qui permet à l'agent de connaître les rapports entre les sujets de conversation. Par exemple, le prédicat *House* a pour sous-prédicat *Kitchen*, qui a lui-même pour sous-prédicat *KitchenSurface*, ce qui signifie que parler de la surface de la cuisine, c'est parler de la cuisine et de la maison, et que l'opinion de l'agent de la surface de la cuisine influe sur son opinion de la maison de manière générale.

1. Dans notre approche, nous les appelons attributs

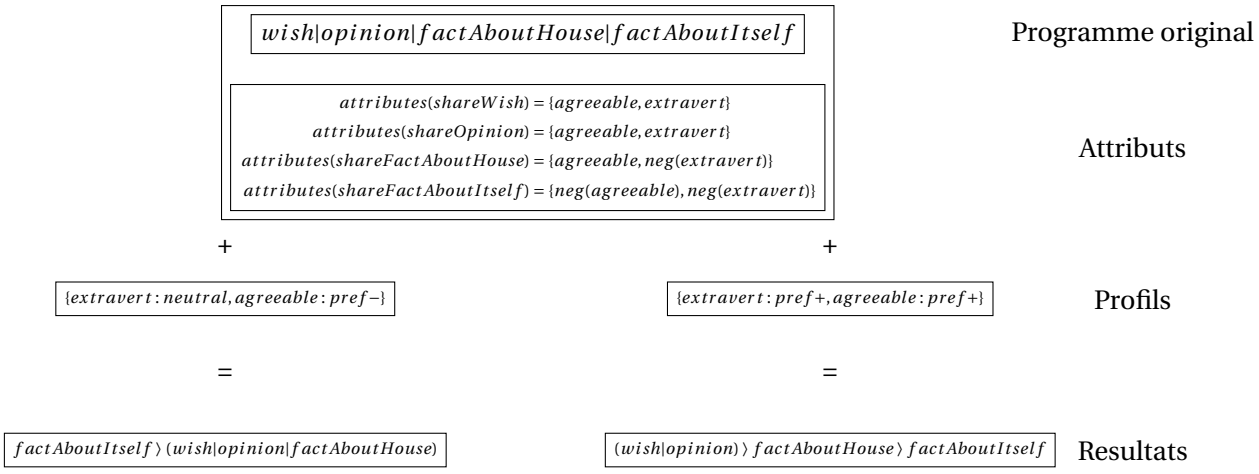

FIGURE 8.1 – Notre approche

Une ontologie d'éléments de conversation contient les différents types d'éléments qui peuvent être crus, transmis ou reçus par l'agent. Ces types d'éléments sont expliqués ci-dessous.

Fait Un fait indique une connaissance absolue de la vérité d'un prédicat. Par exemple, le fait que la superficie de la cuisine est de 12 mètres carrés s'exprime ainsi :

$$Fact(KitchenSurface, 12). \tag{8.1}$$

Interprétation Une interprétation indique une valeur subjective pour un prédicat. Par exemple, l'interprétation selon laquelle la superficie de la cuisine est grande s'exprime ainsi :

$$Interpretation(KitchenSurface, large). \tag{8.2}$$

Lorsque le vendeur transmet une interprétation à l'acheteur, celui-ci peut l'accepter ou la refuser.

Opinion Une opinion représente la disposition vis-à-vis d'un certain prédicat. Elle est exprimée par un nombre entre 0 et 1, 0 signifiant «très négatif» et 1 «très positif». Par exemple, l'opinion que la cuisine satisfait à 85 pour cent les besoins de l'agent s'exprime ainsi :

$$Opinion(Kitchen, 0.85). \tag{8.3}$$

Souhait Un souhait indique la ou les valeurs que l'acheteur considère comme idéales pour une maison. Si un agent désire une cuisine ayant une superficie entre 11 et 20 mètres carrés, on écrira :

$$Wish(KitchenSurface, [11, 20]). \tag{8.4}$$

Enfin, une ontologie des messages formalise la communication. Les types de messages sont *Tell*, *Ask* et *Acknowledge*.

Un message *Tell* est utilisé pour transmettre une information (que ce soit un fait, une interprétation, une opinion ou un souhait). Par exemple, le message informant que la cuisine fait 12 mètres carrés s'exprime ainsi :

$$Tell(Fact(KitchenSurface, 12)). \tag{8.5}$$

Un message *Ask* est utilisé pour demander une information. Par exemple, le message demandant le souhait de l'acheteur concernant la superficie de cuisine s'exprime :

$$Ask(Wish(KitchenSurface)). \tag{8.6}$$

Un message *Acknowledge* est utilisé pour confirmer qu'une information a été reçue, en exprimant éventuellement une opinion. Par exemple, le message indiquant que l'acheteur a reçu l'interprétation du joueur, mais la rejette, s'exprime :

$$Acknowledge(interpretationReceivedNeg). \tag{8.7}$$

8.3 Architecture de l'agent

L'agent dispose d'une base de connaissances constituée de faits, d'interprétations, d'opinions et de souhaits. Cette base de connaissances est représentée en calcul des situations par un fluent *KB*. Le fluent $KB(f,s)$ est vrai si f est considéré par l'agent comme vrai dans la situation s. Par exemple, le fait que l'agent considère dans une situation s que la cuisine est grande est exprimé ainsi :

$$KB(Intepretation(KitchenSurface, large), s). \tag{8.8}$$

Une action *assert* permet d'ajouter un élément à la base de connaissances. Par exemple, l'action $assert(Intepretation(KitchenSurface, large))$ ajoute $Intepretation(KitchenSurface, large)$.

La base de connaissances est mise à jour continuellement au cours de la conversation. Un ensemble de règles permettent de dériver des interprétations à partir des faits et de déterminer l'opinion de l'acheteur vis-à-vis d'un prédicat donné. Ainsi, quand l'agent reçoit un nouveau fait, les interprétations et opinions correspondantes sont également mises à jour. Ces règles sont de la forme $\phi \models fact \rightarrow element$. Par exemple, le lien entre la superficie de la cuisine et son interprétation comme étant grande s'exprime ainsi :

$$\forall x\ 12 \leq x \leq 20 \models Fact(KitchenSurface, x) \rightarrow Intepretation(KitchenSurface, large). \tag{8.9}$$

Ces règles sont utilisées lors de la mise à jour de la base de connaissances. L'axiome de l'état successeur pour *KB* est le suivant :

$$\begin{aligned} KB(f, do(a,s)) \equiv\ & a = assert(f) \\ & \vee\ a = assert(f') \wedge f' \rightarrow f \\ & \vee\ KB(f,s) \wedge \neg invalidates(a,f) \end{aligned} \tag{8.10}$$

$$\begin{aligned} invalidates(a,f) \equiv\ & a = assert(f') \\ & \wedge\ f = Opinion(p,v) \\ & \wedge\ \exists v'\ v \neq v' \\ & \wedge\ (f' = Opinion(p,v') \\ & \quad \vee\ f' \rightarrow Opinion(p,v')). \end{aligned} \tag{8.11}$$

À l'action *add* s'ajoute trois autres actions. *sensePlayerMessage* récupère un élément de conversation envoyé par le joueur et le place dans le fluent *playerMessage*. Si le joueur a dit plusieurs choses

depuis la dernière exécution de *sensePlayerMessage*, c'est le message le plus ancien qui est utilisé (les messages sont placés dans une file). Si le joueur n'a rien dit, le fluent *playerMessage* vaudra la valeur spéciale *nothing*. L'agent peut également envoyer un message au joueur avec l'action *tell* et accuser réception avec l'action *acknowledge*.

L'agent peut se comporter de deux manières différentes : passivement ou activement. Quand il se comporte passivement, il répond aux questions du joueur et prend en compte les informations qui lui sont transmises. L'agent peut également prendre la parole spontanément pour poser des questions spécifiques sur la maison, exprimer son opinion ou donner des informations sur ses souhaits.

```
proc buyer
  (senseUserMessage; if userMessage ≠ nothing
     then reactiveBehavior
     else proactiveBehavior endIf)*
endProc                                                    (8.12)
```

```
proc reactiveBehavior
  π f
  if message = tell(f) then
  evaluate(message)
  else
    message = ask(f)?; answer(f)
  endIf
endProc                                                    (8.13)
```

evaluate est la procédure permettant à l'agent de décider s'il accepte ou non l'information transmise par l'utilisateur. L'agent accepte toujours un fait, mais peut rejeter une interprétation, par exemple si elle n'est pas compatible avec sa propre interprétation des faits. Il en va de même pour une opinion.

```
proc evaluate(message)
  π f; message = tell(f)?;
  if f = Fact(_) then
    assert(f); ackApprove(f)
  else
      ¬incompatible(f)?; assert(f); ackApprove(f)
    |
      ackReject(f)
  endIf
endProc                                                    (8.14)
```

Les procédures $ackApprove(f)$ et $ackReject(f)$ sont deux procédures simples dont les définitions sont $acknowledge(accept)$ et $acknowledge(reject)$.

Le fluent *incompatible* est vrai si f est incompatible, c'est à dire que c'est l'agent tire une interprétation ou une opinion différente de celles que le joueur donne à partir des faits qu'il connaît.

$$incompatible(f,s) \equiv f = e(p,v) \wedge \exists f', v'.v' \neq v \wedge KB(f',s) \wedge f' \rightarrow e(p,v') \tag{8.15}$$

La procédure *answer* permet à l'agent de répondre à une demande de souhait ou d'opinion du joueur. Elle est relativement simple : l'agent répond toujours aux questions.

proc *answer*(f)
  $\pi element, predicate, value$
  $f = element(predicate) \wedge KB(element(predicate, value)?;$
  $tell(element(predicate, value))$
endProc (8.16)

Quand il se comporte activement, l'agent peut choisir de poser des questions spécifiques sur la maison, ou partager une information.

proc *proactiveBehavior*
  askQuestion | *shareInformation*
endProc (8.17)

Quand il partage de l'information, il choisit un type d'information, puis une information de ce type qu'il n'a pas encore dévoilé. *shareWish*, *shareOpinion*, *shareFactAboutHouse* et *shareFactAboutItself* sont des procédures cherchant un fait du type en question qui n'a pas encore été énoncé et le partageant avec le joueur. Leurs implémentations sont laissées de côté, car elles sont longues et n'apportent rien à la présente démonstration.

proc *shareInformation*
  shareWish | *shareOpinion* | *shareFactAboutHouse* | *shareFactAboutItself*
endProc (8.18)

Quand l'agent pose une question, il choisit un prédicat pour lequel il n'a pas encore d'information et pose la question au joueur. Le joueur peut répondre par un fait, une interprétation ou une opinion. Dans le cas d'une interprétation ou d'une opinion, l'agent peut la rejeter, comme quand le joueur partage une information directement.

proc *askQuestion*
  $\pi p.\neg\exists e, v.KB(e(p,v))?;$
  $ask(p); senseUserMessage; evaluate(message)$
endProc (8.19)

8.4 Altération de l'agent

L'agent acheteur peut être personnifié de multiples manières. Nous allons en illustrer deux, et montrer que les deux altérations peuvent être utilisées ensemble sans problèmes.

