THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par
Université Toulouse III - Paul Sabatier
Discipline ou spécialité :
Informatique

Présentée et soutenue par
Valérian Guivarch
Le
8 avril 2014
Titre :
Prise en compte de la dynamique du contexte pour les systèmes ambiants par systèmes multi-agents adaptatifs

---

JURY
Valérie CAMPS, Maitre de Conférence, Université de Toulouse (encadrant)
Amal EL FALLAH SEGHROUCHNI, Professeur, Université Pierre et Marie Curie (rapporteur)
Pierre GLIZE, Ingénieur HDR, CNRS (invité)
Marie-Pierre GLEIZES, Professeur, Université de Toulouse (examinateur)
Juan PAVON, Professeur, Université de Madrid (rapporteur)
André PENINOU, Maitre de Conférence, Université de Toulouse (encadrant)
Laurent VERCOUTER, Professeur, INSA de Rouen (examinateur)

Ecole doctorale : Mathématiques Informatique Télécommunications (MITT)
Unité de recherche : Institut de Recherche en Informatique de Toulouse (IRIT)
Directeur(s) de Thèse : Pierre Glize, Valérie Camps et André Péninou
Rapporteurs : Amal El Fallah Seghrouchni et Juan Pavón

Valérian Guivarch

PRISE EN COMPTE DE LA DYNAMIQUE DU CONTEXTE POUR LES SYSTÈMES AMBIANTS PAR SYSTÈMES MULTI-AGENTS ADAPTATIFS

Directeur :	Pierre Glize, Ingénieur de Recherche HDR, CNRS
Co-Encadrante :	Valérie Camps, Maître de Conférences, UPS
Co-Encadrant :	André Péninou, Maître de Conférences, IUT Blagnac

Résumé

Les systèmes ambiants se composent de nombreux appareils électroniques hétérogènes, souvent mobiles (PDA) ou intégrés (vêtements...), distribués dans l'environnement et interagissant de façon dynamique. Dès lors, l'individu est au centre des préoccupations de la conception de ces systèmes qui peuvent et doivent s'adapter au contexte des utilisateurs et à l'environnement dans lequel ils sont situés. On parle alors de systèmes sensibles au contexte. Cependant, la forte dynamique des systèmes ambiants rend difficile, voire impossible, d'établir à l'avance pour de tels systèmes toutes les règles d'adaptation nécessaires. Permettre aux systèmes ambiants d'être capables d'apprendre par eux-mêmes et dynamiquement quand et comment s'adapter aux comportements des utilisateurs est une direction de recherche intéressante pour apporter une solution à ce problème. Parmi les nombreuses techniques d'apprentissage existantes, peu sont réellement applicables aux systèmes ambiants. En effet, du fait de l'apparition ou la disparition dynamique de dispositifs dans un système ambiant, le nombre de variables à prendre en compte dans l'apprentissage évolue en cours de fonctionnement. Peu de systèmes d'apprentissage peuvent gérer cette contrainte. De plus les changements comportementaux des utilisateurs entraînent des évolutions de la fonctionnalité à apprendre auxquelles le système doit s'adapter sans devoir recommencer entièrement le processus d'apprentissage.

Nous proposons d'utiliser l'approche par Systèmes Multi-Agents Adaptatifs (ou AMAS) afin d'apprendre un comportement pertinent pour un système ambiant en se basant sur l'observation des actions des utilisateurs. L'approche par AMAS permet d'apporter des solutions à des problèmes pouvant être incomplètement spécifiés et pour lesquels il n'existe pas de solution algorithmique *a priori* connue. Elle construit une solution par une approche locale coopérative qui ne dépend pas directement de la fonctionnalité attendue du système à concevoir.

Dans le cadre de notre problème, il s'agit de concevoir un système capable d'observer les actions récurrentes des utilisateurs et d'établir dans quels contextes ces actions sont réalisées afin de suppléer l'utilisateur si une situation similaire se présente. La principale contribution de cette thèse porte sur la conception du système *Amadeus* dont l'objectif est de répondre à ces différentes contraintes de façon radicalement opposée aux systèmes existants, par une approche coopérative et locale à chaque dispositif. Ainsi, notre approche propose de distribuer et d'intégrer une instance d'*Amadeus* (qui est un AMAS) à chaque dispositif

composant le système ambiant. Chaque instance est en charge d'apprendre et de mettre en oeuvre localement le bon comportement à attribuer au dispositif auquel elle est associée en fonction des actions de l'utilisateur. Chaque instance est aussi responsable de la prise en compte de l'apparition et/ou la disparition de nouveaux dispositifs, ainsi que des changements de comportement des utilisateurs. *Amadeus* a donc été conçu comme un AMAS composé d'AMAS dont les capacités d'observation de l'environnement, de prise en compte des feedbacks de l'environnement, d'ajustement mutuel, et d'auto-organisation permettent aux dispositifs d'adapter leur comportement (état) aux actions de l'utilisateur sans requérir l'application d'un algorithme d'apprentissage classique. Enfin, ces mêmes capacités nous permettent de mettre en œuvre un mécanisme de filtrage des données inutiles pour déterminer le comportement adéquat de chaque dispositif.

Les résultats obtenus suite à l'évaluation d'*Amadeus* dans le cadre de différents scénarios ont permis de mettre en évidence (i) sa généricité, les données de l'environnement étant utilisées indépendamment de toute sémantique associée ; (ii) son ouverture en gérant l'apparition ou de la disparition de dispositifs en cours de fonctionnement et (iii) sa proactivité par la réalisation d'actions à la place de l'utilisateur. En particulier, nous avons pu évaluer la capacité d'*Amadeus* a apprendre localement le comportement correct à attribuer à un dispositif par l'observation des actions de l'utilisateur. Cet apprentissage, couplé au mécanisme de filtrage des données inutiles, est réalisé en continu, et s'adapte aux changements survenant dans l'environnement et/ou au niveau des comportements des utilisateurs.

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31062 TOULOUSE cedex 4

Remerciements

Je n'aurais jamais cru que ça ne serait qu'à quelques heures de la soutenance que j'aurais enfin fini d'écrire cette partie. Mais j'ai bien conscience que ces quelques lignes seront certainement les plus lues (du moins jusqu'à ce que je sois reconnu comme éminent chercheur et que ce manuscrit devienne l'incontournable référence à citer), donc forcément cela met la pression... Voici donc la liste des nombreuses personnes qui ont contribué, d'une façon ou d'une autre, à l'achèvement de ma thèse.

Merci tout d'abord à mes rapporteurs Amal El Fallah Seghrouchni et Juan Pavon, respectivement professeur à l'Université Pierre et Marie Curie et professeur à l'Université de Madrid, pour leur évaluation de mon manuscrit et pour leurs retours pertinents, ainsi qu'à Laurent Vercouteur, professeur à l'INSA de Rouen, d'avoir accepté de venir évaluer mon travail en tant que membre de mon jury.

Merci à Pierre Glize, Ingénieur CNRS HDR rattaché à l'Irit. Plus qu'un simple directeur de thèse, tu as été pour moi comme pour beaucoup d'autres doctorants le Gourou nous guidant sur la voie de la coopération. Je t'adresse mes sincères et chaleureux remerciement pour tout ce que tu m'as apporté. Notons néanmoins que ces remerciements sont soumis à conditions, et deviendront *caduc* si tu me poses des questions à ma soutenance !

Merci à Valérie Camps, Maître de Conférences à l'Université Paul Sabatier, et à André Péninou, Maître de Conférences à l'IUT de Blagnac, qui m'ont encadré durant toute ma thèse. A Valérie : je t'ai vu me faire les gros yeux un bon paquet de fois, mais même si du coup tu me faisais un peu peur, j'ai adoré travailler avec toi, merci pour ta rigueur qui m'a été d'un grand secours ! A André : tu t'es retrouvé à découvrir les AMAS en même temps que moi, mais cela t'as permis d'apporter un regard neuf tout le long de ma thèse, merci pour ton aide et tes encouragements !

Merci à Marie-Pierre Gleizes, professeur à l'Université Paul Sabatier, responsable de l'équipe SMAC, et membre de mon jury de thèse. Je me suis rendu compte en lisant les remerciements de Jérémy qu'il disait déjà très bien ce que je comptais dire... Du coup, plutôt que de te resservir une pâle imitation, c'est vis-à-vis de ton rôle de prof que je vais te remercier, pour m'avoir fait découvrir les multi-agents en M1 : sans ça, je serais probablement allé dans un autre parcours... Ouf ! !

Merci de façon plus générale à tous mes collègues des équipes SMAC et SIG qui m'ont accompagné durant ces 3 ans (et quelques...), qu'ils soient permanents, doctorants, docteurs ou même stagiaires. L'ambiance dans laquelle j'ai passé ces dernières années de travail était vraiment fantastique ! Je me permettrai notamment de remercier nommément quelques personnes, en commençant par Sylvain Lemouzy, qui a été mon collègue de bureau durant la grande majorité de ma thèse (malgré ma fuite sur la fin de rédaction...), et qui, tout grognon qu'il soit face à mes tendances sucrio-compulsives, m'a été d'une très grande aide ! Je remercie aussi particulièrement Raja qui m'a aussi énormément aidé sur tout un tas de choses, parfois moins professionnelles, mais tout autant importantes à mes yeux (vous aurez remarqué une ou deux touches significatives de son aide durant mon pot, je pense !). Je remercie aussi Charles et Jérémy (oui, même lui !) pour leur accueil presque chaleureux durant ma période de rédaction. Je remercie ensuite Vic de s'être déplacé à MA soutenance

de thèse, plutôt qu'à celle de Jérémy. Remerciement spécial à Papa Simon pour ses contributions mathématiques d'une grande "trivialité". Et enfin, de façon plus brève, pour m'avoir accompagné tout le temps de ma thèse, je remercie Luc, François, Nono, Nico, Tom, Seb, Teddy, Alex, Julien (j'ai une idée !), l'autre Sylvain, Elsy, Arnaud, Bob, Sameh, Ana-Maria, Greg, Madalina et Dana, ainsi que les anciens stagiaires Jérémy, Lisa et Charlie (qui, en dépit d'un manque d'équilibre flagrant, a quand même fait du très bon boulot !).

Sans exclure les personnes sus-citées, je remercie à présent de façon très large tous mes amis. Quiconque me connait sait l'importance que j'accorde à ma vie sociale, et la fierté que je tire de vous avoir tous autour de moi. Et au risque de faire du favoritisme, je vais citer en premier lieu ceux que j'appelle communément les Palois, ceux qui me suivent depuis mes débuts à la fac et avec qui j'ai la chance d'avoir gardé contact. Quand je suis arrivé tout jeune et frais en première année, j'étais le garçon tout timide et discret que vous avez pu rencontrer... Aujourd'hui, si je suis comme je suis, c'est en très grande partie grâce à vous. Du coup, bah... faut assumer maintenant ! ! J'adresse donc mes remerciements les plus sincères à Laetitia, à mon pti Meumeu, à Dédé (yesh gros, taktak, lève les bras !), à mon coéquipier d'organisation Flo, à Drux, à Céline, aux Blonds, aux Sebs, à Carine, à Fabien, à Alex (même si j'ai un mail qui dit qu'il est plus Palois), aux matheuses Caro, Xelle, Steph et Thomas, et enfin à Adeli... Ad... Ade... non, j'y arrive pas... Bref, merci Bloubi ! !

Pour tous les autres, je sais que si j'essaie de nommer tout le monde, je vais en oublier ! Du coup, je vais jouer les lâches et ne pas le faire, puissiez-vous un jour me pardonner cette faiblesse... Je vais juste faire une petite exception, et remercier nommément Inigo, mon ancien coloc, pour tout son tact, sa subtilité, sa diplomatie, son langage chaste et toujours modéré, bref pour toutes ses nombreuses qualités qui en ont fait un excellent coloc pendant plus de deux ans ! Et pour tous les autres... je me contenterai d'un simple mais sincère merci !