8.4.1 Sélection du type d'information à partager

L'agent peut partager spontanément des informations avec le joueur quand il prend la parole. Le type d'information partagée dans ce cas peut dépendre de la personnalité de l'acheteur. Un personnage extraverti (*extravert*) aura tendance à exprimer des souhaits et des opinions, tandis qu'un personnage introverti aura tendance à exprimer des faits ou poser des questions à propos des faits. Un personnage agréable (*agreeable*) aura tendance à parler de la maison ou de l'environnement, tandis qu'un personnage désagréable aura tendance à parler de lui-même. Dans l'implémentation de [VdBBMH12], cette personnalisation est faite par une énumération des cas possibles (table 8.1).

Notre agent choisit quelle information partager dans la procédure *shareInformation* (equation (8.18)). L'agent non personnifié choisit arbitrairement entre *shareWish*, *shareOpinion*, *shareFactAboutHouse* et *shareFactAboutItself*. En exprimant les attributs auxquels sont associées ces actions, il est possible de transformer automatiquement la procédure en fonction d'un profil.

Nous exprimons le trait «introverti» comme la négation de «extraverti» ($neg(extravert)$). Ainsi, un agent introverti évitera les actions associées à l'extraversion, et vice versa. Nous faisons de même pour les traits «agréable» et «désagréable». Compte tenu de la description des traits *agreeable* et *extravert*, la définition de la fonction *attributes* est la suivante :

$$\begin{aligned} attributes(shareWish) &= \{agreeable, extravert\} \\ attributes(shareOpinion) &= \{agreeable, extravert\} \\ attributes(shareFactAboutHouse) &= \{agreeable, neg(extravert)\} \\ attributes(shareFactAboutItself) &= \{neg(agreeable), neg(extravert)\}. \end{aligned} \tag{8.20}$$

La définition de la fonction *attributes* permet l'utilisation du processus automatique de transformation défini au chapitre 6 pour transformer la procédure *shareInformation* en fonction d'un profil. Les résultats de la transformation [2] pour toutes les valeurs de *agreeable* et *extravert* parmi *pref*−, *neutral* et *pref*+ sont donnés en table (8.4.1).

agreeable	*extravert*	Procédure transformé
pref−	*pref*−	*factAboutItSelf* ⟩ *factAboutHouse* ⟩ (*wish* | *opinion*)
pref−	*neutral*	*factAboutItSelf* ⟩ (*wish* | *opinion* | *factAboutHouse*)
pref−	*pref*+	(*wish* | *opinion* | *factAboutItSelf*) ⟩ *factAboutHouse*
neutral	*pref*−	(*factAboutHouse* | *factAboutItSelf*) ⟩ (*wish* | *opinion*)
neutral	*neutral*	*wish* | *opinion* | *factAboutHouse* | *factAboutItSelf*
neutral	*pref*+	(*wish* | *opinion*) ⟩ (*factAboutHouse* | *factAboutItSelf*)
pref+	*pref*−	*factAboutHouse* ⟩ (*wish* | *opinion* | *factAboutItSelf*)
pref+	*neutral*	(*wish* | *opinion* | *factAboutHouse*) ⟩ *factAboutItSelf*
pref+	*pref*+	(*wish* | *opinion*) ⟩ *factAboutHouse* ⟩ *factAboutItSelf*

TABLE 8.2 – Résultat de la transformation de la procédure *shareInformation* en fonction du profil.

Face à un choix entre plusieurs actions ou procédures, le processus cherche à les classer de manière à ce que les actions ou procédures favorisées par le profil soient préférées à l'exécution. Dans un premier temps, le processus détermine pour chaque procédure si elle appartient à une classe, en

2. Les noms des actions sont abrégés : le préfixe «share» a été retiré

évaluant ses attributs par rapport au profil. Par exemple, pour le profil {*agreeable* : *pref* +, *extravert* : *pref* +}, les procédures *shareWish* et *shareOpinion* ont pour attributs *agreeable* et *extravert* ; elles ont donc pour valeur {*pref* +} et sont de classe C_{pref+}. De la même manière, *shareFactAboutHouse* a pour valeur {*pref* +, *pref* −} et n'a pas de classe, et *shareFactAboutItself* a pour valeur {*pref* −} et est de classe C_{pref-}. Le processus construit alors le programme résultant avec des choix préférentiels entre les différentes classes, des plus favorables aux moins favorables ($C_{req+} > C_{pref+} >$ pas de classe $> C_{pref-} > C_{req-}$). Le programme résultant pour ce profil est donc

$$(\mathit{shareWish} \mid \mathit{shareOpinion}) \rangle \mathit{shareFactAboutHouse} \rangle \mathit{shareFactAboutItself}.$$

Notre approche permet d'obtenir les variations de programmes, et donc de comportement, pour différents profils en exprimant simplement les attributs des différentes actions du programme. Cela permet de définir des comportements complexes sans avoir à spécifier exhaustivement le comportement pour chaque combinaison de trait.

8.4.2 Acceptation des interprétations

Quand le joueur donne des informations à l'agent, l'agent peut choisir de les accepter ou de les refuser. Les faits sont toujours acceptés, car l'agent considère que le joueur ne ment jamais. Les opinions et interprétations peuvent par contre être refusées si elles sont en contradiction avec ce que pense l'agent (si l'agent a une interprétation différente des mêmes faits). Ce comportement est régi par la procédure *evaluate* (8.14).

Pour personnifier l'agent, on peut modifier ce comportement. Ainsi, un agent extraverti aura tendance à présenter implicitement son opinion en réponse à une information («Super !»), tandis qu'un agent introverti se contentera de confirmer qu'il a compris («D'accord.»). De plus, un agent désagréable aura tendance à ne même pas fournir de réponse. Enfin, un agent agréable aura plutôt tendance à accepter les opinions des autres qu'un agent désagréable.

Dans la procédure existante, les réponses possibles pour l'agent sont *ackApprove* et *ackReject*, qui communiquent l'acceptation ou le rejet de l'information de manière neutre. Pour permettre à l'agent d'exprimer autre chose, nous introduisons de nouvelles procédures et définissons deux familles d'actions, correspondant aux différentes manières de communiquer l'approbation et la désapprobation.

Tout d'abord, une procédure *ackApproveWithOpinion* qui permet à l'agent de confirmer qu'il accepte ce qui lui a été dit, tout en donnant son opinion sur cette information. Cette procédure n'est disponible que si l'agent a une opinion sur le sujet. Cette procédure est associée à un comportement

extraverti.

$$
\begin{array}{l}
\textbf{proc } ackApproveWithOpinion(f) \\
\quad \pi\, predicate, opinion \\
\quad f = \_(predicate, \_) \wedge KB(Opinion(predicate, opinion))? \\
\quad \textbf{if } opinion > 0.5 \textbf{ then} \\
\quad\quad acknowledge(accept\_happy) \\
\quad \textbf{else} \\
\quad\quad acknowledge(accept\_unhappy) \\
\quad \textbf{endIf} \\
\textbf{endProc}
\end{array} \tag{8.21}
$$

$$attributes(ackApproveWithOpinion) = \{extravert\} \tag{8.22}$$

Ensuite, une procédure *ackApproveSilently* qui permet à l'agent de ne pas communiquer son approbation. Concrètement, la procédure ne fait rien. Une autre procédure *ackRejectSilently* permet à l'agent de refuser silencieusement. Ces deux procédures sont distinctes, car elles font partie de familles différentes : l'une peut être utilisée pour l'approbation, l'autre pour la désapprobation. Toutes deux sont associées à un comportement désagréable.

$$\textbf{proc } ackApproveSilently(f) \quad nil \textbf{ endProc} \tag{8.23}$$

$$\textbf{proc } ackRejectSilently(f) \quad nil \textbf{ endProc} \tag{8.24}$$

$$attributes(ackApproveSilently) = \{neg(agreeable)\} \tag{8.25}$$

$$attributes(ackRejectSilently) = \{neg(agreeable)\} \tag{8.26}$$

Deux familles signalent quelles procédures sont interchangeables avec quelles autres :

$$family(\{ackApprove, ackApproveWithOpinion, ackApproveSilently\}), \tag{8.27}$$

$$family(\{ackReject, ackRejectSilently\}). \tag{8.28}$$

Enfin, l'action d'ajouter un message à la base est liée au trait «agréable».

$$attributs(assert(f)) = \{agreeable\} \tag{8.29}$$

Le processus de transformation utilise ces familles pour remplacer $ackApprov$ et *ackReject* par des alternatives plus appropriées au profil psychologique de l'agent. En outre, le choix non-déterministe entre l'approbation et la désapprobation pourra être remplacé par un choix préférentiel en fonction du profil.

Ainsi, pour un agent extraverti ($\{extravert : pref+, agreeable : neutral\}$), le résultat de la transformation est le suivant. Les différences sont soulignées. Pour chaque procédure faisant partie d'une famille de procédures (en l'occurrence, *ackApprove* et *ackReject*), chaque membre de la famille est évalué et classé à l'aide des règles définies à la section 5.2. Ainsi, *ackApproveWithOpinion* est de classe $pref+$, puisque son unique attribut est *extravert*, donc la valeur dans le profil est $pref+$. De même, *ackRejectSilently* et *ackApproveSilently* sont de classe $pref-$ et *ackApprove* et *ackReject* n'ont pas de

classe. Les procédures de classes négatives[3] sont écartées, et les procédures du programme sont remplacées par un choix entre les procédures restantes dans une famille, avec une préférence pour les procédures de plus grande classe : la procédure $ackApprove(f)$ est remplacée par la procédure $(ackApproveWithOpinion(f)\rangle ackApprove(f)$. C'est la seule transformation qui ait un effet dans cet exemple.

proc *evaluate*(*message*)
  πf; *message* = *tell*(f)?;
  if f = *Fact*(_) **then**
    assert(f); $\underline{(ackApproveWithOpinion(f)\rangle ackApprove(f))}$
  else
    $\neg incompatible(f)?; assert(f); \underline{(ackApproveWithOpinion(f)\rangle ackApprove(f))}$
  |
    ackReject(f)
  endIf
endProc (8.30)

Pour un agent désagréable ({*extravert* : *neutral*, *agreeable* : *pref* −}), le résultat est le suivant. En plus du remplacement des procédures par un choix entre les procédures de la famille, la transformation des choix non-déterministes a également un effet. Pour chaque choix non-déterministe «|», on compare les parties gauches et droites : si on peut prouver que l'une est plus grande que l'autre par les règles définies à la section 5.3, le choix est remplacé par un choix préférentiel «⟩» avec une préférence pour le plus grand des deux. Dans ce cas, on prouve que

$$ackRejectSilently(f)\rangle ackReject(f)$$

est préférable à

$$\neg incompatible(f)?; assert(f); (ackApproveSilently(f)\rangle ackApprove(f)).$$

proc *evaluate*(*message*)
  πf; *message* = *tell*(f)?;
  if f = *Fact*(_) **then**
    assert(f); $\underline{(ackApproveSilently(f)\rangle ackApprove(f))}$
  else
    $\underline{ackRejectSilently(f)\rangle ackReject(f)}$
  $\underline{\rangle}$
    $\underline{\neg incompatible(f)?; assert(f); (ackApproveSilently(f)\rangle ackApprove(f))}$
  endIf
endProc (8.31)

3. Une procédure de classe négative (C_-) est une procédure dont l'affinité est inférieure à {*pref* −} (voir section 5.2.6).

Enfin, pour un agent agréable et introverti ($\{extravert : pref-, agreeable : pref+\}$), aucune action des familles d'actions n'a de classe positive, donc seule la transformation du choix à un effet. Ici, on peut prouver que $\neg incompatible(f)?; assert(f); ackApprove(f)$ est supérieur à $ackReject(f)$.

```
proc evaluate(message)
  πf; message = tell(f)?;
  if f = Fact(_) then
    assert(f); ackApprove(f)
  else
      ¬incompatible(f)?; assert(f); ackApprove(f)
    ⟩
      ackReject(f)
  endIf
endProc
```
(8.32)

Ainsi, notre approche permet de définir des altérations complexes du comportement en définissant simplement les nouveaux comportements à incorporer et les attributs liés à ces nouveaux comportements et aux anciens. La procédure de transformation automatique utilise les attributs pour déterminer quand incorporer ces nouveaux comportements dans le programme, ce qui permet une altération complexe sans nécessiter une expression complexe.

8.4.3 Combinaison des transformations

Ces deux transformations touchant des points différents du programme, il est tout à fait possible de les appliquer simultanément pour obtenir un agent présentant les deux altérations de comportement. Avec le programme agent résultant, un agent extraverti va à la fois partager ses souhaits et opinions spontanément, et répondre aux informations qui lui sont données en donnant son avis dessus.

Des problèmes ne se posent que dans les cas où les altérations touchent des parties similaires du programme, auquel cas le résultat dépend du processus de transformation lui-même. Pour cette raison, nous considérons que le processus est semi-automatique : le concepteur de l'application peut vouloir valider le résultat de la transformation.

8.5 Conclusion

Dans ce chapitre, nous avons illustré l'utilisation de notre approche dans le cadre d'un scénario concret ayant été évalué. Nous avons montré que notre approche permet une expression plus simple et plus claire de la modification du comportement qu'une approche classique.

Un aspect du scénario est hors de portée de notre approche : [VdBBMH12] évoque la possibilité d'utiliser des valeurs réelles plutôt que booléennes pour modéliser les traits (par exemple, une extraversion de 0,45 plutôt que simplement 0 ou 1). Nous avons choisi dans cette étude de limiter les valeurs du profil à 5 valeurs discrètes (*req+*, *req−*, *pref+*, *pref−* et *neutral*). Cependant, cette limitation à des valeurs discrètes n'est pas une contrainte forte, et il est envisageable de mener la même étude

en choisissant de représenter le profil par des valeurs réelles, ce qui permettrait d'offrir ce genre de possibilités.

Quatrième partie

Conclusion

Chapitre 9

Conclusion

Dans cette thèse, nous avons proposé une approche constructive de l'altération d'agents intelligents. Cette approche repose sur l'introduction de nouveaux comportements dans un agent existant, par le biais de la manipulation du programme décrivant le comportement de l'agent. Cette approche est générique, et fonctionne pour différents types d'altérations : la personnalisation et la personnification. Nous appelons le résultat de cette approche des agents paramétriques (PAGE) : le comportement de l'agent résultant est paramétré par un profil.

Afin de pouvoir manipuler formellement la notion de personnalisation, nous avons défini les concepts de personnalisation forte et personnalisation faible, de manière à distinguer la simple idée de modifier le comportement en fonction de l'utilisateur, de la technique consistant à établir formellement un profil de l'utilisateur et à l'utiliser pour adapter le système.

Nous avons choisi le langage d'agents IndiGolog comme cadre pour notre étude, et avons proposé WAIG, un formalisme basé sur IndiGolog approprié à l'expression d'applications Web, pour étendre cette étude aux applications Web.

Nous avons proposé PAGE framework, un cadre formel pour la manipulation de programmes IndiGolog. Ce cadre comprend un ensemble de transformations permettant de modifier des programmes pour altérer le comportement d'agents, notamment pour infléchir les choix, tout en gardant la garantie du bon fonctionnement des programmes. Ce cadre comprend également une définition formelle de la notion d'adaptation à un profil et la définition d'une comparaison permettant de déterminer si un programme est plus adapté qu'un autre pour un profil donné.

Enfin, nous avons proposé PAGE process, un processus semi-automatique utilisant PAGE framework pour introduire de nouveaux comportements dans les programmes agents, plus adaptés à un profil donné. PAGE process demande au concepteur d'application de spécifier les rapports entre les éléments du programme et les éléments du profil, et utilise cette information pour choisir la variante du programme, au sens des transformations de PAGE framework, la plus appropriée pour un profil donné.

Cette approche et ce processus sont illustrés par plusieurs scénarios issus des domaines de la personnalisation et de la personnification, permettant de démontrer les capacités d'expression de l'approche, et sa pertinence dans le cadre d'applications réelles.

Chapitre 10

Perspectives

Dans le présent document, nous avons présenté une approche innovante : l'altération du comportement d'agents rationnels par l'application de transformations à leurs programmes. Cette approche est le point d'arrivée après de nombreux détours, et le temps aura manqué pour explorer complètement le champ des questions ouvertes par cette idée. Dans ce chapitre, nous parcourons les voies qu'ouvre le formalisme PAGE, nous revenons sur les choix qui ont été faits pendant la création du formalisme et nous examinons d'éventuels ajouts au formalisme lui-même.

10.1 Prolongements

10.1.1 Outillage

Lors de la création d'un agent personnalisé ou personnifié, ou de l'ajout de personnification ou de personnalisation, le concepteur d'application doit fournir un grand nombre d'informations : la théorie de l'action, le programme de l'agent, les attributs des actions, les profils, les familles d'actions, les procédures sur lesquelles appliques PAGE process, etc. La conception d'application est de fait une opération complexe qui demande bien souvent des allers-retours entre les différents composants de l'application.

Une grande partie des difficultés peuvent être résolues en assistant le concepteur d'application à l'aide d'outils appropriés. On peut ainsi envisager un outil d'aide à la conception d'application qui disposerait par exemple des capacités suivantes.

— L'outil informe le concepteur d'application des informations manquantes pour le fonctionnement ou pour la transformation de l'agent, affichant une feuille de route des éléments à définir.

— Quand le concepteur travaille sur la définition des entrées de PAGE process, l'outil affiche en temps réel le résultat des transformations sur le programme, permettant au concepteur de voir immédiatement l'effet de ses définitions et d'itérer rapidement pour arriver au résultat voulu.

— L'outil permet au concepteur de faire varier les valeurs du profil à l'aide de curseurs et de voir en temps réel le programme transformé et son exécution, de manière à vérifier rapidement que la transformation est conforme pour les différentes valeurs du profil.

— Dans PAGE framework, les attributs sont liés à des actions. L'outil utilise cette information pour mettre en valeur les parties du programme contenant des actions pouvant être affectées

par le profil, permettant ainsi au concepteur de concentrer son attention sur les parties du programme qui peuvent être transformées.

10.1.2 Expérimentation

Nous avons illustré l'utilité de PAGE en présentant son utilisation dans plusieurs scénarios, notamment des scénarios concrets présents dans la littérature et ayant fait l'objet d'expérimentations. Ces scénarios ont permis de montrer que PAGE est capable d'exprimer différentes formes d'applications et d'agents personnalisés ou personnifiés. Cependant, ces scénarios ne disent rien de la facilité d'utilisation de PAGE.

Pour montrer l'utilité réelle de PAGE en tant que formalisme pour l'expression d'agents et d'applications, il serait nécessaire d'effectuer une expérimentation mettant en jeu des concepteurs d'applications découvrant PAGE et devant l'utiliser pour créer un agent ou une application.

10.1.3 Preuve

Nous avons choisi d'ancrer nos travaux dans un formalisme logique pour plusieurs raisons, évoquées au cours de ce document. Parmi ces raisons, il en est une que nous n'avons pas explorée : la preuve. L'utilisation d'un formalisme logique permet d'effectuer des preuves automatiques sur les systèmes modélisés. Appliqué à PAGE, cela permettrait de prouver le fonctionnement des agents, et notamment de prouver l'agent peut s'exécuter quelle que soit la valeur du profil. Par exemple, on pourra prouver qu'un programme donné saura présenter des articles tous les utilisateurs, quels que soient leurs goûts, sans jamais manquer d'articles et sans qu'aucune combinaison de valeurs du profil soit problématique.

Combiné à un outil d'aide à la conception d'application, un système de preuve sur les programmes permettrait d'avertir le concepteur quand la transformation qu'il a définie casse le fonctionnement du programme pour certaines valeurs du profil.

10.2 Choix

Lors de la création de PAGE framework, nous avons été confrontés à de nombreux choix de conception. Certains de ces choix ont été faits après étude des possibilités et en connaissance de cause, tandis que d'autres ont été faits plus arbitrairement faute de recul sur les conséquences de ces choix sur les possibilités d'expression du cadre. Nous présentons ici les choix qu'il pourrait être intéressant de réévaluer.

10.2.1 Profil sur deux axes

Nous avons choisi de prendre les valeurs de profil parmi deux dimensions : l'une pour signifier la direction d'une préférence, et l'autre pour en indiquer l'intensité. La principale motivation de ce choix est la possibilité d'exprimer une contrainte forte, et donc de supprimer des morceaux de programmes pendant la transformation. D'autres solutions sont envisageables à ce problème, comme l'utilisation de plus de valeurs sur un même axe.

10.2.2 Profil discret

Si on laisse de côté la différence entre *pref* et *req*, le profil peut prendre trois valeurs : positive, négative, ou nulle. Ces valeurs permettent d'exprimer un grand nombre d'applications, mais se retrouvent insuffisantes quand un profil moins tranché est nécessaire. C'est par exemple le cas du personnage non joueur dans le jeu de vente (chapitre 8), chez qui le profil binaire conduit à un comportement caricatural.