C'est à présent à mes actuels colocs que j'adresse un petit message, eux qui ont vécu mes derniers mois de thèse (notamment l'intense période de rédaction). Pour Arcady, le mauvais coloc : tu vois un mauvais coloc ? Bah pareil ! Merci de m'avoir rappelé ces 11 derniers jours à quel point le temps qui me séparait de l'instant fatidique s'amenuisait, c'était certes stressant, mais aussi superbement bien présenté ! A Arthur, le bon coloc, je tenais à te dire en des termes simples que miaou miaou, miaou miaou miaou miaou, miaou miaou miaou miaou. Je sais que je ne mâche pas mes mots, mais cela vient du fond du coeur !

Et enfin, j'adresse bien sûr un remerciement général à ma famille. Je vais même me permettre une petite faiblesse émotive pour transmettre, du fond du coeur, mes plus sincères remerciements à mon Papa et à ma Maman pour tout ce que vous avez fait, tout ce que vous faites et tout ce que vous continuerez toujours à faire pour moi. Vous avez toujours sue faire face à toutes les difficultés qui pouvaient se présenter pour me permettre d'arriver là où j'en suis. Cette thèse est aussi un peu la votre... Je remercie tout autant mon frère Laurent : quand je vois à quel point tu es quelqu'un d'extraordinaire, je sais que tu es définitivement mon frère. Et enfin, je termine avec ma soeur Damiana : toutes les blagues qui me viennent à l'esprit pour personnaliser mes remerciements sont définitivement inappropriés pour un document officiel tel qu'un manuscrit de thèse... mais je sais que tu sais que je pense à ce que tu penses que je pense ! Et même plus encore !

Table des matières

Introduction

Contexte de l'étude

Ce travail de thèse s'inscrit dans le domaine des systèmes ambiants. La notion d'informatique ambiante (ou informatique ubiquitaire) a été initialement introduite par [Weiser, 1991]. Contrairement à l'informatique "conventionnelle" où l'utilisateur interagit consciemment avec un unique dispositif, l'informatique ubiquitaire s'intègre dans une structure composée de plusieurs dispositifs et fonctionne de façon invisible pour l'utilisateur ; elle s'intègre dans son environnement sans requérir son attention complète. Une telle vision de l'informatique nécessite alors trois composantes technologiques :

1. Des ordinateurs bons marchés, avec un faible coût en énergie et proposant des technologies d'interactions convenables ;
2. Un réseau connectant ces différents ordinateurs entre eux ;
3. Des logiciels d'informatique ubiquitaire.

Au cours des dernières années, nous avons pu assister à une très forte démocratisation des dispositifs électroniques. Cela concerne en premier lieu les dispositifs liés à l'informatique, avec les ordinateurs portables ou fixes, ainsi que les multiples appareils qui y sont connectés, tel que les imprimantes, les webcams, des enceintes, etc. D'autres dispositifs présents dans une habitation, fonctionnant initialement de façon indépendante, sont progressivement devenus des entités membres d'un réseau domestique : téléviseur, console de jeux, etc. La mobilité de certains dispositifs (les téléphones mobiles, en particulier) augmente les possibilités d'un tel réseau, un dispositif pouvant se retrouver connecté à différents réseaux en fonction de sa localisation. Enfin, la connexion de ces réseaux à Internet a encore agrandi ce réseau, tout dispositif électronique devenant peu à peu susceptible d'être connecté à un très vaste réseau de dispositifs. Les deux premières composantes de l'informatique ambiante proposées par [Weiser, 1991] sont donc déjà très présentes dans nos environnements quotidiens. Nous envisageons donc de contribuer dans cette thèse à la troisième composante.

Pour notre étude, nous définissons un système ambiant comme *un système constitué d'un ensemble dynamique de dispositifs physiquement distribués, capable de percevoir des données de cet environnement au travers de capteurs, et de modifier l'état de cet environnement au travers d'effecteurs*. Il est cependant fréquent que chaque dispositif présent dans un système ambiant possède un fonctionnement totalement indépendant de celui des autres dispositifs. Par conséquent, tous ces dispositifs ne sont généralement pas capables de former nativement un en-

semble dont le comportement sera cohérent et satisfaisant pour les utilisateurs qui évoluent dans ce système. Il est donc nécessaire de disposer de logiciels informatiques ubiquitaires capables d'assurer le comportement cohérent d'un système ambiant, indépendamment des constituants de ce système.

Problématique

Nous nous intéressons ici à la conception d'un système capable d'assurer une gestion cohérente d'un système ambiant dans sa globalité. Pour concevoir un tel système, la principale difficulté réside dans la complexité des systèmes ambiants. Ils peuvent en effet être composés d'un très grand nombre de dispositifs hétérogènes et physiquement distribués, et la topologie formée par ces dispositifs peut fortement évoluer dans le temps. De plus, les utilisateurs de ces dispositifs peuvent modifier au cours du temps la manière dont ils les manipulent. Envisager alors la totalité des cas d'utilisation pour l'ensemble des dispositifs existants et l'ensemble des topologies possibles à partir de ces dispositifs, d'autant plus en considérant les évolutions d'utilisation en cours de fonctionnement, ne peut clairement pas aboutir à une solution réaliste à long terme.

Une autre solution possible serait de concevoir un système qui, au lieu de ne fonctionner que dans certains cas d'utilisation préétablis, adapte son fonctionnement en fonction de son contexte. De tels systèmes sont alors décrits comme *sensibles au contexte*. Cette capacité d'adaptation en fonction du contexte implique d'être capable (i) de percevoir et de reconnaître le contexte courant, et (ii) de déterminer quel comportement adopter en fonction de ce contexte.

Notre étude de différents systèmes assurant une gestion sensible au contexte dans le cadre des systèmes ambiants a mis en évidence la nécessité que de tels systèmes soient dotés de capacités d'apprentissage, afin d'enrichir leur comportement face aux nouvelles situations susceptibles de se présenter. Cependant, les systèmes ambiants forment un domaine d'application très particulier, car ils sont très dynamiques et en interactions constantes avec des utilisateurs. Nous avons donc défini différentes contraintes qu'un processus d'apprentissage doit respecter pour être appliqué à un système ambiant :

- il doit réaliser son traitement de façon non intrusive, afin de gêner le moins possible les utilisateurs ;
- il doit réaliser son traitement en continu, afin de modifier dynamiquement le comportement appris en cas de changements dans le système ambiant, sans pour autant recommencer son traitement depuis le début ;
- il doit être capable de fonctionner en dépit de l'apparition et de la disparition de certains dispositifs appartenant au système ambiant, notamment à cause de la mobilité de certains de ces dispositifs.

Contribution

Ce travail de thèse porte sur la conception du système multi-agent adaptatif *Amadeus*, dont l'objectif est d'apprendre le comportement à attribuer à un système ambiant en fonction

du contexte. En plus des contraintes établies précédemment (concernant le processus d'apprentissage de notre système), nous avons posé certaines hypothèses concernant le fonctionnement d'*Amadeus* :

- il doit déterminer son comportement au travers d'un processus d'apprentissage sans connaissance *a priori*, qu'il s'agisse de connaissance sur le comportement qu'il cherche à apprendre, sur les données manipulées ou sur les préférences des utilisateurs. Sa seule connaissance réside dans sa croyance que les actions des utilisateurs représentent le "bon comportement" à adopter, et que dans deux situations similaires, les utilisateurs ont tendance à réaliser les mêmes actions ;
- Il doit disposer des données suffisantes à son processus d'apprentissage pour obtenir un fonctionnement minimal suffisant ;
- Afin de respecter la contrainte de non intrusivité vis-à-vis des utilisateurs, il doit réaliser son apprentissage au travers d'un processus d'observation/d'imitation. Il doit alors observer les actions récurrentes des utilisateurs afin d'apprendre à les réaliser à leur place, plutôt que de gêner les utilisateurs en tentant des actions au hasard pour en évaluer les effets. En revanche, il lui est possible de réaliser quelques actions incorrectes sans entraîner de rejet de la part des utilisateurs, du moment qu'il s'enrichit de ces erreurs pour améliorer son comportement ;
- il doit assurer la gestion d'un système ambiant sans requérir de composant central percevant l'ensemble des données issus du système ambiant, et contrôlant l'ensemble des dispositifs.

La conception d'*Amadeus* repose sur l'approche des systèmes multi-agents adaptatifs, ou approche par AMAS. Cette approche est adaptée à la conception de systèmes multi-agents dont l'objectif est de résoudre des problèmes complexes, incomplètement spécifiés, et pour lesquels il n'existe pas de solution algorithmique connue *a priori*. Nous avons alors conçu *Amadeus* comme un AMAS constitué d'agents *dispositif*, chaque agent *dispositif* étant en charge de la gestion d'un des dispositifs du système ambiant. Plus précisément, *Amadeus* a été conçu comme un AMAS d'AMAS, c'est-à-dire que chaque agent *dispositif* est lui-même un AMAS.

Enfin, l'instanciation des AMAS à ce problème nous a amené à définir trois mécanismes : le mécanisme de recherche d'une plage de valeurs AVRT [1], le mécanisme d'auto-ordonnancement de l'agent *self-ordered* et le mécanisme de filtrage de variables inutiles. Il s'agit de mécanismes facilitant la mise en place d'interactions entre agents coopératifs, réutilisables dans la conception d'autres AMAS. Ils forment donc une contribution pour la conception d'AMAS.

Organisation du manuscrit

Ce manuscrit se compose de trois parties :

- La première partie, consacrée à l'état de l'art, s'intéresse d'abord aux notions de contexte et de sensibilité de contexte. Nous présentons ensuite les propriétés d'un système de gestion sensible au contexte d'un système ambiant. Puis nous présentons

1. Adaptive Value Range Tracker

un certain nombre de systèmes répondant (au moins partiellement) à notre problématique, tout en étudiant s'ils vérifient les propriétés que nous avons établies. Enfin, nous passons en revue les principaux algorithmes d'apprentissage existants, en analysant leur pertinence vis-à-vis de notre problématique ;

- La seconde partie est dédiée à la présentation des contributions de ce travail de thèse. Après une présentation générale de l'approche par AMAS, nous montrons comment nous avons instancié cette approche pour la conception d'*Amadeus*. *Amadeus* a été conçu en suivant la méthode ADELFE ; elle commence par une phase d'étude des besoins préliminaires et finaux, puis par une phase d'analyse du système où l'adéquation de l'approche par AMAS à notre problème à été vérifiée, et où les agents d'*Amadeus* ont été définis. La phase de conception a porté sur la définition du comportement nominal des différents agents composant *Amadeus*, puis sur la définition de leur comportement coopératif. Cette partie se termine par la phase d'implémentation, puis par une présentation des différents outils et mécanismes génériques de coopération entre agents d'un AMAS que nous avons conçus et développés ;
- La troisième partie porte sur l'évaluation d'*Amadeus* et présente les résultats obtenus. Elle commence par décrire l'outil de simulation de systèmes ambiants que nous utilisons pour réaliser nos différentes études. Puis, nous évaluons les capacités d'apprentissage d'*Amadeus*. Nous étudions ensuite ses capacités d'ouverture, autrement dit son adaptation à l'apparition ou à la disparition de dispositifs dans son environnement en cours de fonctionnement ; nous étudions également son adaptation aux changements de préférences de l'utilisateur. Nous évaluons enfin *Amadeus* en présence de plusieurs utilisateurs, et terminons par une évaluation de sa capacité à filtrer les variables inutiles.

Nous terminons ce manuscrit en faisant un bilan du travail réalisé, et en présentant quelques perspectives envisageables à court et moyen terme.

Première partie

État de l'art

1 La sensibilité au contexte dans les systèmes ambiants

D'ABORD cantonnée à un déploiement sur des ordinateurs, l'informatique a fortement évolué ces dernières années avec l'essor des technologies mobiles et intégrées. L'information ne transite plus par un unique appareil sous le contrôle d'un utilisateur, mais par de nombreux appareils qui fournissent de l'information ou des services à l'utilisateur. Un système informatique est alors distribué dans l'environnement réel sous la forme de dispositifs physiques avec lesquels l'utilisateur peut interagir. Nous parlons alors de "système ambiant".