Il serait intéressant d'étudier la possibilité d'un profil à valeurs réelles, entre -1 et 1 par exemple. Un tel choix permettrait l'expression d'une plus grande diversité de profils, ce qui serait utile pour créer des agents personnifiés plus uniques et moins stéréotypés.

Comment répercuter un tel profil sur les transformations ? La question reste ouverte. L'utilisation de logiques floues ou probabilités est une piste : elle pourrait permettre l'expression de valeurs réelles dans le profil sans pour autant invalider les raisonnements qui découlent de la représentation actuelle du profil.

10.3 Ajouts

Enfin, des possibilités d'ajouts au formalisme PAGE lui-même n'ont pas pu être explorées pleinement.

10.3.1 Introduction d'aléatoire dans le programme

Un agent transformé par la version actuelle de PAGE process se comporte toujours de la même manière. Si un choix non-déterministe a été transformé en choix préférentiel, alors l'une des options sera toujours préférée à l'autre.

Il serait intéressant d'apporter de la diversité aux exécutions d'un programme transformé sous la forme de hasard dans les choix. Ainsi, la préférence d'un choix sur un autre pourrait se traduire par une probabilité plus grande pour ce choix.

Prenons par exemple programme choisissant entre prendre le train et l'avion.

$$takeTrain \mid takePlane$$

Pour un profil préférant le train, le programme résultant choisit toujours le train tant que c'est possible.

$$takeTrain \rangle takePlane$$

Une transformation permettant d'introduire du hasard permettrait d'obtenir le programme suivant, exprimant que le train doit être préféré dans 75% des cas.

$$rollDice; \textbf{if } dice < 0.75 \textbf{ then } takeTrain \rangle takePlane \textbf{ else } takePlane \rangle takeTrain \textbf{ endIf}$$

Le sensing de IndiGolog pourrait être utilisé pour gérer l'aléatoire : le jet de dé peut être considéré comme le résultat d'une action de sensing.

10.3.2 Transformation à l'exécution

Tout au long de nos travaux, nous avons considéré que la transformation du programme se fait avant l'exécution. Il est cependant possible de faire autrement, et de transformer des morceaux du programme pendant l'exécution. Il est possible d'envisager une construction IndiGolog $PAGE(\delta, p)$ qui exécuterait le résultat de la transformation de δ pour le profil p, p étant un fluent fonctionnel pouvant être manipulé par l'agent. En supposant que le prédicat $T(\delta, \delta_p, p)$ signifie que δ_p est le résultat de la transformation de δ pour le profil p (prédicat qui n'est pas très loin de la définition actuelle de PAGE process), une première ébauche de la sémantique de l'opérateur $PAGE$ pourrait être :

$$Trans(PAGE(\delta, P), s, \delta', s') \equiv \exists \delta_p T(\delta, \delta_p, p) \wedge Trans(\delta_p, s, \delta', s'),$$

$$Final(PAGE(\delta, P), s) \equiv \exists \delta_p T(\delta, \delta_p, p) \wedge Final(\delta_p, s).$$

Une telle transformation à l'exécution permettrait notamment de transformer en fonction d'un profil dynamique, qui peut changer pendant l'exécution, par exemple en observant l'utilisateur pour adapter le profil, ou en altérant le comportement en fonction d'émotions capable de changer dynamiquement.

Cinquième partie

Annexes

Annexe A

Exemples

Ce chapitre regroupe et détaille les différents exemples utilisés au long de ce mémoire. Y sont décrits les scénarios, leur modélisation dans le calcul des situations, et un programme IndiGolog implémentant le comportement par défaut de l'agent.

A.1 Un agent-robot visitant une maison

Un robot visite une maison composée de plusieurs pièces. Chaque pièce peut être fermée à clef ou non et peut contenir des clefs vers d'autres pièces ou de l'or. Le robot peut porter autant d'objets qu'il souhaite et son but et de trouver et ramasser l'or.

Deux fluents sont utilisés pour représenter l'état du monde :

— $position(x, s)$, x étant un objet (une clef ou l'or), est un fluent fonctionnel dont la valeur est une pièce ou la constante *inventory*, qui désigne que le robot porte l'objet (il est dans son inventaire) dans la situation s.

— $closed(r, s)$ est un fluent relationnel signifiant que la pièce r est fermée à clef dans la situation s.

Une situation initiale possible est décrite par le jeu de formules suivant.

$$position(key(room_3), s_0) = inventory \tag{A.1}$$

$$position(key(room_2), s_0) = room_3 \tag{A.2}$$

$$position(key(room_3), s_0) = room_1 \tag{A.3}$$

$$position(gold, s_0) = room_2 \tag{A.4}$$

$$closed(room_1, s_0) \tag{A.5}$$

$$closed(room_2, s_0) \tag{A.6}$$

$$closed(room_3, s_0) \tag{A.7}$$

L'agent peut effectuer deux actions différentes : ouvrir une porte avec la clef et ramasser un objet dans une pièce ouverte. Les axiomes de préconditions sont les suivants :

$$\forall r\ Poss(openDoorWithKey(r), s) \equiv closed(r, s) \wedge position(key(r), s) = inventory \tag{A.8}$$

$$\forall x\ Poss(take(x), s) \equiv \exists r\ position(x) = r \wedge \neg closed(r) \tag{A.9}$$

Comme on s'y attend, ouvrir une porte a pour effet d'ouvrir une porte, et ramasser un objet à pour effet de le mettre dans l'inventaire. Ceci est traduit dans les axiomes de l'état successeur suivants :

$$closed(r, do(a), s) \equiv a = \neg openDoorWithKey(r) \wedge closed(r, s) \tag{A.10}$$

$$position(x, do(a, s)) = p \equiv a = take(x) \wedge p = inventory \vee a \neq take(x) \wedge p = position(x, s) \tag{A.11}$$

Le programme est le suivant. D'une part, une procédure $loot(r)$ ramasse tous les objets d'une pièce, et le programme principal (procédure main) consiste à ouvrir des pièces et à ramasser leur contenu jusqu'à ce que le robot ait trouvé l'or.

proc $loot(r)$
  while $\exists i.position(i) = r$;
    do $\pi i.(position(i) = r)?; take(i)$
endProc
proc $main$
  while $\neg position(gold) = inventory$;
    do $\Sigma(\pi d.closed(r)?; openWithKey(r); loot(r))$
endProc

```
%%%%%%%%%%%%%%%%%%%%%%%%
%% Roguing the wumpus %%
%%%%%%%%%%%%%%%%%%%%%%%%
%% A lone adventurers enters a dungeon looking for fame and gold.
%% Well, mostly gold.

:- style_check(-discontiguous). % SWI dependent!
:-multifile prim_action/1, causes_val/4, poss/2, proc/2, prim_fluent/1.
:- ['moded_indigolog'].

%%%%%%%%%%%%%%%%%
%% Environment %%
%%%%%%%%%%%%%%%%%

room(1).
room(2).
room(3).

%% I only do this for the money
item(gold).
```

```
%% A key that opens a specific room
item(key(R)) :- room(R).

%% A pick that can be used to open any door
item(pick).

%% Where the items can be
location(X) :- room(X).
location(inventory).


%%%%%%%%%%%%
%% Fluents %%
%%%%%%%%%%%%

prim_fluent(position(X)) :- item(X).
prim_fluent(closed(R)) :- room(R).

initially(position(key(1)), inventory).
initially(position(key(3)), 1).
initially(position(pick), 3).
initially(position(key(2)), 3).
initially(position(gold), 2).
initially(closed(1), true).
initially(closed(2), true).
initially(closed(3), true).
initially(_, false).


%%%%%%%%%%
%% Action %%
%%%%%%%%%%

%% The robot can open doors that aren't locked
%% prim_action(open(D)) :- room(D).
%% poss(open(D), and(closed(D), neg(locked(D)))).
%% causes_val(open(D), closed(D), false, true).

%% It can open locked doors with the key
prim_action(openWithKey(D)) :- room(D).
poss(openWithKey(D), and(closed(D), position(key(D)) = inventory)).
```

```
%% causes_val(openWithKey(D), locked(D), false, true).
causes_val(openWithKey(D), closed(D), false, true).

%% Or, it can just break them
prim_action(break(D)) :- room(D).
poss(break(D), closed(D)).
%% causes_val(break(D), locked(D), false, true).
causes_val(break(D), closed(D), false, true).

%% TODO : lockpick

prim_action(lockpick(D)) :- room(D).
poss(lockpick(D), and(closed(D), position(pick) = inventory)).
causes_val(lockpick(D), closed(D), false, true).

%% Once the doors are open, it can loot the room
prim_action(take(X)) :- item(X).
poss(take(X), some(d, and(and(room(d), position(X) = d), neg(closed(d))))).
causes_val(take(X), position(X), inventory, true).

%%TODO : drop

%%%%%%%%%%%%%%%%%%%%%%%%%
%% Some debugging tools %%
%%%%%%%%%%%%%%%%%%%%%%%%%
prim_action(debug(F)) :- prim_fluent(F).
poss(debug(_), true).
execute(debug(F), _, H) :- write('value of '), write(F), write(' : '),
   has_val(F, V, H), write(V), nl.

prim_action(debug_all).
poss(debug_all, true).
execute(debug_all, _, H) :- forall(prim_fluent(F), execute(debug(F), _, H)).

%%%%%%%%%%%%%%%%
%% Boilerplate %%
%%%%%%%%%%%%%%%%
execute(A,Sr,_) :- ask_execute(A,Sr).
exog_occurs(_) :- fail.

%%%%%%%%%%%%%%%
```

```
%% Procedures %%
%%%%%%%%%%%%%%%

%proc(open*(D), ndet(ndet(open(D), openWithKey(D)), break(D))).
proc(loot(D), while(some(i, position(i) = D),
    pi(i, [?(position(i) = D), take(i)]))).

proc(main, while(neg(position(gold) = inventory),
 search(pi(d, [?(closed(d)), openWithKey(d), loot(d)])))).

%%%%%%%%%%%%%%%%%%%%
%% Personification %%
%%%%%%%%%%%%%%%%%%%%

:- ['process'].

family([openWithKey(X), break(X), lockpick(X)]).

attributes(openWithKey(_), []).
attributes(break(_), [brutal]).
attributes(lockpick(_), [skilled]).

profile1(brutal, -1).
profile1(skilled, 1).

profile2(brutal, 2).
profile2(skilled, 0).


demo(Profile) :- process(Profile, main, P), write(P), nl, indigolog(P).
demo :- indigolog(main), demo(profile1), demo(profile2).
```

Annexe B

Preuves

Les définitions des prédicats *Trans* et *Final* pour les opérateurs Golog sont rappelés en annexe C.

B.1 Possibilité d'exécution du choix préférentiel

Lemme 1.

$$Do(\delta_1|\delta_2, s, s') \equiv Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$$

Démonstration.

$$\begin{aligned}
& Do(\delta_1|\delta_2, s, s') \\
\equiv & \exists \delta'. Trans^*(\delta_1|\delta_2, s, \delta', s') \wedge Final(\delta', s') \\
\equiv & \exists \delta', \delta'', s''. Trans(\delta_1|\delta_2, s, \delta'', s'') \wedge Trans(\delta'', s'', \delta', s') \wedge Final(\delta', s') \\
\equiv & \exists \delta', \delta'', s''. [Trans(\delta_1, s, \delta'', s'') \vee Trans(\delta_2, s, \delta'', s'')] \wedge Trans(\delta'', s'', \delta', s') \wedge Final(\delta', s') \\
\equiv & \exists \delta'. [Trans^*(\delta_1, s, \delta', s') \vee Trans^*(\delta_2, s, \delta', s')] \wedge Final(\delta', s') \\
\equiv & Do(\delta_1, s, s') \vee Do(\delta_2, s, s') \qquad \text{(B.1)}
\end{aligned}$$

□

Lemme 2.

$$Do(\delta, s, s') \implies \exists \delta'', s''. Trans(\delta, s, \delta'', s'')$$

Démonstration. Immédiat à partir de la définition de *Do*. □

Théorème 1. *Si $\delta_1 \mid \delta_2$ peut s'exécuter, alors $\delta_1 \rangle \delta_2$ le peut aussi :*

$$\exists s'. Do(\delta_1 \mid \delta_2, s, s') \implies \exists s''. Do(\delta_1 \rangle \delta_2, s, s''). \qquad (4.5)$$

Démonstration. Démonstration par l'absurde. Supposons

$$\exists s'. Do(\delta_1|\delta_2, s, s') \qquad \text{(B.2)}$$

et

$$\neg \exists s''. Do(\delta_1 \rangle \delta_2, s, s'') \qquad \text{(B.3)}$$

$$
\begin{aligned}
&\neg\exists s'.Do(\delta_1\rangle\delta_2, s, s') \\
\equiv&\neg\exists\delta'.Trans^*(\delta_1\rangle\delta_2, s, \delta', s') \wedge Final(\delta', s) \\
\equiv&\neg\exists\delta'\delta''s''.Trans(\delta_1\rangle\delta_2, s, \delta'', s'') \wedge Trans^*(\delta'', s'', \delta', s') \wedge Final(\delta', s') \\
\equiv&\forall\delta'\delta''s''.\neg Trans(\delta_1\rangle\delta_2, s, \delta'', s'') \vee \neg Trans^*(\delta'', s'', \delta', s')\neg \vee Final(\delta', s')
\end{aligned} \tag{B.4}
$$

Or

$$
\begin{aligned}
&\neg Trans(\delta_1\rangle\delta_2, s, \delta'', s'') \\
\equiv&\neg[\exists s''.Do(\delta_1, s, s'') \wedge Trans(\delta_1, s, \delta', s') \vee \neg\exists s''.Do(\delta_1, s, s'') \wedge Trans(\delta_2, s, \delta', s')] \\
\equiv&[\neg(\exists s''.Do(\delta_1, s, s'')) \vee \neg Trans(\delta_1, s, \delta', s')] \wedge [\exists s''.Do(\delta_1, s, s'') \vee \neg Trans(\delta_2, s, \delta', s')] \\
\equiv&\neg(\exists s''.Do(\delta_1, s, s'')) \wedge \exists s''.Do(\delta_1, s, s'') \\
&\vee \neg(\exists s''.Do(\delta_1, s, s'')) \wedge \neg Trans(\delta_2, s, \delta', s') \\
&\vee \neg Trans(\delta_1, s, \delta', s') \wedge \exists s''.Do(\delta_1, s, s'') \\
&\vee \neg Trans(\delta_1, s, \delta', s') \wedge \neg Trans(\delta_2, s, \delta', s')
\end{aligned} \tag{B.5}
$$

$\neg(\exists s''.Do(\delta_1, s, s'')) \wedge \exists s''.Do(\delta_1, s, s'')$ est faux de manière évidente.

$\neg Trans(\delta_1, s, \delta', s') \wedge \exists s''.Do(\delta_1, s, s'')$ est faux d'après le lemme 2.

$(\neg\exists s''.Do(\delta_1, s, s'')) \implies \exists s''.Do(\delta_2, s, s'')$, d'après le lemme 1 et les hypothèses. On peut donc en déduire $\neg(\exists s''.Do(\delta_1, s, s'')) \wedge \neg Trans(\delta_2, s, \delta', s')$ d'après le lemme 2.

Donc

$$
\begin{aligned}
&\neg Trans(\delta_1\rangle\delta_2, s, \delta'', s'') \\
\implies&\neg Trans(\delta_1, s, \delta', s') \wedge \neg Trans(\delta_2, s, \delta', s') \\
\equiv&\neg Trans(\delta_1|\delta_2, s, \delta', s')
\end{aligned} \tag{B.6}
$$

Donc, d'après B.4 et B.6

$$
\begin{aligned}
&\neg\exists s'.Do(\delta_1\rangle\delta_2, s, s') \\
\implies&\forall\delta'\delta''s''.\neg Trans(\delta_1|\delta_2, s, \delta'', s'') \vee \neg Trans^*(\delta'', s'', \delta', s')\neg \vee Final(\delta', s') \\
\equiv&\neg\exists\delta'\delta''s''.Trans(\delta_1|\delta_2, s, \delta'', s'') \wedge Trans^*(\delta'', s'', \delta', s') \wedge Final(\delta', s') \\
\equiv&\neg\exists\delta'.Trans^*(\delta_1|\delta_2, s, \delta', s') \wedge Final(\delta', s) \\
\equiv&\neg\exists s'.Do(\delta_1|\delta_2, s, s')
\end{aligned} \tag{B.7}
$$

Ce qui est en contradiction avec les hypothèses. Donc

$$
\exists s'.Do(\delta_1|\delta_2, s, s') \implies \exists s''.Do(\delta_1\rangle\delta_2, s, s'') \tag{B.8}
$$

□

B.2 Restrictions

Lemme 3. *La propriété «est une restriction de» est transitive et réflexive.*

Démonstration. La démonstration de la transitivé se fait par récurrence sur le nombre d'étapes élémentaire des programmes. Supposons la propriété vérifiée au rang $n-1$. Supposons δ_1, δ_2 et δ_3 trois programmes dont l'exécution peut faire au maximum n actions élémentaires. Supposons que δ_3 est une restriction de δ_2 et que δ_2 est une restriction de δ_1.

$$Final(\delta_3, s) \implies Final(\delta_2, s)$$
$$Trans(\delta_3, s, \delta_3', s') \implies \exists \delta_2'\ Trans(\delta_2, s, \delta_2', s') \wedge restriction(\delta_3', \delta_2')$$
$$Final(\delta_2, s) \implies Final(\delta_1, s)$$
$$Trans(\delta_2, s, \delta_2', s') \implies \exists \delta_1'\ Trans(\delta_1, s, \delta_1', s') \wedge restriction(\delta_2', \delta_1')$$

Donc

$$Final(\delta_3, s) \implies Final(\delta_1, s)$$
$$Trans(\delta_3, s, \delta_3', s') \implies \exists \delta_2'\ \exists \delta_1'\ Trans(\delta_1, s, \delta_1', s') \wedge restriction(\delta_2', \delta_1') \wedge restriction(\delta_3', \delta_2')$$

δ_1', δ_2' et δ_3' sont le fruit de l'exécution d'une étape élémentaire dans δ_1, δ_2 et δ_3, leur exécution peut donc prendre au maximum $n-1$ étapes élémentaires. Donc

$$restriction(\delta_2', \delta_1') \wedge restriction(\delta_3', \delta_2') \implies restriction(\delta_3', \delta_1')$$

D'où

$$Trans(\delta_3, s, \delta_3', s') \implies \exists \delta_1'\ Trans(\delta_1, s, \delta_1', s') \wedge restriction(\delta_3', \delta_1')$$

C'est à dire $restriction(\delta_3, \delta_1)$. Donc, par récurrence, la propriété *restriction* est transitive. □

Démonstration. La démonstration pour la réflexivité se fait également par récurrence sur le même principe :

$$Final(\delta_1, s) \implies Final(\delta_1, s)$$
$$Trans(\delta_1, s, \delta_1', s') \implies \exists \delta_2'\ Trans(\delta_1, s, \delta_1', s') \wedge restriction(\delta_1', \delta_1')$$

□

Lemme 4. *$\delta_1 \rangle \delta_2$ est une restriction de $\delta_1 | \delta_2$.*

Démonstration. Par définition :

$$\begin{aligned} & Final(\delta_1 \rangle \delta_2, s) \\ \equiv & Final(\delta_1, s) \vee \neg \exists s".Do(\delta_1, s, s") \wedge Final(\delta_2, s) \\ \implies & Final(\delta_1, s) \vee Final(\delta_2, s) \\ \equiv & Final(\delta_1 | \delta_2, s) \end{aligned} \tag{B.9}$$

$$\begin{aligned} & Trans(\delta_1 \rangle \delta_2, s, \delta', s') \\ \equiv\ & \exists s".Do(\delta_1, s, s") \wedge Trans(\delta_1, s, \delta', s') \\ & \vee \neg \exists s".Do(\delta_1, s, s") \wedge Trans(\delta_2, s, \delta', s') \\ \implies\ & Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\ \equiv\ & Trans(\delta_1 | \delta_2, s, \delta', s') \end{aligned} \tag{B.10}$$

□

Lemme 5. *Si δ_1 est une restriction de δ_2, alors $\delta_1|\delta_2$ est une restriction de δ_2.*

Démonstration. En supposant $restriction(\delta_1, \delta_2)$.

$$\begin{aligned} & Final(\delta_1|\delta_2, s) \\ \equiv\ & Final(\delta_1, s) \vee Final(\delta_2, s) \\ \Longrightarrow\ & Final(\delta_2, s) \vee Final(\delta_2, s) \\ \equiv\ & Final(\delta_2, s) \end{aligned} \tag{B.11}$$

$$\begin{aligned} & Trans(\delta_1|\delta_2, s, \delta', s') \\ \equiv\ & Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\ \Longrightarrow\ & [\exists \delta_2'\ Trans(\delta_2, s, \delta_2, s') \wedge restriction(\delta_2', \delta')] \vee Trans(\delta_2, s, \delta', s') \\ \Longrightarrow\ & [\exists \delta_2'\ Trans(\delta_2, s, \delta_2', s') \wedge restriction(\delta_2', \delta')] \vee [\exists \delta_2''\ Trans(\delta_2, s, \delta_2'', s') \wedge restriction(\delta_2'', \delta')] \\ \equiv\ & \exists \delta_2'\ Trans(\delta_2, s, \delta_2', s') \wedge restriction(\delta_2', \delta') \end{aligned} \tag{B.12}$$