De tels systèmes se composent généralement d'un grand nombre de dispositifs hétérogènes. Ces dispositifs sont reliés entre eux et effectuent de très nombreux échanges d'information, ce qui rend difficile (voire impossible) la délimitation précise de ces systèmes ambiants. Cette délimitation est d'autant plus difficile que certains dispositifs sont susceptibles de disparaître en cours de fonctionnement, tandis que d'autres peuvent apparaître. Une gestion centralisée d'un système ambiant basée sur des règles ad-hoc impliquerait alors de déterminer un nombre bien trop important de cas d'utilisations. Dès lors, une telle solution nous semble donc inappropriée.

Au vu des caractéristiques particulières des systèmes ambiants, une des pistes envisageables pour en assurer le bon fonctionnement général est de rendre ces systèmes ambiants "sensibles au contexte" (*context-aware*). [Coutaz *et al.*, 2005] établit notamment que les défis de l'informatique ambiante à large échelle peuvent être résolus par une approche structurée et flexible du contexte. Il s'agit alors de rendre les systèmes ambiants auto-adaptables au contexte des utilisateurs, en leur attribuant des capacités de perception sur leurs environnements et de raisonnement sur les informations perçues.

Dans ce premier chapitre, nous nous intéressons aux définitions du contexte et de la sensibilité au contexte proposées dans la littérature du domaine. Nous présentons ensuite certaines propriétés qu'un système de gestion du contexte se doit de posséder pour être appliqué à un système ambiant. Enfin, nous étudions les outils proposés pour la gestion du contexte en système ambiant.

1.1 Définitions

1.1.1 Le contexte

La notion de contexte est sujette à de nombreuses études et recherches visant à en donner une définition claire et précise. Nous présentons ici une liste non exhaustive des définitions disponibles dans la littérature, ordonnées selon leur ordre chronologique.

[Schilit *et al.*, 1995] définissent le contexte comme les informations répondant aux questions "*where you are* (où es-tu)", "*who you are with* (qui est avec toi)" et "*what resources are nearby* (quels ressources sont proches de toi)". Ils considèrent que le contexte ne se limite pas à la localisation de l'humain, et incluent alors la luminosité, le niveau de bruit, l'architecture du réseau, le coût et la bande passante de communication, et même la situation sociale de l'utilisateur.

[Kokinov, 1995] propose une définition plus précise du contexte. Il commence par le définir comme étant l'ensemble de toutes les entités qui influencent le comportement cognitif de l'humain (ou du système) dans une occasion particulière. Il associe ensuite quatre propriétés au contexte :

1. **Le contexte est un "état d'esprit" :** Un humain percevant son environnement possède une représentation interne de cet environnement. Cette représentation ne contient pas la totalité des informations perçues, mais plutôt le sous-ensemble des informations qui influencent son comportement. Le contexte est alors considéré comme un "état d'esprit" du système cognitif, et c'est cette représentation interne qui influence réellement le comportement de l'humain. Le contexte d'une entité (humain ou système) est alors constitué des informations issues de la représentation interne que l'entité se fait de son environnement plutôt que de l'état réel de cet environnement.
2. **Le contexte n'a pas de frontière clairement définie :** L'auteur illustre ce fait par un exemple simple portant sur la reconnaissance de caractères. Si nous considérons l'ensemble composé des caractères suivants :

   A 13 C

   la majorité des gens reconnaîtront les caractères A, B et C. Tandis que dans l'ensemble suivant :

   9 13 7

   il est probable qu'ils reconnaissent bien les caractères 9, 13 et 7. Les mêmes symboles 13 aura donc été reconnus de façon différente selon leur contexte. Pourtant, si l'on prend les ensembles suivants :

   9 M 7 9 N 7 9 A 7 9 13 7 9 C 7

   le quatrième ensemble est cette fois lu comme un 9, un B et un 17. Les caractères 13, correctement lus dans le contexte précédent, se retrouvent donc interprétés différemment dans un contexte encore plus large. Il est donc difficile d'établir les frontières entre les données qui influencent la tâche courante (les informations contextuelles), ici la tâche de reconnaissance de caractères, et celles qui ne le font pas.
3. **Le contexte est composé par l'association d'éléments pertinents :** L'auteur définit deux critères différents pour déterminer la pertinence de l'appartenance d'un élément au

contexte. Le premier est la "pertinence causale", qui est définie par rapport à l'objectif du système. La pertinence d'un élément est liée à l'existence d'une chaîne de relations de causalité reliant cet élément avec l'objectif. Cependant, le critère de pertinence causale ne peut se faire qu'*a posteriori*, une fois l'objectif atteint, afin d'établir l'influence d'un élément sur la réalisation d'un objectif.

Le second est la "pertinence associative", qui est définie par rapport à l'ensemble des éléments formant le contexte et est mesurée par le niveau de connexion entre un élément particulier et tous les autres éléments de ce contexte. Le premier critère permet d'établir clairement si un élément appartient au contexte ou pas, alors que ce second critère (considérant qu'au final, tous les éléments du contexte sont reliés les uns aux autres) attribue plutôt un niveau d'appartenance d'un élément au contexte. Ce second critère peut s'évaluer en cours de fonctionnement, avant que l'objectif ne soit appris.

4. **Le contexte est dynamique :** Si nous modifions l'objectif ou le mode de raisonnement du système, il est évident que le contexte changera, mais ce type de changement est relativement rare en général. En revanche, il est plus fréquent que dans certains systèmes (tels que les systèmes ambiants), des changements dans son environnement se produisent, comme l'ajout, le changement ou la suppression d'un élément par exemple. De même, la perception de l'environnement par le système peut évoluer, avec par exemple comme la découverte d'un nouvel élément

Dans son étude portant sur un framework pour la création d'applications sensibles au contexte (notamment une application de création de notes contextualisées), [Brown, 1996] définit le contexte comme étant une combinaison d'éléments de l'environnement que le système connaît, comme par exemple la localisation, l'heure de la journée, la saison de l'année, la température, etc.

[Pascoe, 1998] considère le contexte comme un concept subjectif défini par l'entité qui le perçoit. Pour une entité, le contexte pourra être sa localisation spatiale, tandis que pour une autre, il s'agira de sa localisation temporelle, ou bien de son état émotionnel.

[Dey et Abowd, 2000] apportent une définition plus générale du contexte, qui est la définition la plus utilisée par les concepteurs de systèmes sensibles au contexte. Ils considèrent qu'il s'agit de l'ensemble de toutes les informations qui peuvent être utilisées pour caractériser la situation d'une entité.

Pour [Chaari *et al.*, 2005], le contexte est l'ensemble des paramètres externes à l'application pouvant influer son comportement en définissant de nouvelles vues sur ses données et ses services. Ils rajoutent que l'ensemble des paramètres du contexte peut être couvert par cinq axes : le mode de communication, l'utilisateur, le terminal, la localisation et l'environnement. Un changement de valeur sur l'un ces axes définit alors une nouvelle situation contextuelle à laquelle l'application doit s'adapter. La figure 1.1 montre une représentation de deux situations contextuelles différentes pour une même action (ici, afficher le dossier médical d'un patient) au travers de trois de ces axes. Le contexte n'est en effet pas le même si l'utilisateur est un infirmier qui regarde le dossier à l'extérieur du cabinet sur un PDA, ou s'il s'agit du médecin qui le regarde sur le PC du cabinet.

[Strassner *et al.*, 2009] conservent l'idée de contexte lié à une entité proposée par [Pascoe, 1998], en le définissant comme l'ensemble des connaissances mesurées (les faits tels qu'ils

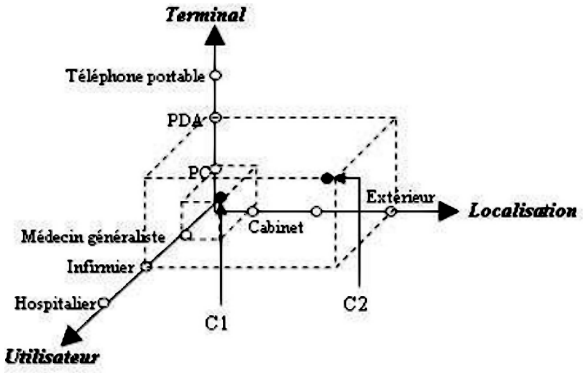

Figure 1.1 – Représentation multi-axiale du contexte selon [Chaari *et al.*, 2005]

sont perçus) ou inférés (résultant d'un apprentissage et d'un raisonnement computationnel appliqué aux contextes passés et présent) qui décrivent l'état et l'environnement dans lequel une entité existe ou a existé.

Enfin, [Brézillon, 2010] se base sur la définition de [Brezillon et Pomerol, 1999] et définit le contexte comme ce qui contraint le focus d'un acteur sans y intervenir explicitement. Il complète cette définition en précisant que "parler de contexte n'a de sens que relativement au focus" d'un acteur, que "comme le focus évolue, le contexte a une dynamique" et que "le contexte est ancré dans un domaine".

Discussion Différents aspects ressortent de ces définitions. Comme le souligne [Schmidt *et al.*, 1999], le contexte ne concerne pas uniquement la localisation de l'utilisateur, où même celle des différents éléments composant l'environnement ambiant, mais un plus large panel d'informations. La sensibilité à la localisation n'est qu'un sous-problème plus restreint de la sensibilité au contexte.

Il est possible de lister les différents types d'informations contextuelles, comme le font plusieurs des définitions énoncées précédemment. Le contexte peut alors être décrit comme les informations concernant la localisation, l'heure, le profil de l'utilisateur, la température, etc. Cependant, nous considérons une énumération explicite de tous les types de données contextuelles comme une solution limitée, car cela implique de prendre le risque d'exclure certaines données contextuelles dès lors qu'elles n'entrent pas dans une des catégories établies, ou qu'elles apparaitraient postérieurement à la conception du système. De plus, comme énoncé par [Olaru, 2011], le contexte ne se limite pas à une liste de propriétés, mais peut aussi être constitué d'associations entre différentes informations qui, prises séparément, ne sont pas incluses dans les données contextuelles. Par conséquent, une définition correcte du contexte doit être assez générale pour inclure tout type de données, comme c'est le cas pour la définition de [Dey et Abowd, 2000].

La notion de subjectivité proposée notamment par [Pascoe, 1998] et [Strassner *et al.*, 2009] nous semble tout aussi pertinente, car dans un système aussi vaste qu'un système ambiant, il est impossible de caractériser le contexte de la totalité du système. Il nous semble alors que le contexte se doit d'être défini selon le point de vue local de chacune des entités composant le système.

Enfin, le lien défini par [Brézillon, 2010] entre le focus d'une entité, autrement dit la tâche qu'elle doit réaliser, et le contexte de cette entité nous semble également essentiel.

Pour ces raisons, nous proposons de définir le contexte comme "**l'ensemble des informations extérieures à l'activité d'une entité, décrivant l'environnement tel qu'elle le perçoit et ayant un impact sur l'exécution d'une ou plusieurs tâches liées à son activité**". Une information appartenant au contexte d'une entité est alors appelée *information contextuelle*.

1.1.2 La sensibilité au contexte

Dans la section précédente, nous avons étudié différentes définitions de la notion de contexte, et en avons extrait une définition regroupant les aspects qui nous semblaient les plus pertinents pour notre étude. Nous nous intéressons à présent aux définitions proposées pour décrire la notion de "sensibilité au contexte". Il faut cependant noter que, au vu des différences entre les définitions de contexte existantes, il est inévitable que les différentes définitions de sensibilité au contexte proposées ne soient pas toutes adaptées à notre définition.

[Schilit et Theimer, 1994] ont été les premiers à introduire la notion de sensibilité au contexte, au travers de la conception d'un système d'*active map service*, proposant des informations à un utilisateur en fonction de sa localisation. Ils définissent alors la sensibilité au contexte comme la capacité des applications d'un utilisateur mobile à découvrir et à réagir aux changements survenant dans l'environnement où ils sont situés. [Schilit *et al.*, 1995] définit ensuite 4 catégories d'applications sensible au contexte, selon que la tâche à réaliser consiste uniquement à obtenir des informations ou à réaliser des actions, et que cette tâche soit réalisée manuellement ou automatiquement (voir tableau 1.1).