□

Lemme 6. *δ_1 et δ_2 sont des restrictions de $\delta_1|\delta_2$.*

Démonstration.

$$\begin{aligned} & Final(\delta_1, s) \\ \Longrightarrow\ & Final(\delta_1, s) \vee Final(\delta_2, s) \\ \equiv\ & Final(\delta_1|\delta_2, s) \end{aligned} \tag{B.13}$$

$$\begin{aligned} & Trans(\delta_1, s, \delta', s') \\ \Longrightarrow\ & Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\ \equiv\ & Trans(\delta_1|\delta_2, s, \delta', s') \end{aligned} \tag{B.14}$$

De même δ_2. □

Lemme 7. *$\pi x.\phi(x)?;\delta(x)$ est une restriction de $\pi x.\delta(x)$*

Démonstration.

$$\begin{aligned} & Final(\pi x.\phi(x)?;\delta(x), s) \\ \equiv\ & \exists x\ Final(\phi(x)?;\delta(x), s) \\ \equiv\ & \exists x\ Final(\phi(x)?, s) \wedge Final(\delta(x), s) \\ \Longrightarrow\ & \exists x\ Final(\delta(x), s) \\ \equiv\ & Final(\pi x\, \delta(x), s) \end{aligned} \tag{B.15}$$

$$\begin{aligned} & Trans(\pi x.\phi(x)?;\delta(x), s, \delta', s') \\ \equiv\ & \exists x\ Trans(\phi(x)?;\delta(x), s, \delta', s') \\ \equiv\ & \exists x\ Trans(\phi(x)?, s, \delta', s') \wedge Trans(\delta(x), s, \delta', s') \\ \Longrightarrow\ & \exists x\ Trans(\delta(x), s, \delta', s') \\ \equiv\ & Trans(\pi x\, \delta(x), s, \delta', s') \end{aligned} \tag{B.16}$$

□

Théorème 3. *Soit δ_t le programme obtenu par transformation de δ avec la transformation T1, T2, T3, T4 ou T5. Alors δ_t est une restriction de δ :*

Démonstration. Pour T1, cela découle immédiatement du lemme 4.

Pour T2, d'après le lemme 4 $\pi x.\phi(x)?;\delta(x)\rangle\pi x.\delta(x)$ est une restriction de $\pi x.\phi(x)?;\delta(x)|\pi x.\delta(x)$. Or, d'après le lemme 7, $\pi x.\phi(x)?;\delta(x)$ est une restriction de $\pi x.\delta(x)$. Donc, d'après le lemme 5, le programme $\pi x.\phi(x)?;\delta(x)|\pi x.\delta(x)$ est une restriction de $\pi x.\delta(x)$. De fait, $\pi x.\phi(x)?;\delta(x)\rangle\pi x.\delta(x)$ est une restriction de $\pi x.\delta(x)$.

Pour T3, cela découle directement du lemme 6.

Pour T4, cela découle directement du lemme 7. □

Annexe C

Sémantique de IndiGolog

Le sens des prédicats *Trans* et *Final* est donné lors de l'introduction de ConGolog dans la section 2.2.2. Pour rappel :

> Tout d'abord, le prédicat *Trans* qui exprime comment exécuter une étape d'un programme : $Trans(\delta, s, \delta', s')$ exprime que l'exécution d'une étape de δ dans la situation s mène à la situation s', avec δ' restant à être exécuté. Évidemment, *Trans* ne suffit pas, car il ne permet pas d'exprimer qu'un programme est terminé. Cela est fait par le prédicat *Final* : $Final(\delta, s)$ exprime que le programme δ peut être considéré comme terminé dans la situation s (plus rien à exécuter).

On utilise dans cette annexe la notion de configuration : une paire (δ, s) est une configuration si δ est un programme et s une situation. On dira que la configuration (δ, s) évolue en (δ', s') si la formule $Trans(\delta, s, \delta', s')$ est vraie, et que la configuration (δ, s) est finale si $Final(\delta, s)$.

C.1 Programme vide

$$Trans(nil, s, \delta, s') \equiv False$$

$$Final(nil, s) \equiv True$$

(nil, s) ne peut jamais évoluer et est toujours une configuration finale.

C.2 Action primitive

$$Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s)$$

$$Final(a, s) \equiv False$$

(a, s) évolue en $nil, do(a[s], s))$ si $a[s]$ est possible dans s. a désigne un programme Golog tandis que $a[s]$ désigne l'action du calcul des situations correspondantes. (a, s) n'est jamais finale, car a reste à exécuter.

C.3 Test ou attente

$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nul \wedge s' = s'$$

$$Final(\phi?, s) \equiv False$$

$(\phi?, s)$ évolue en (nil, s) si $\phi[s]$ est vrai. Comme précédemment, ϕ désigne la condition Golog et $\phi[s]$ la formule du calcul des situations correspondantes. $(\phi?, s)$ n'est jamais finale car le test reste à évaluer.

C.4 Séquence

$$Trans(\delta_1;\delta_2, s, \delta', s') \equiv$$
$$\exists\gamma.\delta' = (\gamma;\delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s')$$

$$Final(\delta_1;\delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

$(\delta_1;\delta_2, s)$ peut évoluer en exécutant δ_1 si ce dernier n'est pas final, ou δ_2 s'il l'est. $(\delta_1;\delta_2, s)$ est finale si (δ_1, s) et (δ_2, s) le sont.

C.5 Branchement non-déterministe

$$Trans(\delta_1|\delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_1, s, \delta', s')$$

$$Final(\delta_1;|\delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$$

La configuration $(\delta_1;|\delta_2, s)$ peut évoluer (respectivement est finale) si l'un ou l'autre de (δ_1, s) ou (δ_2, s) le peut (respectivement est finale).

C.6 Choix non-déterministe d'argument

$$Trans(\pi v.\delta, s, \delta', s') \equiv \exists x.Trans(\delta^v_x, s, \delta', s')$$

$$Final(\pi v.\delta, s) \equiv \exists x.Final(\delta^v_x, s)$$

$(\pi v.\delta, s)$ peut évoluer (respectivement est finale) si il existe un x tel que (δ^v_x, s) peut évoluer (respectivement est finale. δ^v_x est le programme résultant de la substitution de v par x dans δ

C.7 Itération non-déterministe

$$Trans(\delta^*, s, \delta', s') \equiv \exists\gamma.(\delta' = \gamma;\delta^*) \wedge Trans(\delta, s, \gamma, s')$$

$$Final(\delta^*, s) \equiv True$$

(δ^*, s) peut évoluer en (δ', δ^*, s), c'est à dire en exécutant δ puis à nouveau δ^*. (δ^*, s) peut aussi ne pas évoluer, puisqu'elle est toujours finale.

C.8 Conditionnelle synchronisée

$$Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf}, s, \delta', s') \equiv$$
$$\phi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s')$$

$$Final(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf}, s) \equiv \phi[s] \wedge Final(\delta_1, s) \vee \neg\phi[s] \wedge Final(\delta_2, s)$$

if ϕ **then** δ_1 **else** δ_2 **endIf** se comporte comme δ_1 ou δ_2 en fonction de la valeur de $\phi[s]$. Cette construction est synchronisée, c'est-à-dire que rien ne peut se produire entre l'évaluation de $\phi[s]$ et l'exécution de la première étape de δ_1 ou δ_2.

C.9 Boucle synchronisée

$$Trans(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s, \delta', s') \equiv$$
$$\exists `\gamma.(\delta' = \gamma; \textbf{while } \phi \textbf{ do } \delta) \wedge \phi[s] \wedge Trans(\delta, s, \gamma, s')$$

$$Final(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s) \equiv \neg\phi[s] \vee Final(\delta, s)$$

while ϕ **do** δ **endWhile** est une version synchronisée de $\delta^*; \neg\phi? : \delta$ est exécuté tant que ϕ est vrai. (**while** ϕ **do** δ **endWhile**, s) est finale si $\phi[s]$ est faux.

C.10 Exécution concurrente

$$Trans(\delta_1||\delta_2, s, \delta', s) \equiv$$
$$\exists\gamma.\delta' = (\gamma||\delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \exists\gamma.\delta' = (\delta_1||\gamma) \wedge Trans(\delta_1, s, \gamma, s')$$

$$Final(\delta_1||\delta_2, s) \equiv Final(\delta_1) \wedge Final(\delta_2)$$

Exécuter une étape de $\delta_1||\delta_2$ consiste à exécuter une étape de l'un en laissant l'autre inchangé. $\delta_1||\delta_2$ est terminé si δ_1 et δ_2 le sont.

C.11 Concurrence priorisée

$$Trans(\delta_1 \rangle\rangle \delta_2, s, \delta', s') \equiv$$
$$\exists\gamma.\delta' = (\gamma \rangle\rangle \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee$$
$$\exists\gamma.\delta' = (\delta_1 \rangle\rangle \gamma) \wedge Trans(\delta_2, s, \gamma, s') \wedge \neg\exists\zeta, s''.Trans(\delta_1, s, \zeta, s'')$$

$$Final(\delta_1||\delta_2, s) \equiv Final(\delta_1) \wedge Final(\delta_2)$$

Cette construction se comporte comme la précédente, à ceci près qu'une étape de δ_2 ne peut être exécutée que si aucune étape n'est possible pour δ_1. La condition de terminaison est la même.

C.12 Itération concurrente

$$Trans(\delta^{||}, s, \delta', s') \equiv \exists \gamma . \delta' = (\gamma || \delta^{||}) \wedge Trans(\delta, s, \gamma, s')$$

$$Final(\delta^{||}, s) \equiv True$$

Exécuter une étape de $\delta^{||}$ consiste à exécuter une étape de δ en parallèle avec $\delta^{||}$ lui-même. Cela signifie qu'un nombre quelconque d'instances de δ peut tourner en parallèle. $\delta^{||}$ est toujours final : il est possible de n'exécuter aucune instance de δ.

Bibliographie

[ABB+04] G. Antoniou, M. Baldoni, C. Baroglio, V. Patti, R. Baumgartner, T. Eiter, M. Herzog, R. Schindlauer, H. Tompits, F. Bry, et al. Reasoning methods for personalization on the semantic web. *Annals of Mathematics*, 2 :1–24, 2004.

[AMG13] Sean Andrist, Bilge Mutlu, and Michael Gleicher. Conversational gaze aversion for virtual agents. In *Intelligent Virtual Agents*, pages 249–262. Springer, 2013.

[AV05] Alessandro Acquisti and Hal R Varian. Conditioning prices on purchase history. *Marketing Science*, 24(3) :367–381, 2005.

[Bat94] Joseph Bates. The role of emotion in believable agents. *Commun. ACM*, 37(7) :122–125, 1994.

[BBCP01] Matteo Baldoni, Cristina Baroglio, Alessandro Chiarotto, and Viviana Patti. Programming goal-driven web sites using an agent logic language. In *Practical Aspects of Declarative Languages*, pages 60–75. Springer, 2001.

[BBH05] M. Baldoni, C. Baroglio, and N. Henze. Personalization for the semantic web. *Reasoning Web*, pages 95–95, 2005.

[BBP04] M. Baldoni, C. Baroglio, and V. Patti. Web-based adaptive tutoring : an approach based on logic agents and reasoning about actions. *Artificial Intelligence Review*, 22(1) :3–39, 2004.

[BF+95] C. Boutilier, N. Friedman, et al. Nondeterministic actions and the frame problem. In *Workign Notes of the AAAI Spring Symposium on Extending Theories of Action*, pages 39–44, 1995.

[BFPAGS+08] Yolanda Blanco-Fernández, José J Pazos-Arias, Alberto Gil-Solla, Manuel Ramos-Cabrer, and Martín López-Nores. Semantic reasoning : A path to new possibilities of personalization. In *The Semantic Web : Research and Applications*, pages 720–735. Springer, 2008.

[BGMP01] Matteo Baldoni, Laura Giordano, Alberto Martelli, and Viviana Patti. Reasoning about complex actions with incomplete knowledge : a modal approach. In *Theoretical Computer Science*, pages 405–426. Springer, 2001.

[BHL98] F. Bacchus, J.Y. Halpern, and H.J. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Arxiv preprint cs/9809013*, 1998.

[BK94] Michael P Bieber and Steven O Kimbrough. On the logic of generalized hypertext. *Decision Support Systems*, 11(2) :241–257, 1994.

[BM04] Dave Beckett and Brian McBride. Rdf/xml syntax specification (revised). *W3C recommendation*, 10, 2004.

[BM06] Jorge A Baier and Sheila A McIlraith. On planning with programs that sense. *KR*, 6 :492–502, 2006.

[BP03] J.A. Baier and J.A. Pinto. Planning under uncertainty as golog programs. *Journal of Experimental & Theoretical Artificial Intelligence*, 15(4) :383–405, 2003.

[BRS+00] Craig Boutilier, Raymond Reiter, Mikhail Soutchanski, Sebastian Thrun, et al. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the National Conference on Artificial Intelligence*, pages 355–362. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999, 2000.

[Bru92] PL Brusilovsky. Intelligent tutor, environment and manual for introductory programming. *Educational and Training Technology International*, 29(1) :26–34, 1992.

[Bru96] P. Brusilovsky. Methods and techniques of adaptive hypermedia. *User modeling and user-adapted interaction*, 6(2) :87–129, 1996.

[BS11] François Bouchet and Jean-Paul Sansonnet. Influence of personality traits on the rational process of cognitive agents. In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology-Volume 02*, pages 81–88. IEEE Computer Society, 2011.

[BSW96] Peter Brusilovsky, Elmar Schwarz, and Gerhard Weber. Elm-art : An intelligent tutoring system on world wide web. In *Intelligent tutoring systems*, pages 261–269. Springer, 1996.

[Bur99] Robin Burke. The wasabi personal shopper : A case-based recommender system. In *In Proceedings Of The 11th National Conference On Innovative Applications Of Artificial Intelligence*, pages 844–849. AAAI Press, 1999.

[BVJPO13] Timothy Bickmore, Laura Vardoulakis, Brian Jack, and Michael Paasche-Orlow. Automated promotion of technology acceptance by clinicians using relational agents. In *Intelligent Virtual Agents*, pages 68–78. Springer, 2013.

[CCL99] Robert Chignoli, Pierre Crescenzo, and Philippe Lahire. Customization of links between classes. *Rapport Technique*, pages 99–18, 1999.

[CELN07] Jens Claßen, Patrick Eyerich, Gerhard Lakemeyer, and Bernhard Nebel. Towards an integration of golog and planning. In *Proc. IJCAI*, volume 7, pages 1846–1851, 2007.

[CFTJ09] G. Castellano, A. Fanelli, M. Torsello, and L. Jain. Innovations in web personalization. *Web Personalization in Intelligent Environments*, pages 1–26, 2009.

[CL07] Yukun Cao and Yunfeng Li. An intelligent fuzzy-based recommendation system for consumer electronic products. *Expert Systems with Applications*, 33(1) :230–240, 2007.

[CM92] P. T. Costa and R. R. McCrae. *The NEO PI-R professional manual*. Odessa, FL : Psychological Assessment Resources, 1992.

[CV04] Alexandra Cristea and Michael Verschoor. The lag grammar for authoring the adaptive web. In *Information Technology : Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 1, pages 382–386. IEEE, 2004.

[Das08] Mehdi Dastani. 2APL : a practical agent programming language. In *AAMAS '08 :The seventh international joint conference on Autonomous agents and multiagent systems*, volume 16, pages 214–248, Estoril, Portugal, 2008. Springer-Verlag.

[Dav89] Fred D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3) :319–340, 1989.

[DBHW99] Paul De Bra, Geert-Jan Houben, and Hongjing Wu. Aham : a dexter-based reference model for adaptive hypermedia. In *Proceedings of the tenth ACM Conference on Hypertext and hypermedia : returning to our diverse roots : returning to our diverse roots*, pages 147–156. ACM, 1999.

[DBSS05] Paul De Bra, Natalia Stash, and David Smits. Creating adaptive web-based applications. In *Tutorial at the 10th International Conference on User Modeling, Edinburgh, Scotland*. Citeseer, 2005.

[DGL99a] Giuseppe De Giacomo and Hector J Levesque. An incremental interpreter for high-level programs with sensing. In *Logical Foundations for Cognitive Agents*, pages 86–102. Springer, 1999.

[DGL99b] Giuseppe De Giacomo and Hector J Levesque. Projection using regression and sensors. In *International joint conference on artificial intelligence*, volume 16, pages 160–165. LAWRENCE ERLBAUM ASSOCIATES LTD, 1999.

[DGLS01] Giuseppe De Giacomo, Hector J Levesque, and Sebastian Sardina. Incremental execution of guarded theories. *ACM Transactions on Computational Logic (TOCL)*, 2(4) :495–525, 2001.

[DHN03] P. Dolog, N. Henze, and W. Nejdl. Logic-based open hypermedia for the semantic web. In *Proceedings of the Int. Workshop on Hypermedia and the Semantic Web, Hypertext 2003 Conference, Nottinghan, UK*. Citeseer, 2003.

[DHNS03] P. Dolog, N. Henze, W. Nejdl, and M. Sintek. Towards the adaptive semantic web. *Principles and Practice of Semantic Web Reasoning*, pages 51–68, 2003.

[DPB11a] Georges Dubus, Fabrice Popineau, and Yolaine Bourda. A formal approach to personalization. In *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on*, pages 233–238. IEEE, 2011.

[DPB11b] Georges Dubus, Fabrice Popineau, and Yolaine Bourda. Situation calculus and personalized web systems. In *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*, pages 569–574. IEEE, 2011.

[DPBS13] Georges Dubus, Fabrice Popineau, Yolaine Bourda, and Jean-Paul Sansonnet. Parametric reasoning agents extend the control over standard behaviors. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2013 IEEE/WIC/ACM International Joint Conferences on*, volume 2, pages 163–170. IEEE, 2013.

[ERBP08] Rick Evertsz, Franck E. Ritter, Paolo Busetta, and Matteo Pedrotti. Realistic behaviour variation in a BDI-based cognitive architecture. In *Proc. of SimTecT'08*, Melbourne, Australia, 2008.

[FFL04] Alexander Ferrein, Christian Fritz, and Gerhard Lakemeyer. On-line decision-theoretic golog for unpredictable domains. In *KI 2004 : Advances in Artificial Intelligence*, pages 322–336. Springer, 2004.

[FM06] C. Fritz and S. McIlraith. Decision-theoretic golog with qualitative preferences. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR06)*, pages 153–163, Lake District, UK, June 2006.

[FMRR90] Gerhard Fischer, Thomas Mastaglio, Brent Reeves, and John Rieman. Minimalist explanations in knowledge-based systems. In *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on*, volume 3, pages 309–317. IEEE, 1990.

[FNW98] Peter Fröhlich, Wolfgang Nejdl, and Martin Wolpers. Kbs-hyperbook-an open hyperbook system for education. In *Proceedings of the ED-MEDIA World Conference on Educational Multimedia and Hypermedia*. Citeseer, 1998.

[FU10] F. Fischer and G. Unel. Reasoning in semantic web-based systems. *Semantic Web Information Management*, 1 :127, 2010.

[G$^+$09] W3C Owl Working Group et al. Owl 2 web ontology language document overview. *W3C Recommendation*, 27 :1205–1214, 2009.

[GL00] H. Grosskreutz and G. Lakemeyer. Turning high-level plans into robot programs in uncertain domains. In *ECAI*, pages 548–552, 2000.

[GL01] H. Grosskreutz and G. Lakemeyer. Belief update in the pgolog framework. *KI 2001 : Advances in Artificial Intelligence*, pages 213–228, 2001.

[GLL00] G. De Giacomo, Y. Lespérance, and H.J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1) :109–169, 2000.

[GLLS09] G. Giacomo, Y. Lespérance, H.J. Levesque, and S. Sardina. IndiGolog : A high-level programming language for embedded reasoning agents. *Multi-Agent Programming :*, pages 31–72, 2009.

[GM04] Jonathan Gratch and Stacy Marsella. A domain-independent framework for modeling emotion. *Journal of Cognitive Systems Research*, 5(4) :269–306, 2004.

[Gol90] L. R. Goldberg. An alternative description of personality : The big-five factor structure. *Journal of Personality and Social Psychology*, 59 :1216–1229, 1990.

[Hay04] B. Hayes-Roth. What makes characters seem life-like ? In *Life-like Characters. Tools, Affective Functions and Applications*, Heidelberg, 2004. Springer.

[HDN04] N. Henze, P. Dolog, and W. Nejdl. Reasoning and ontologies for personalized e-learning in the semantic web. *Educational Technology & Society*, 7(4) :82–97, 2004.