	Manuelle	Automatique
Information	Sélection à proximité	Reconfiguration contextuelle automatique
Commande	Commande contextuelle	Actions contextuellement déclenchées

Tableau 1.1 – Catégories d'applications sensibles au contexte d'après [Schilit *et al.*, 1995]

[Pascoe, 1998] décline la capacité d'un système à être sensible au contexte en quatre niveaux :

- La perception contextuelle qui permet uniquement à un système de capturer les informations du contexte afin de pouvoir les présenter à l'utilisateur. Le système est alors capable de *percevoir* son environnement.
- L'adaptation contextuelle qui permet aussi au système de modifier son comportement pour s'intégrer avec davantage de transparence à l'environnement de l'utilisateur. Le système est alors capable de *réagir* aux évolutions de son environnement.
- La découverte de ressources contextuelles qui consiste à découvrir et à utiliser des ressources supplémentaires auprès d'autres entités pour compléter ses connaissances sur son contexte. Le système est alors capable d'*interagir* avec son environnement.
- L'augmentation contextuelle qui permet de compléter l'environnement en lui associant des données en fonction d'un contexte donné. Le système est alors capable d'*enrichir* son environnement.

Pour [Ryan *et al.*, 1998], la sensibilité au contexte décrit la capacité d'un système à percevoir et à agir sur des informations sur son environnement, telles que la localisation, le temps, la température ou l'identité de l'utilisateur.

Pour [Dey et Abowd, 2000], une application est sensible au contexte si elle est capable de percevoir son contexte et si elle possède un comportement dynamique guidé par sa connaissance sur son environnement.

[Lieberman et Selker, 2000] considèrent que, contrairement aux applications classiques où le système reçoit en entrée des données fournies explicitement par l'humain (*input*) et se base uniquement sur ces entrées pour produire une sortie explicite (*output*), une application sensible au contexte tient compte de données qui ne lui sont pas fournies explicitement (autrement dit, des données qu'il perçoit ou récupère par lui-même). Dans une telle vision de la sensibilité au contexte, toute donnée n'appartenant pas aux entrées ou aux sorties explicites d'une application mais influençant néanmoins son traitement appartient alors à son contexte.

Une autre définition proposée par [Korkea-Aho, 2000] établit qu'un système est sensible au contexte s'il peut extraire, interpréter et utiliser des informations de contexte, et adapter sa fonctionnalité au contexte d'utilisation courant.

Pour [Rohn, 2003], un système sensible au contexte diffère d'un système classique car il est :

- adaptatif : l'auteur définit cette propriété comme la capacité à apprendre les préférences de l'utilisateur et à s'ajuster en conséquence ;
- réceptif : il anticipe les besoins de l'utilisateur dans un environnement évolutif ;
- un système proactif : il possède un objectif et est capable de prendre des initiatives, plutôt que de simplement réagir à l'environnement ;
- un système autonome : il peut agir indépendamment, sans intervention humaine.

[Barkhuus, 2003] donne une définition proche de celle de [Schilit et Theimer, 1994], en caractérisant la sensibilité au contexte comme la capacité des applications à détecter et réagir aux variables de l'environnement de façon autonome.

En se basant sur ces différentes définitions, nous pouvons remarquer que deux aspects principaux caractérisent la sensibilité au contexte. Le premier concerne la capacité à percevoir son contexte ; cette notion de perception/détection est présente dans toutes les définitions. Le second aspect porte sur la capacité à exploiter le contexte perçu. Être sensible au contexte ne consiste donc pas uniquement à percevoir son contexte, mais aussi à agir différemment en fonction de celui-ci.

Comme [Korkea-Aho, 2000], nous considérons la sensibilité au contexte d'un système comme "**la capacité à percevoir, interpréter et utiliser les différentes informations relatives au contexte courant pour adapter dynamiquement sa fonctionnalité**".

1.2 Sensibilisation au contexte dans le cadre des systèmes ambiants

Nous nous intéressons à présent aux travaux réalisés dans le but de concevoir un système capable de rendre un système ambiant sensible au contexte. Si nous reprenons notre définition (section 1.1.2), il s'agit donc de donner à un système ambiant la capacité d'adapter dynamiquement sa fonctionnalité selon son contexte courant.

[Coutaz *et al.*, 2005] proposent un modèle général d'un système sensible au contexte. Ce modèle est représenté en figure 1.2, illustrant les différents niveaux de son infrastructure, de la récupération des données contextuelles à leur exploitation.

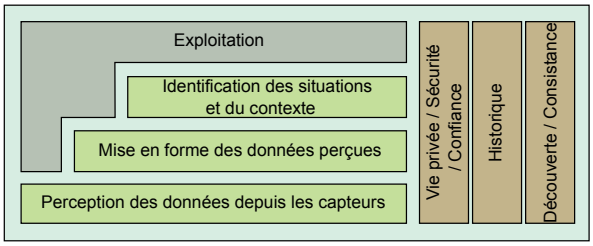

Figure 1.2 – Modèle générale d'un système sensible au contexte selon [Coutaz *et al.*, 2005]

Le premier niveau (*sensing*) concerne l'acquisition de données depuis des capteurs. Le second niveau (*perception*) traite de la mise en forme des données capturées en un modèle de contexte, indépendamment de la méthode de capture utilisée par la couche inférieure. Le troisième niveau porte sur l'*identification* de la situation contextuelle courante, tandis que le quatrième niveau se rapporte à l'"*exploitation* du contexte. Chacun de ces niveaux doit être associé à des mécanismes pour supporter les problèmes de vie privée, de gestion d'historique et de découverte.

Rendre un environnement ambiant sensible au contexte implique donc d'y intégrer un système sensible au contexte gérant ces quatre niveaux. Il s'agit alors d'étudier les propriétés requises pour un tel système, puis d'étudier les solutions (partielles ou complètes) proposées par la littérature.

1.2.1 Propriétés requises pour la sensibilisation au contexte en environnement ambiant

Les systèmes ambiants possèdent des caractéristiques particulières qui font de leur sensibilisation au contexte un problème complexe : ils sont composés de dispositifs distribués et hétérogènes, certains de ces dispositifs pouvant apparaître et disparaître en cours de fonctionnement, et reliés entre eux par de nombreuses interconnexions. De plus, les nombreux utilisateurs évoluant dans ces systèmes ambiants sont eux-mêmes source de dynamique : leurs préférences (et par conséquent leurs comportements) peuvent évoluer au cours du temps, et il peut leur arriver de réaliser des actions incohérentes.

Pour permettre à un système ambiant d'être sensible au contexte, certaines propriétés doivent être respectées. Plusieurs propriétés ont été proposées dans d'autres études([Dey

et al., 2001],[Gustavsen, 2002],[Hofer *et al.*, 2003],[Euzenat *et al.*, 2008]), nous présentons ici celles qui nous ont semblé les plus pertinentes :

- **Généricité** : La représentation des données contextuelles peut être très diverse, allant d'un simple doublet <nom – valeur> jusqu'à une ontologie complète. Un système est dit générique si sa représentation des données contextuelles lui permet d'intégrer n'importe quel type de données ;
- **Expressivité** : Une donnée contextuelle ne se limite pas à une valeur, elle est aussi associée à une sémantique. En fonction de la modélisation du contexte utilisée, il est possible de disposer d'une sémantique de données plus ou moins riche ;
- **Distribution** : Un système distribué est physiquement et/ou logiquement décomposé en différents processus fonctionnant sans requérir la présence d'un processus central (un serveur, par exemple), et limitant leurs échanges aux données nécessaires et suffisantes à leur fonctionnement ;
- **Ouverture** : Un système est ouvert s'il peut intégrer et exploiter de nouvelles données en cours de fonctionnement sans pour autant nécessiter d'être (re)configuré et/ou de recommencer la totalité de ses traitements. Il s'agit aussi de pouvoir supporter la disparition de données connues en cours d'exécution, parfois juste temporairement ;
- **Proactivité** : La proactivité d'un système désigne sa capacité à anticiper et à prendre des décisions concernant une situation, et à agir en conséquence. Un système proactif ne s'arrête donc pas à la perception et à l'interprétation de la situation dans laquelle il se trouve ; il est aussi capable d'agir pour améliorer la situation courante au lieu d'attendre qu'un utilisateur le fasse par lui-même ;
- **Confidentialité** : La confidentialité permet de garantir à l'utilisateur que ses données contextuelles privées ne sont pas divulguées à des tiers et qu'il en maîtrise le contenu et l'usage ;
- **Explicabilité** : L'explicabilité d'un système concerne sa capacité à donner, en continu ou en cas de demande d'un utilisateur, les faits et le raisonnement associé qui expliquent son comportement.

Pour évaluer les différents systèmes de sensibilisation au contexte décrits dans la section suivante, nous nous basons sur des critères issus de ces propriétés. Chaque système est alors évalué selon qu'il respecte parfaitement (++), significativement (+), légèrement (-) ou pas du tout (–) chacun des critères. Le tableau 1.2 décrit de façon plus précise ce que représente le niveau de respect (ou non respect) des différents critères.

1.2.2 Systèmes existants pour la sensibilisation au contexte

La littérature propose un certain nombre de systèmes permettant de rendre un environnement ambiant sensible au contexte. Parmi eux, certains appartiennent à la catégorie des gestionnaires de contexte. Un gestionnaire de contexte est défini par [Rottenberg *et al.*, 2012] comme une "entité logicielle responsable de la collecte, de la gestion (traitement et filtrage) et de la présentation des informations de contexte aux applications". Il n'a donc pas pour objectif de répondre aux sept critères précédemment énoncés, leur fonctionnement se limitant aux aspects récupération et présentation des données contextuelles plutôt qu'aux aspects utilisation de ces données. Il est cependant intéressant de présenter dans cette étude

	Gradation du critère			
	--	-	+	++
Généricité (Intégration d'un nouveau type de donnée)	Le concepteur doit intégrer explicitement tout nouveau type de données et reconsidérer une partie de la conception du système.	Le concepteur doit intégrer explicitement tout nouveau type de données et reconsidérer légèrement le fonctionnement du système.	Le concepteur doit intégrer explicitement tout nouveau type de données, mais aucune reconfiguration du fonctionnement du système n'est exigée.	Intégration directe ou automatisée d'un nouveau type de donnée sans intervention ni reconfiguration du fonctionnement du système par le concepteur.
Expressivité (sémantique associée aux données traitées)	Une donnée est uniquement limitée au doublet <nom - valeur>.	Sémantique ad-hoc, ne permettant pas de réel traitement sur les données.	Le niveau de sémantique permet des traitements limités sur ces données (filtrage de données inutiles).	Des traitements élaborés sur ces données sont possibles (gestion des incohérences, déduction de nouvelles données, etc.).
Distribution des traitements	Tout est centralisé sur un serveur.	Un serveur central est nécessaire pour certaines opérations (l'enregistrement des données perçues, le listing des dispositifs existants, etc.).	Il n'y a pas de serveur central, mais les échanges de données ne se limitent pas aux données requises par chaque processus local (envoie de données inutiles).	Chaque processus ne reçoit et ne diffuse que ce qui est strictement nécessaire.
Ouverture par intégration de nouveaux dispositifs	Impose une reconfiguration explicite par le concepteur et l'arrêt du système.	L'utilisateur peut effectuer la reconfiguration, mais en réinitialisant le système.	L'intégration en cours de fonctionnement par l'utilisateur est limitée à certains dispositifs.	Tout dispositif peut être ajouté/supprimé par l'utilisateur sans arrêter le système.
Proactivité	Le système limite sa sensibilité au contexte à la présentation passive des données contextuelles, sans adapter son comportement aux données perçues.	Le système présente les données contextuelles de façon active, en fonction des besoins des consommateurs de ces données.	Le système exploite les données contextuelles perçues pour réaliser des actions et/ou des propositions d'actions, conformément au comportement qu'il lui est attribué par le concepteur ou l'utilisateur.	Le système exploite les données contextuelles perçues pour réaliser des actions et/ou des propositions d'actions, en s'adaptant en cours de fonctionnement en fonction des nouveaux contextes rencontrés.
Confidentialité	L'aspect confidentialité des données n'est absolument pas géré par ce système.	Le système propose quelques mécanismes de base pour gérer l'aspect confidentialité des données.	Le système propose un système dont l'aspect confidentialité des données est suffisamment géré pour le rendre applicable aux systèmes ambiants.	L'accès confidentialité des données contextuelles est lui-même sensible au contexte.
Explicabilité	Aucune explication du comportement du système ne peut être présentée à l'utilisateur.	Il est possible d'interpréter *a posteriori* le comportement du système, mais ce n'est pas une fonctionnalité de base du système.	Le système propose *a posteriori* des explications sur son comportement.	Le système propose en cours de fonctionnement des explications sur son comportement.