[HPSB+04] Ian Horrocks, Peter F Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, et al. Swrl : A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21 :79, 2004.

[HRHL01] Nick Howden, Ralph Rannquist, Andrew Hodgson, and Andrew Lucas. Intelligent agents - summary of an agent infrastructure. In *Proc. of the 5th International Conference on Autonomous Agents*, Montreal, 2001.

[Jac06] Cédric Jacquiot. *Modélisation logique et générique des systèmes d'hypermédias adaptatifs*. PhD thesis, PhD thesis, Department of Computer science, France, Supelec, 2006.

[JBP+06] C. Jacquiot, Y. Bourda, F. Popineau, A. Delteil, and C. Reynaud. Glam : A generic layered adaptation model for adaptive hypermedia systems. In *Adaptive Hypermedia and Adaptive Web-Based Systems*, page 131–140. Springer, 2006.

[JRP08] Oliver P. John, Richard W. Robins, and Lawrence A. Pervin, editors. *Handbook of Personality : Theory and Research*. The Guilford Press, 3rd edition, 2008.

[JVH07] Hong Jiang, Jose M. Vidal, and Michael N. Huhns. eBDI : an architecture for emotional agents. In *AAMAS '07 : Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–3, New York, NY, USA, 2007. ACM.

[KBP09] Evgeny Knutov, Paul De Bra, and Mykola Pechenizkiy. Ah 12 years later : a comprehensive survey of adaptive hypermedia methods and techniques. *The New Review of Hypermedia and Multimedia*, 15(1) :5–38, 2009.

[Kim02] Won Kim. Personalization : Definition, status, and challenges ahead. *Journal of object technology*, 1(1) :29–40, 2002.

[KL10] Shakil M Khan and Yves Lespérance. A logical framework for prioritized goal change. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems : volume 1-Volume 1*, pages 283–290. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[Lak99] G. Lakemeyer. On sensing and off-line interpreting in golog. *Logical Foundations for Cognitive Agents. Springer*, pages 173–189, 1999.

[Lan95] Ken Lang. Newsweeder : Learning to filter netnews. In *Proceedings of the 12th International Machine Learning Conference (ML)*, 1995.

[LDAP08] Mei Lim, Joao Dias, Ruth Aylett, and Ana Paiva. Improving adaptiveness in autonomous characters. In Helmut Prendinger, James Lester, and Mitsuru Ishizuka, editors, *Intelligent Virtual Agents*, volume 5208 of *Lecture Notes in Computer Science*, pages 348–355. Springer, 2008.

[Lev96] Hector J. Levesque. What is planning in the presence of sensing ? In William J. Clancey and Daniel S. Weld, editors, *AAAI/IAAI, Vol. 2*, pages 1139–1146. AAAI Press / The MIT Press, 1996.

[LNR87] John E Laird, Allen Newell, and Paul S Rosenbloom. Soar : An architecture for general intelligence. *Artificial intelligence*, 33(1) :1–64, 1987.

[LRL+97] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R.B. Scherl. GOLOG : A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1) :59–83, 1997.

[MCF+94] Tom M Mitchell, Rich Caruana, Dayne Freitag, John McDermott, and David Zabowski. Experience with a learning personal assistant. *Communications of the ACM*, 37(7) :80–91, 1994.

[MCRK07] Lori Malatesta, George Caridakis, Amaryllis Raouzaiou, and Kostas Karpouzis. Agent personality traits in virtual environments based on appraisal theory predictions. In *Artificial and Ambient Intelligence, Language, Speech and Gesture for Expressive Characters,achie AISB'07*, Newcastle, UK, 2007.

[MCS00] Bamshad Mobasher, Robert Cooley, and Jaideep Srivastava. Automatic personalization based on web usage mining. *Communications of the ACM*, 43(8) :142–151, 2000.

[MH68] J. McCarthy and P. Hayes. *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University, 1968.

[MHvdB+12] Tijmen Joppe Muller, Annerieke Heuvelink, Karel van den Bosch, Ivo Swartjes, et al. Glengarry glen ross : Using bdi for sales game dialogues. In *AIIDE*, 2012.

[MJZ03] Bamshad Mobasher, Xin Jin, and Yanzan Zhou. Semantically enhanced collaborative filtering on the web. In *Proceedings of the First EuropeanWeb Mining Forum – EWMF 2003*, pages 57–76. Springer, 2003.

[Moo79] Robert C Moore. *Reasoning about knowledge and action*. PhD thesis, Massachusetts Institute of Technology, 1979.

[MS+00] S.A. McIlraith, R. Scherl, et al. What sensing tells us : Towards a formal theory of testing for dynamical systems. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 483–490. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999, 2000.

[MS01] S. McIlraith and T.C. Son. Adapting golog for programming the semantic web. In *Fifth International Symposium on Logical Formalizations of Commonsense Reasoning*, page 195–202, 2001.

[MS02] S. McIlraith and T.C. Son. Adapting golog for composition of semantic web services. In *Principles of knowledge representation and reasoning-international conference*, page 482–496, 2002.

[MSd+09] M. McRorie, I. Sneddon, E. de Sevin, E. Bevacqua, and C. Pelachaud. A model of personality and emotional traits. In *Intelligent Virtual Agents (IVA 2009)*, volume 5773 of *LNAI*, pages 27–33, Amsterdam, NL, 2009. Springer-Verlag.

[OCC88] Andrew Ortony, Gerald L. Clore, and Allan Collins. *The Cognitive Structure of Emotions*. Cambridge university press, Cambridge, UK, cambridge university press edition, 1988.

[Paz99] Michael J. Pazzani. A framework for collaborative, content-based and demographic filtering. *Artif. Intell. Rev.*, 13(5-6) :393–408, 1999.

[PB07] Michael J. Pazzani and Daniel Billsus. Content-based recommendation systems. In *The Adaptive Web*, volume 4321, pages 325–341. Springer Berlin / Heidelberg, 2007.

[Pel00] C. Pelachaud. Some considerations about embodied agents. In *Int. Conf. on Autonomous Agents*, Barcelona, 2000.

[PPPS03] Dimitrios Pierrakos, Georgios Paliouras, Christos Papatheodorou, and Constantine D Spyropoulos. Web usage mining as a tool for personalization : A survey. *User Modeling and User-Adapted Interaction*, 13(4) :311–372, 2003.

[PS+08] Eric Prud'Hommeaux, Andy Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

[Rei91] R. Reiter. The frame problem in the situation calculus : A simple solution (sometimes) and a completeness result for goal regression. *Artificial intelligence and mathematical theory of computation : papers in honor of John McCarthy*, 27 :359–380, 1991.

[Rei01] R. Reiter. On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logic (TOCL)*, 2(4) :433–457, 2001.

[RG+95] Anand S Rao, Michael P Georgeff, et al. Bdi agents : From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*, pages 312–319. San Francisco, 1995.

[RH96] D. Rousseau and B. Hayes-Roth. Personality in synthetic characters. In *Technical report, KSL 96-21*. Knowledge Systems Laboratory, Stanford University, 1996.

[RIS+94] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens : An open architecture for collaborative filtering of netnews. In *Proceedings of ACM Conference on Computer Supported Cooperative Work*, pages 175–186, 1994.

[RN10] S.J. Russell and P. Norvig. *Artificial intelligence : a modern approach*. Prentice hall, 2010.

[RVMC97] P. Rizzo, M. V. Veloso, M. Miceli, and A. Cesta. Personality-driven social behaviors in believable agents. In *AAAI Symposium on Socially Intelligent Agents*, pages 109–114, 1997.

[Sch99] R. Scherl. A golog specification of a hypertext system. In *Logical Foundations for Cognitive Agents*, page 309–324. Springer-Verlag, 1999.

[Sch01] Eric Schwarzkopf. An adaptive web site for the um2001 conference. In *Proceedings of the UM2001 Workshop on Machine Learning for User Modeling*, pages 77–86, 2001.

[SKKR01] Badrul Sarwar, George Karypis, Joseph Konstan, and John Reidl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web (WWW)*, pages 285–295, 2001.

[SL93] Richard B Scherl and Hector J Levesque. The frame problem and knowledge-producing actions. In *Proceedings of the National Conference on Artificial Intelligence*, pages 689–689. Citeseer, 1993.

[SL03] R.B. Scherl and H.J. Levesque. Knowledge, action, and the frame problem. *Artificial Intelligence*, 144(1) :1–39, 2003.

[SL10] S. Sardina and Y. Lespérance. Golog speaks the bdi language. *Programming Multi-Agent Systems*, pages 82–99, 2010.

[Sou01] Mikhail Soutchanski. An on-line decision-theoretic golog interpreter. In *IJCAI*, pages 19–26. Citeseer, 2001.

[SP07] S. Sardina and L. Padgham. Goals in the context of bdi plan failure and planning. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 7. ACM, 2007.

[SP11] S. Sardina and L. Padgham. A bdi agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1) :18–70, 2011.

[SPLL00] Steven Shapiro, Maurice Pagnucco, Yves Lespérance, and Hector J Levesque. Iterated belief change in the situation calculus. In *PRINCIPLES OF KNOWLEDGE REPRESENTATION AND REASONING-INTERNATIONAL CONFERENCE-*, pages 527–538. Morgan Kaufmann Publishers ; 1998, 2000.

[SS03] Sebastian Sardiña and Steven Shapiro. Rational action in agent programs with prioritized goals. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 417–424. ACM, 2003.

[TS03] Theophanis Tsandilas and M.C. Schraefel. Adaptive presentation supporting focus and context. In *Workshop on Adaptive Hypermedia and Adaptive Web Based Systems*, pages 193–200, 2003.

[VdBBMH12] Karel Van den Bosch, Arjen Brandenburgh, Tijmen Joppe Muller, and Annerieke Heuvelink. Characters with personality ! In *Intelligent Virtual Agents*, pages 426–439. Springer, 2012.

[VDSHLH09] K. Van Der Sluijs, J. Hidders, E. Leonardi, and G.J. Houben. Gal : A generic adaptation language for describing adaptive hypermedia. In *1st International Workshop on Dynamic and Adaptive Hypertext : Generic Frameworks, Approaches and Techniques*, pages 13–24, 2009.

[WHDB98] H. Wu, G.J. Houben, and P. De Bra. Aham : A reference model to support adaptive hypermedia authoring. In *Proceedings of the Conference on Information Science, Antwerp*, volume 51, page 76, 1998.

[Wik14] Wikipedia. Saint-empire romain germanique, 2014.

[WPHT02] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufman, 2002.

[ZBR+09] Nadjet Zemirline, Yolaine Bourda, Chantal Reynaud, Fabrice Popineau, et al. Mesam : A protégé plug-in for the specialization of models. In *Proceedings of the 11th International Protégé Conference*, 2009.