Tableau 1.2 – Description des différents niveaux de respects vis-à-vis des critères

les différents gestionnaires de contexte applicables aux environnements ambiants, afin de voir les possibilités de couplages de tels outils avec d'autres systèmes focalisés sur les autres aspects.

Nous allons à présent étudier plusieurs systèmes existants pour la sensibilisation au contexte en système ambiant (incluant des gestionnaires de contexte) selon les sept critères précédemment énoncés. Ces différents systèmes sont présentés dans un ordre chronologique.

1.2.2.1 Intelligent Classroom

Présentation [Franklin et Flaschbart, 1998] proposent l'idée d'*Intelligent Environment* comme un environnement capable d'exploiter les données contextuelles perçues afin d'adapter son fonctionnement. Ils exploitent plus particulièrement cette idée en proposant l'*Intelligent Classroom*, un système de gestion d'une classe de cours capable d'adapter le fonctionnement des effecteurs de la classe (lampes, grand écran, etc.) en fonction de l'activité de l'enseignant.

Dans ce système, le concepteur dispose d'un certain nombre d'algorithmes pour interpréter l'activité de l'utilisateur, c'est-à-dire l'enseignant. Il peut alors utiliser un langage déclaratif spécifique au système pour définir un plan d'actions à réaliser en fonction du contexte dans lequel se situe l'enseignant. Ce langage permet alors de donner un comportement précis au système afin d'accompagner correctement l'enseignant dans ses activités.

Évaluation La majorité des critères recherchés n'est pas respectée ici : il s'agit d'un système centralisé ne permettant pas d'ajouter de nouveaux dispositifs en cours de fonctionnement, n'utilisant que des données non génériques avec une sémantique spécifique et ne proposant pas de traitements liés à l'aspect confidentialité des données. En revanche, il propose une première étape concernant l'aspect adaptation. En effet, en se basant sur des règles préétablies par le concepteur, il adapte l'environnement ambiant à l'activité de l'utilisateur. Cette adaptation se limite cependant aux situations et actions initialement prévues par le concepteur, et le système dans son état actuel ne propose pas de justification du choix de ses actions.

1.2.2.2 ACHE

Présentation Le système ACHE proposé par [Mozer et Miller, 1998] est un système de contrôle de la luminosité des pièces d'une maison en fonction des activités et des préférences des utilisateurs. Ce système a été conçu dans le cadre du projet Adaptive House de [Mozer *et al.*, 1995].

L'efficacité du contrôle de la luminosité réalisé par le système ACHE dépend de deux critères : la satisfaction de l'utilisateur et l'économie en termes de coût énergétique. Le fonctionnement du système intègre pour cela un mécanisme d'apprentissage par renforcement (voir section 2.2.2) intégrant en priorité les actions de l'utilisateur dans son comportement, mais réalisant 5% du temps une autre action que celle réalisée précédemment par l'utilisateur, afin de tendre vers un comportement plus satisfaisant en coût énergétique que celui proposé par l'utilisateur. Ces tentatives d'améliorations continuent progressivement jusqu'à ce que l'utilisateur le contredise, auquel cas le système sait qu'il a atteint la limite de la satisfaction simultanée de ces deux critères. La nouvelle action choisie est celle que le système croit la moins couteuse en terme de consommation d'énergie, cette croyance se basant sur des évaluations de coût d'actions données *a priori* par le concepteur.

Évaluation Les données exploitées ici se limitent à des données temporelles, de localisation (en position ou en zone), en état de lampes et en luminosité. La modélisation utilisée alors est très ad-hoc (modélisation de types de données préétablies, tels que la luminosité, la température, le bruit, etc.), mais permet certains traitements tel que l'observation des effets d'une lampe sur la luminosité d'une pièce, ou l'estimation de la luminosité d'une pièce si le système éteignait la lampe. En revanche, la généricité est limitée. Aucune information n'est donnée sur l'aspect ouverture, le système étant appliqué à une maison déjà conçue. D'après le fonctionnement du système, il semble possible d'intégrer de nouveaux dispositifs, mais cela impliquerait de réinitialiser le système, notamment l'apprentissage réalisé. L'aspect confidentialité n'est pas géré, tout comme la distribution du système, celui-ci étant

intégralement centralisé. Enfin, bien qu' ne propose pas d'explications sur son comportement, le système ACHE est capable de réaliser des actions sur son environnement en se basant sur un mécanisme d'apprentissage basé sur le Q-Learning (voir chapitre 2.2.2) lui permettant d'améliorer son comportement en cours de fonctionnement.

1.2.2.3 Context Toolkit

Présentation Le *Context Toolkit* de [Dey *et al.*, 2001] est une des premières architectures de gestion du contexte proposée dans la littérature. Le schéma 1.3 illustre son fonctionnement.

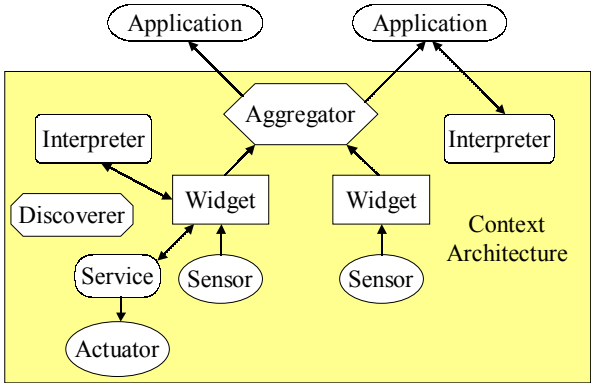

Figure 1.3 – Le Context Toolkit [Dey *et al.*, 2001]

Le Context Toolkit propose un ensemble de cinq composants : le Widget, l'Interpreter, l'Aggregator, le Service et le Discoverer. Ces composants forment l'architecture du gestionnaire de contexte. À cette architecture sont connectées une ou plusieurs applications qui, grâce à leurs interactions avec le Context Toolkit, deviennent sensibles au contexte.

- **Widget** : il sert d'interface entre un capteur et l'infrastructure du système. Il masque les détails sous-jacents concernant la manière dont une donnée est recueillie ; il ne présente ainsi que la donnée elle-même, à moins qu'une application ne demande explicitement des détails sur le capteur utilisé. Une donnée catégorisée en tant que donnée de localisation est ainsi présentée de la même façon quelle que soit la technologie utilisée par le capteur. Le Widget est donc responsable de l'encapsulation et de la transmission de sa donnée en cas de changement de valeur.
- **Interpreter** : il permet l'interprétation des données perçues par les Widgets, en transformant les informations bas niveau issues des capteurs en informations de plus haut niveau. Ce traitement peut alors être vu comme la génération de nouvelles données à partir des données existantes.
- **Aggregator** : il gère la centralisation des données contextuelles. Ainsi, une application n'a pas besoin de se connecter à l'ensemble des Widgets qui lui fournissent des données utiles, mais uniquement à l'Aggregator.
- **Service** : il peut être vu comme le pendant du Widget : ce dernier perçoit des données depuis l'environnement (*input*), tandis qu'un Service est capable d'agir sur l'environnement (*output*). Ainsi, lorsqu'une action à effectuer sur l'environnement est commune à plusieurs applications, il est possible de créer un Service lié à cette action afin de capitaliser cette compétence ; les applications n'ont plus alors à se soucier de la façon

dont cette action est réalisée, il leur suffit de commander cette action auprès du Service associé.
- **Discoverer** : ce composant connaît l'ensemble des composants de l'architecture, ainsi que toutes les informations les concernant : l'adresse de chaque Widget et la donnée qu'il fournit, l'action que propose chaque Service, etc. A son lancement, chaque composant commence par notifier sa présence au Discoverer. Celui-ci a aussi pour objectif de déterminer si un composant disparaît.

La modélisation des informations contextuelles se fait grâce à un modèle de type <nom – valeur> (<*key - value*>). Chaque donnée contextuelle est donc associée à un attribut ; par exemple, un contexte possible est {LumièreSalon = "`éteinte`", LumièreChambre ="`allumée`", LocalisationUtilisateur = "`chambre`", etc.}.

Évaluation La possibilité d'ajouts de nouveaux senseurs, en leur associant des Widgets supplémentaires qui s'abonnent automatiquement au Discoverer, ainsi que les capacités du Discoverer à détecter la disparition de Widget en cours de fonctionnement, rendent ce système d'être ouvert. La modélisation des informations contextuelles dans le serveur se fait grâce à un modèle de type <clé - valeur>, ce qui limite fortement l'expressivité des données contextuelles. Aucune contrainte spécifique dans la modélisation du contexte n'interdit l'ajout d'un nouveau type de donnée contextuelle ; le critère de généricité est donc respecté. La distribution est en revanche très limitée, étant donnée la nécessité d'utiliser un serveur central pour rendre le contexte accessible aux applications. Nous ne pouvons pas le qualifier de proactif, car il a été conçu spécifiquement pour gérer l'agrégation et à la présentation de données contextuelles. Il n'explique pas non plus ses traitements. Cependant, bien que n'intégrant pas nativement de mécanismes pour gérer la confidentialité des données, le Context Toolkit a été amélioré par [Covington *et al.*, 2001] pour "contextualiser" l'accès aux données au travers du concept de rôles, et ainsi de respecter l'aspect confidentialité.

1.2.2.4 Hydrogen

Présentation Le gestionnaire de contexte *Hydrogen*, proposé par [Hofer *et al.*, 2003], a été conçu pour être utilisé sur des mobiles, et doit donc répondre à certaines contraintes comme la légèreté, l'extensibilité ou la robustesse. L'objectif ici est le même que le *Context Toolkit* : permettre à des applications de percevoir des informations appartenant au contexte.

Hydrogen est un système conçu spécialement pour la gestion de contexte pour des appareils mobiles. Comme représenté dans la figure 1.4, chaque élément du système ambiant supporte une architecture à trois niveaux : le niveau Application qui fait office d'interface avec les applications du mobile, le niveau Management qui s'occupe de la gestion des données contextuelles et le niveau Adaptor qui fait office d'interface avec les capteurs. Un serveur de contexte (*ContextServer*) est inclus au niveau "management" de chaque mobile. Au minimum, ce serveur contient les données extraites des capteurs locaux. Mais si un autre mobile est à proximité, alors les deux mobiles peuvent se connecter (via Bluetooth par exemple) et partager leurs contextes le temps de la connexion. Une information contextuelle est ici modélisée sous forme d'une classe ContextObject, (figure 1.4). Nativement, le système propose cinq classes spécifiques (le TimeContext, le LocationContext, le DeviceContext, le

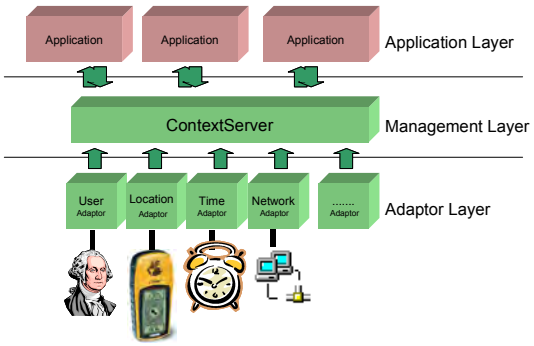

Figure 1.4 – Architecture du système *Hydrogen* de [Hofer *et al.*, 2003]

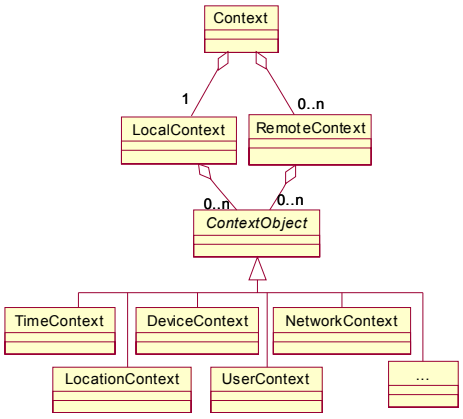

Figure 1.5 – Diagramme de classe du contexte dans le système *Hydrogen* de [Hofer *et al.*, 2003]

UserContext et le NetworkContext), mais il est possible de rajouter de nouvelles classes en spécialisation la classe ContextObject. Une donnée de contexte peut être un LocalContext si elle provient d'un capteur local ou un RemoteContext si elle provient d'une autre instance d'*Hydrogen* fonctionnant sur un autre dispositif.

Évaluation L'ouverture du système est bien proposée, car il est possible d'ajouter un nouveau dispositif dans l'environnement en cours de fonctionnement, en lui associant juste une nouvelle instance d'*Hydrogen*. Il en est de même pour l'aspect généricité, grâce à la possibilité d'ajouter des nouveaux types de données contextuelles, bien que cela nécessite une intervention explicite du concepteur. L'aspect proactivité n'est en revanche pas abordé, tout comme l'aspect explicabilité. L'expressivité du modèle utilisé est assez limitée : elle permet de distinguer différentes catégories de données contextuelles mais ne proposent pas de traitements supplémentaires exploitant cette catégorisation. La confidentialité des données est brièvement abordée, en précisant qu'il est possible pour un utilisateur de spécifier quelles données sont partageables et quelles données sont confidentielles. Enfin, concernant la distribution, chaque dispositif possède sa propre instance d'*Hydrogen*, et ces différentes instances peuvent s'échanger leurs données. Le fait que chaque dispositif dispose d'un serveur permet un respect partiel de la distribution ; en effet, l'application n'a pas à se connecter à un serveur distant pour obtenir les données contextuelles demandées. Cependant, toutes les données perçues sont considérées comme des données contextuelles, sans tenir compte de leur pertinence par rapport à la tâche effectuée par l'entité. Il est donc possible de retrouver une donnée perçue mais inutile, ou de ne pas connaître une donnée qui existe quelque part dans le système mais qui n'est pas perçue directement. La distribution est donc existante, mais nous semble peu pertinente.

1.2.2.5 iDorm

Présentation Le système iDorm proposé par [Hagras *et al.*, 2004] permet d'assurer le contrôle d'un espace ambiant au travers d'un ensemble d'agents dits "intelligents". Ce sys-

tème est appliqué à une chambre à titre d'exemple. L'ensemble de la chambre, dispositifs inclus, est modélisé au travers d'une interface de réalité virtuelle, dans laquelle sont intégrées les données perçues, afin d'être visible par l'utilisateur. Le fonctionnement du système commence, dans une première phase, par observer le comportement de l'utilisateur dans la chambre, et extraire de ses observations un comportement approprié à donner aux différents dispositifs de la chambre. Cet apprentissage est réalisé grâce à un algorithme basé sur la logique floue. Puis dans une seconde phase, le système contrôle le système ambiant à la place de l'utilisateur. Si le système est contredit par l'utilisateur, il repasse dans la première phase pour recommencer à observer l'utilisateur, et ainsi compléter son apprentissage.

Évaluation Les données perceptibles et modifiables sont prédéterminées dans le système, ne serait-ce que pour assurer un rendu correct de ces données au niveau de la modélisation virtuelle présentée à l'utilisateur ; la généricité des données n'est donc pas assurée. De même, l'ajout ou la suppression d'un dispositif dans l'environnement requiert une intervention de l'utilisateur pour intégrer ce même dispositif au niveau du modèle virtuel, et implique une réinitialisation du système pour être prise en compte par l'algorithme d'apprentissage. L'expressivité des données est ad-hoc ; elle permet une bonne représentation du modèle virtuel mais ne permet pas de traitements supplémentaires. La distribution est juste partielle, car malgré une distribution de certains traitements au niveau des agents présents dans l'environnement, un serveur central reste nécessaire. La gestion de la confidentialité des données n'est pas abordée. La proactivité est bien présente : le système se base sur ses observations pour agir sur l'environnement afin de satisfaire l'utilisateur. De plus, le comportement du système n'est pas basé sur des règles préétablies, mais sur l'apprentissage du comportement de l'utilisateur, cet apprentissage pouvant ensuite s'enrichir et se corriger en cours de fonctionnement, grâce à l'observation des retours de l'utilisateur et l'intégration de ces retours en tant que nouvelle règle. Bien qu'un traitement *a posteriori* sur les règles générées permettrait d'interpréter le fonctionnement du système, l'expressivité n'est pas une fonctionnalité proposée par le système.

1.2.2.6 CoBrA

Présentation Le système *CoBrA* (COntext BRoker Architecture) proposé par [Chen *et al.*, 2005] a un objectif plus spécifique que les systèmes précédents : il est orienté vers la gestion de contexte en espace restreint (notamment les salles de réunion) plutôt que vers les environnements ambiants en général. L'objectif de ce système est d'associer un Context Broker à un espace physique, celui-ci ayant pour objectif de modéliser les données contextuelles de cet espace, et de les rendre ainsi accessibles et utilisables pour des Personal Agents associés à chaque utilisateur. Ces agents sont conçus de façon ad-hoc, et n'appartiennent pas au fonctionnement de CoBra. Concernant les données contextuelles, elles sont modélisées grâce à l'ontologie COBRA-ONT qui, dans sa version 0.3, permet de définir 88 classes et 125 propriétés différentes.

La figure 1.6 permet de visualiser les quatre fonctionnalités d'un Context Broker. Les fonctionnalités du Context Acquisition Module et du Context Knowledge Base sont respectivement équivalentes aux fonctionnalités proposées par le Widget et par l'Aggregator

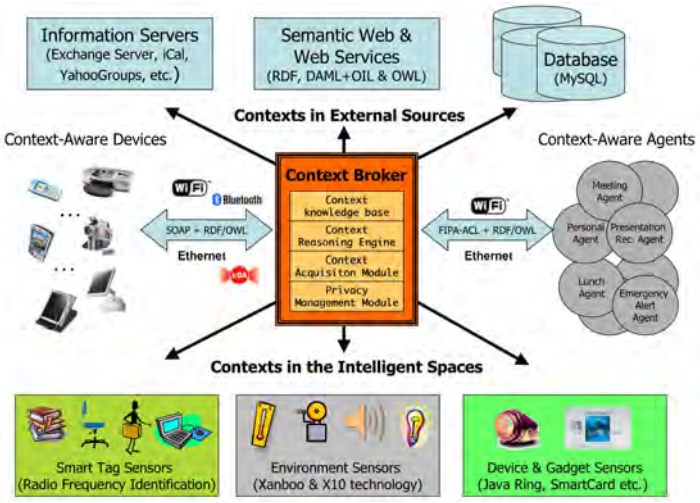

Figure 1.6 – Le système *CoBrA* de [Chen *et al.*, 2005]

Context Toolkit de [Dey *et al.*, 2001] (à savoir l'acquisition et le partage de données contextuelles). En Les deux autres fonctionnalités sont :

- Le Context Reasoning Engine qui permet au Context Broker d'appliquer des processus de raisonnement à partir du modèle du contexte. Il est notamment capable d'utiliser l'ontologie COBRA-ONT pour déduire des connaissances supplémentaires sur le contexte. Par exemple, si une personne se trouve dans une salle et qu'une réunion a lieu dans cette salle, il peut déduire que cette personne est en réunion, et sera probablement indisponible jusqu'à la fin de cette réunion. Le Context Reasoning Engine peut aussi utiliser un certain nombre d'heuristiques associées à l'espace dont est responsable le Context Broker pour détecter des inconsistances dans le modèle du contexte. Par exemple, si une personne est détectée à deux positions distinctes du fait d'un capteur défectueux, une heuristique informant qu'une personne ne peut être à deux endroits différents à un même moment permet au Context Reasoning Engine de détecter une inconsistance dans le modèle du contexte (les auteurs ne donnent pas de précisions sur le processus de résolution de ces inconsistances).
- Le Privacy Management Module gère la restriction d'accès aux données de l'utilisateur. En utilisant le langage Rei, conçu par [Kagal *et al.*, 2003], il est possible d'établir des concepts ontologiques pour modéliser les droits d'accès (interdiction, conditions nécessaires, etc.). Il est aussi possible de spécifier des règles d'inférences liant plusieurs données. Par exemple, s'il est spécifié qu'en connaissant le numéro de téléphone d'un utilisateur, il est possible de connaître son adresse alors que l'accès à cette donnée est refusé, le Privacy Management Module en déduira que l'accès au numéro de téléphone est interdit lui aussi.

Évaluation Le critère de proactivité n'est pas respecté puisque ce n'est pas le but de CoBrA. La distribution n'est pas non plus respectée ici car, bien qu'il soit possible de créer plusieurs Context Broker, il n'y a pas de communication entre eux. Nous n'avons donc pas une distribution du contexte mais plutôt un ensemble de contextes centralisés, et il revient alors

à l'agent personnel de fusionner les données provenant des différents Context Broker. Le critère d'ouverture est respecté, mes les informations utilisables se limitent à celles inclues dans le domaine établi par l'ontologie du système. L'expressivité est respectée ; l'inférence de nouvelles données est possible grâce aux ontologies associées aux données existantes. De même, les ontologies sont exploitées de façon à permettre le respect du critère de confidentialité des données. En revanche, l'utilisation d'une ontologie particulière sans qu'il ne soit possible de l'augmenter en cours de fonctionnement implique le non-respect du critère de généricité. Enfin, le Context Reasoning Engine ne propose pas nativement d'expliciter ses traitements, aussi l'explicabilité n'est pas abordée par CoBrA.

1.2.2.7 Contexteur

Présentation Le contexteur de [Rey, 2005] est une abstraction logicielle permettant de fournir des données contextuelles à des applications. La figure 1.7 illustre le modèle d'un contexteur. En entrée, il perçoit des données de contexte depuis des capteurs ou un autre contexteur, auxquels sont associées des méta-données exprimant la qualité de ces données. Ces données sont exploitées par le noyau fonctionnel, dont le comportement est paramétré préalablement par le concepteur. Ce comportement peut dépendre des informations reçues par d'autres contexteurs, ces informations étant appelées des "contrôles d'entrée". En sortie, le contexteur peut fournir les données et méta-données contextuelles issues de son traitement et à destination d'applications ou d'autres contexteurs. Il peut aussi envoyer des informations, appelées "contrôles de sortie", à d'autres contexteurs pour influencer leur comportement, ces informations devenant alors des contrôles d'entrées pour ces contexteurs. L'utilisation du contexteur permet ainsi de décomposer les processus de traitements des données contextuelles depuis les capteurs jusqu'aux applications consommatrices.

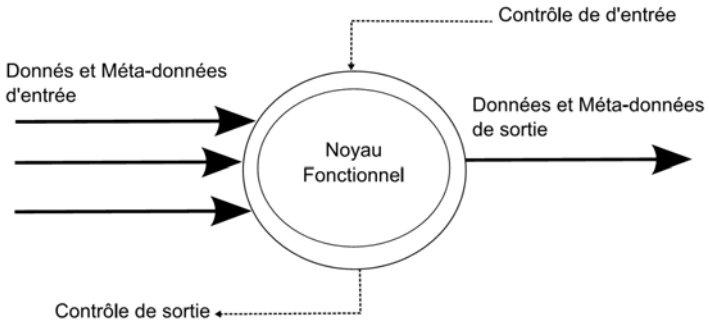

Figure 1.7 – Le contexteur [Rey, 2005]

Évaluation L'intégration d'un nouveau type de donnée requiert la création d'un nouveau contexteur capable de le gérer, afin de lui associer ses méta-données. Ces méta-données permettent d'augmenter l'expressivité de ces données, en leur associant des informations supplémentaires (précision, latence, résolution, etc.). L'ajout d'un nouveau dispositif peut se faire en cours de fonctionnement par association d'un contexteur à ce dispositif. Le critère de proactivité dépend du comportement attribué aux différents contexteurs, et rien dans le noyau central d'un contexteur ne l'oblige à expliquer son fonctionnement. Il ne propose

pas de mécanisme sur la confidentialité des données. L'approche utilisée permet une décentralisation complète du système, les contexteurs s'échangeant uniquement les informations requises.

1.2.2.8 ASK-IT

Présentation Le système ASK-IT proposé par [Spanoudakis et Moraitis, 2006] est une architecture multi-agent de gestion des données contextuelles en environnement ambiant permettant aux utilisateurs d'accéder facilement aux données et services qui les environnent. En particulier, ASK-IT porte une attention particulière aux utilisateurs soumis à des handicaps particuliers, afin que les services proposés leur soient adaptés.

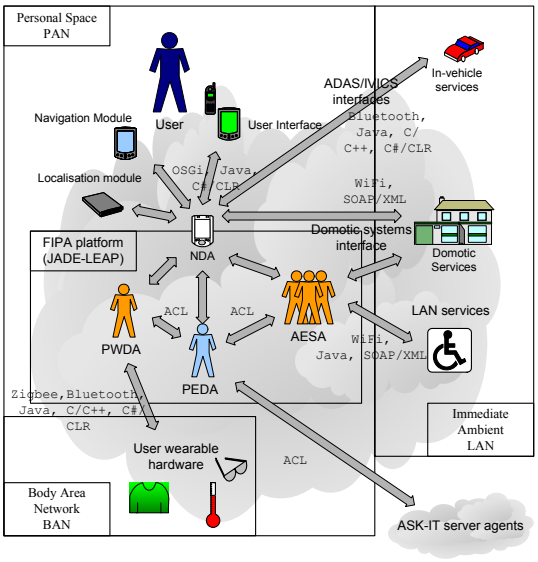

Figure 1.8 – Architecture côté client du système ASK-IT [Spanoudakis et Moraitis, 2006]

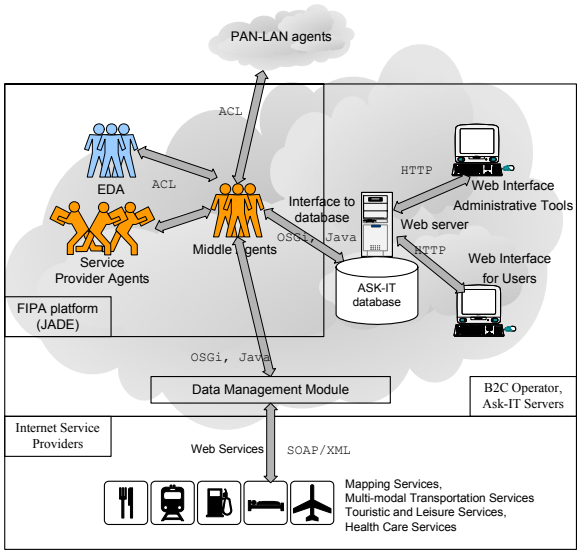

Figure 1.9 – Architecture côté serveur du système ASK-IT [Spanoudakis et Moraitis, 2006]

Les figures 1.8 et 1.9 représentent respectivement l'architecture ASK-IT côté client et serveur. Les données contextuelles sont gérées par différents types d'agents, en fonction du type de données. Du côté client, l'agent *PEDA* est l'agent central, en charge de fournir à l'utilisateur des services disponibles (c'est son agent personnel), et c'est lui qui interagit avec le serveur. Il communique avec l'agent *PWDA* qui gère les interactions avec les capteurs de l'utilisateur, ainsi qu'avec les agents *AESA* qui gèrent la bonne gestion de l'environnement direct de l'utilisateur en fonction des habitudes et besoins de celui-ci. Du côté du serveur, les agents *Middle* servent d'annuaires pour les agents *Provider*, qui fournissent des services à l'utilisateur. Les agents *EDA*, quant à eux, fournissent une expertise concernant différents types de handicaps, afin de spécialiser les réponses aux requêtes en tenant compte des besoins et requis liés aux différents handicaps éventuels de l'utilisateur. Les données contextuelles manipulées par ces différents agents sont modélisées grâce à une ontologie partagée entre tous les agents.

Évaluation L'utilisation d'une ontologie commune permet d'avoir une sémantique riche associée à chaque donnée, mais tout ajout d'une nouvelle donnée implique une intervention explicite du concepteur afin de l'insérer convenablement dans l'ontologie, et afin d'ajuster le comportement de l'agent qui sera en charge de cette nouvelle donnée. La topologie client-serveur adoptée par le système ASK-IT entraîne la nécessité d'un élément central, malgré une distribution d'une partie des traitements à réaliser. Concernant l'ouverture du système, l'association d'un agent à tout nouveau dispositif du système par le concepteur est nécessaire mais n'implique pas de redémarrer le système. En revanche, aucun mécanisme n'est proposé pour assurer le respect de la confidentialité des données. Enfin, bien qu'il n'existe pas de mécanisme expliquant le comportement adopté par le système, il est capable de proposer à l'utilisateur des services disponibles et jugés pertinent en fonction de son contexte et grâce aux règles de fonctionnement attribuées aux différents agents.

1.2.2.9 SIM

Présentation [Baek *et al.*, 2007] présente l'architecture SIM, un framework pour la gestion et l'exploitation des données issues de capteurs en système ambiant. En particulier, il assure une transformation correcte entre les données de bas niveaux (perçues par les capteurs) et les données de haut niveau, comme la localisation. Il étudie en particulier les valeurs obtenues depuis les différents capteurs afin de détecter les conflits possibles, comme l'obtention de plusieurs valeurs différentes pour une même information, afin d'en déterminer la valeur correcte. Par exemple, si deux informations de localisation donnent deux valeurs différentes, il calculera une approximation de la vraie valeur en se basant sur les deux valeurs qu'il possède (en calculant la position moyenne entre les deux positions perçues, par exemple). Le système inclut un agent dit "intelligent" dont l'objectif est de modifier l'état des différents effecteurs de l'environnement en fonction des données perçues, grâce à un comportement établi au préalable par le concepteur.

Évaluation Ce système s'intéresse principalement à la justesse et à la cohérence des données. Cela est possible grâce à l'expressivité des données, bien que la modélisation de ces données soit peu générique et ad-hoc à ce système (pas d'ontologie utilisée, par exemple). Le contrôle du système ambiant est centralisé au niveau de l'agent "intelligent". Aucune information n'est donnée quant à la capacité du système à permettre l'ajout et/ou la suppression de dispositifs en cours de fonctionnement. Il ne semble pas non plus proposer de mécanisme de gestion de la confidentialité des données exploitées. Enfin, concernant la proactivité du système, l'agent "intelligent" semble permettre d'assurer un contrôle du système ambiant basé sur des règles préétablies par le concepteur ; mais le peu de détails apportés sur son fonctionnement rend impossible de savoir s'il fournit des explications sur les choix de ses actions.

1.2.2.10 Système de Tapia

Présentation [Tapia *et al.*, 2008] propose une architecture multi-agent pour la gestion d'un environnement ambiant. Il s'agit d'une architecture distribuée constituée d'agents capables

d'interagir avec les différents dispositifs de l'environnement ambiant en cours de fonctionnement.

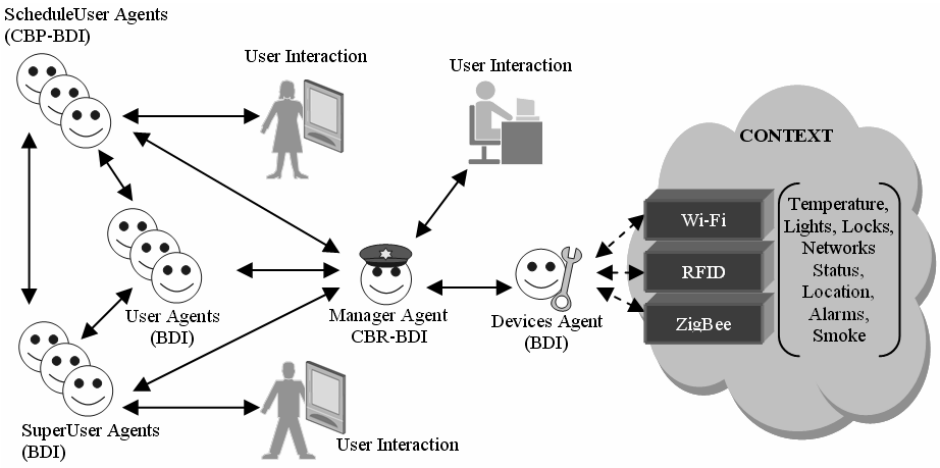

Figure 1.10 – Le système de [Tapia *et al.*, 2008]

La figure 1.10 illustre les différents agents du système :

- L'agent *Device* gère les interactions entre les dispositifs hardware (capteurs, effecteurs) et le système ;
- L'agent *Manager* est un agent cognitif possédant deux rôles. D'une part, il gère l'ensemble des informations perçues depuis les agents *Device* afin de connaître la localisation de l'utilisateur et le statut de l'environnement ambiant (température, luminosité, etc.). D'autre part, il utilise un mécanisme basé sur le Case Based Reasoning (voir section 2.2.1.7) pour associer à chaque situation l'action qui avait été réalisée dans la situation la plus similaire, et ainsi déterminer la meilleure action à réaliser afin d'attribuer des tâches aux autres agents ;
- L'agent *ScheduleUser* planifie les activités journalières de l'utilisateur à l'aide du mécanisme appelé le *Case-Based Planning* proposé par ([Bedia et Corchado, 2002]). Le planning obtenu est dynamique en fonction des besoins de chaque utilisateur, il peut donc évoluer si les préférences d'un utilisateur change ;
- L'agent *SuperUser* gère les actions à réaliser par les agents *User* et reçoit des rapports périodiques qu'il transmet à l'agent *Manager* ;
- L'agent *User* gère les données personnelles de l'utilisateur. Il transmet des informations en continu aux agents *ScheduleUser* et *SuperUser*, et s'assure que les actions demandées par ces derniers sont réalisées.

Évaluation Le point fort de ce système concerne l'aspect proactivité, implémenté en utilisant le CBR et permettant au système d'améliorer en continu son comportement, en augmentant ou en modifiant ses connaissances. Cependant, il n'offre pas de justification à l'utilisateur sur ses choix d'actions, et ne fournit pas non plus de mécanisme pour la confidentialité des données des utilisateurs. Les données sont modélisées de façon ad-hoc, ne leur donnant alors qu'une expressivité limitée Une intervention explicite du concepteur au niveau des agents Device est nécessaire pour ajouter un nouveau type de données. L'ajout d'un dispositif est impossible en cours de fonctionnement. Enfin, malgré une distribution

des traitements au sein de différents agents, l'aspect distribution est limité par la présence de l'agent central Manager.

1.2.2.11 Système de Zaidenberg

Présentation Le système proposé par [Zaidenberg, 2009] a pour objectif d'attribuer un comportement satisfaisant pour l'utilisateur à un système ambiant. Il se base sur un apprentissage par renforcement (voir section 2.2.2), grâce auquel il va tenter de déterminer, pour chaque situation rencontrée, l'action qui satisfera l'utilisateur. C'est donc indirectement l'utilisateur qui, par son approbation ou ses contradictions, va apprendre au système quel comportement adopter. L'environnement ambiant de l'utilisateur est représenté par un ensemble de prédicats.

Le choix d'un comportement résultant d'un apprentissage est justifié par le fait que la confiance de l'utilisateur envers le système ambiant est un prérequis essentiel à son acceptation. Donc, un système dont le comportement est conçu *a priori* risque de ne pas être adapté aux spécificités d'un utilisateur particulier, et doit donc être appris dynamiquement en fonction des actions de ce dernier.

Évaluation L'apprentissage par renforcement mis en oeuvre par ce système lui permet d'adopter un comportement proactif dans l'environnement ambiant, qui est alors capable de s'améliorer face à de nouvelles situations rencontrées. De plus, la modélisation des états de l'environnement utilisée par cet algorithme d'apprentissage permet au système de justifier à tout moment ses choix d'actions. Cette modélisation limite cependant l'expressivité des données tout en obligeant l'utilisateur à expliciter le format de toute nouvelle donnée en prédicat. Il est possible de faire disparaître et réapparaître des dispositifs, mais tous les dispositifs doivent nécessairement être déclarés au démarrage du système pour être utilisables.

1.2.2.12 ATRACO

Présentation Le système du projet ATRACO, décrit notamment par [Minker *et al.*, 2010], vise à concevoir des écosystèmes ambiants. Un écosystème ambiant est défini par [Goumopoulos *et al.*, 2008] comme *a space populated by connected devices and services that are interrelated with each other, the environment and the people, supporting the users' everyday activities in a meaningful way* [1].

Le système ATRACO cherche à établir l'activité dans laquelle est engagé un utilisateur, afin de créer une "sphère d'activité" composée des ressources (dispositifs, connaissances et agents) pour l'assister dans cette activité. Toutes ces connaissances sont modélisées grâce à des différentes ontologies, le système ATRACO intégrant des mécanismes d'alignement entre ces différentes ontologies. Ces connaissances sont couplées à un mécanisme d'apprentissage basé sur la logique floue et sont exploitées afin d'observer les actions des utilisateurs pour en déduire un comportement à attribuer aux dispositifs du système ambiant.

1. Notre traduction : "un espace doté de dispositifs et services connectés entre eux, à l'environnement et aux gens, assistant les utilisateurs dans leurs activités quotidiennes de façon significative."

Évaluation La distribution est un des deux seuls critères à ne pas être respecté, car le système ATRACO possède un élément central dans son architecture. L'autre critère non respecté est celui de la généricité, bien qu'il propose des mécanismes permettant l'intégration de nouveau type de données. La modélisation des données sous forme d'ontologies assure le respect du critère d'expressivité. Le système propose aussi des mécanismes pour la confidentialité des données qui exploitent des politiques d'accès modélisées sous forme d'ontologies et dépendant du contexte dans lequel se situent les utilisateurs. ATRACO est aussi capable de gérer l'apparition et de la disparition de dispositifs en cours de fonctionnement. Enfin, le système est capable d'adapter de façon proactive l'environnement des utilisateurs pour l'assister dans son activité courante à partir de règles apprises qu'il peut présenter ensuite aux utilisateurs.

1.2.2.13 SPACES

Présentation Le système SPACES de [Romero *et al.*, 2010] est une version étendue du système COSMOS de [Conan *et al.*, 2007]. Il s'agit d'un framework orienté composant pour la gestion du contexte en environnement ambiant, basé sur la notion de *noeud de contexte*.

Un noeud de contexte est un composant logiciel dont le comportement, les interfaces requises et les interfaces fournies sont définis par le concepteur. Il peut être actif ou passif selon qu'il agisse de façon proactive ou se limite à répondre aux sollicitations. Il peut communiquer par observation ou par notification selon qu'il recherche l'information ou attend de la recevoir. Enfin il peut être passant ou bloquant selon qu'il transmette les informations reçues ou pas. De plus, un grand nombre d'opérateurs applicables aux données de contextes sont fournis au concepteur (fusion de données, détection de seuil, etc.). Il est alors possible de concevoir puis d'assembler différents noeuds de contexte pour créer un graphe de noeuds de contexte. Cet ensemble de noeuds de contexte assure alors la gestion du contexte en fonction de leur organisation et des comportements attribués à chacun des noeuds. L'apport principal de SPACES par rapport à COSMOS concerne la possibilité de distribuer les noeuds de contexte.

Évaluation Le premier aspect à souligner concerne la propriété de distribution du système qui est ici parfaitement respectée. Le système adopte aussi un comportement proactif qui est le résultat de la composition d'éléments logiciels. Cependant, le système ne justifie pas son comportement auprès de l'utilisateur, et il ne lui assure pas non plus la confidentialité de ses données. Il est tout à fait capable d'intégrer de nouveaux dispositifs, et donc de nouveaux noeuds de contexte, en cours de fonctionnement. Enfin, la modélisation des données est basée sur la classification des types média MIME, ce qui limite sensiblement la généricité des données mais leur donne une certaine expressivité.

1.2.2.14 Système de Dujardin

Présentation [Dujardin *et al.*, 2011] propose un système multi-agent destiné à la gestion domotique d'une maison. Selon le mode de fonctionnement choisi par l'utilisateur, ce système peut contrôler la maison ou se contenter de proposer des actions. L'ensemble des dispositifs

de l'environnement ambiant est représenté grâce à un simulateur de système ambiant, ce dernier récupérant l'ensemble des données envoyées par les différents capteurs réels sur un bus logiciel afin de mettre à jour sa simulation.

L'objectif de ce système est de contrôler un environnement ambiant à l'aide de règles. Le système se compose d'un ensemble d'agents associés aux différents dispositifs du système ambiant, chacun avec leurs buts à atteindre (sous forme de conditions à respecter, tel que luminosité $<$ lum_ max) et des possibilités d'interactions. Chaque agent utilise un planificateur pour établir des listes d'actions possibles pour atteindre son but à partir de ses connaissances. Puis il choisit parmi ces plans celui qui lui semble préférable en fonction de ses motivations. Les motivations choisies ici sont orientées écologie, c'est-à-dire que des actions comme éteindre ou diminuer sont préférées à des actions comme allumer ou augmenter.

Évaluation L'utilisation d'un simulateur pour représenter chaque donnée permet d'associer une sémantique ad-hoc à chacune de ces données ; cela limite fortement la généricité de ces données sans permettre pour autant de réels traitements liés à leur sémantique. Les traitements de gestion des différents dispositifs sont distribués sous forme d'agents, mais requièrent un accès au simulateur central. L'ouverture du système nécessite une intervention explicite de l'utilisateur et une reconfiguration du système (au niveau du simulateur notamment), et aucun mécanisme concernant la confidentialité n'est proposé. Enfin, à l'aide de règles possédant un format simple et compréhensible par l'utilisateur, le système est capable de planifier puis de réaliser des actions en fonction des données perçues, mais le choix de ses actions n'est pas explicitement justifié à l'utilisateur.

1.3 Discussion

Dans cette première partie de l'état de l'art, nous avons présenté plusieurs outils pour mettre en oeuvre la sensibilisation au contexte dans un système ambiant. Parmi ces outils, nous distinguons les gestionnaires de contexte qui gèrent la perception et l'interprétation du contexte, en laissant l'aspect utilisation à des processus clients. La propriété de proactivité n'est donc logiquement pas mise en oeuvre dans de tels systèmes.

D'autres systèmes pour mettre en oeuvre la sensibilisation au contexte ne se limitent pas à l'observation et à la mise à disposition des données contextuelles, mais utilisent ces données pour rendre un système ambiant capable d'adapter son état ou sa fonctionnalité. De tels systèmes peuvent être eux-mêmes classés en deux catégories : ceux dont le comportement s'appuie sur une base de règles préétablies, et ceux capables de modifier leur comportement par apprentissage.

L'étude de ces systèmes par rapport à notre grille d'analyse (tableau 1.3), montre que, généralement, un système respectant le critère de généricité ne respecte pas le critère d'expressivité, et inversement. Cela peut s'expliquer par le fait que plus la modélisation du contexte tend à fournir un niveau important d'expressivité, plus le modèle utilisé est sophistiqué et spécialisé. Par conséquent, il est difficile d'y intégrer un nouveau type de donnée sans intervention du concepteur devient difficile. D'un autre côté, une donnée exploitée uniquement

Auteurs	Système	Année	Généricité	Expressivité	Distribution	Ouverture	Proactivité	Confidentialité	Explicabilité
Franklin	Intelligent Classroom	1998	-	-	--	--	+	--	-
Mozer	ACHE	1998	--	+	--	-	++	--	-
Dey	Context Toolkit	2001	++	--	-	++	--	++	--
Hofer	Hydrogen	2003	+	-	+	++	--	-	--
Hagras	iDorm	2004	--	-	-	--	++	--	-
Chen	CoBrA	2005	--	++	--	++	--	+	-
Rey	Contexteur	2005	-	+	++	++	+	--	--
Spanoudakis	ASK-IT	2006	--	++	-	++	+	--	--
Baek	SIM	2007	-	+	-	--	+	--	--
Tapia	...	2008	-	-	-	--	++	--	-
Conan	Spaces	2008	-	+	++	++	+	--	--
Zaidenberg	...	2009	-	-	-	+	++	--	++
Minker	ATRACO	2010	-	++	-	++	++	++	+
Dujardin	...	2011	--	-	-	--	+	--	-

Tableau 1.3 – Respect des critères définis en section 1.2.1 par les différents systèmes

pour sa valeur sera totalement dénuée d'expressivité, mais sera parfaitement générique.

Bien qu'ils ne prennent pas en compte le critère de proactivité, plusieurs gestionnaires de contexte (tels que le Context Toolkit ou CoBrA), fournissent de bons résultats concernant les autres critères, notamment la distribution, l'expressivité et l'ouverture. En revanche, dès que nous cherchons à étudier les systèmes respectant le critère de proactivité, les performances de ces systèmes vis-à-vis des autres critères se dégradent. Le comportement des systèmes basés sur des règles *ad-hoc* est lui-même limité aux situations connues. Enfin, les systèmes dotés de la capacité d'apprentissage, qui nous semblent pourtant les plus intéressants, ont les critères de distribution et d'ouverture généralement pénalisés. Il semble en effet que ces systèmes, malgré leur capacité à modifier leur comportement, ne peuvent que difficilement réagir à la réelle dynamique des systèmes ambiants, notamment lors de l'apparition et de la distribution de dispositifs dans l'environnement.

Nous envisageons la conception d'un système complémentaire à un gestionnaire de contexte. Ce système serait en particulier en charge du respect du critère de proactivité qui n'est pas présent dans les gestionnaires de contexte. Cependant, le respect de ce critère implique que ce système complémentaire soit capable d'apprendre dynamiquement le comportement à attribuer à un système ambiant. Or, il est inutile de disposer d'un gestionnaire de contexte gérant par exemple l'aspect ouverture si nous lui associons un système gérant par apprentissage la proactivité qui soit incapable de tenir compte de nouvelles données, ou de fonctionner malgré la disparition d'une donnée. Il est donc nécessaire de disposer d'une méthode d'apprentissage apte à attribuer les bons comportements à un système ambiant sans pour autant dégrader le respect des critères du gestionnaire de contexte sur lequel il sera appliqué.

Dans la section suivante, nous allons donc analyser les principales méthodes d'apprentissages existantes, en évaluant leur pertinence dans le cadre des systèmes ambiants.

2 Apprentissage d'un comportement

DANS ce chapitre, nous nous intéressons aux techniques d'apprentissage utilisables pour qu'un système ambiant s'enrichisse durant son activité des comportements jugés satisfaisants en temps réel par ses utilisateurs. Dans cette section, nous commençons par définir différents concepts nécessaires à la compréhension du domaine de l'apprentissage artificiel. Nous présentons ensuite les algorithmes d'apprentissage les plus classiques, ainsi que des techniques de filtrage de données inutiles utilisées dans le but d'améliorer les processus d'apprentissage. Nous terminons par une évaluation de la pertinence de ces techniques pour l'apprentissage automatique vis-à-vis de notre problématique.

2.1 Notions relatives à l'apprentissage artificiel

Nous présentons ici plusieurs notions liées au domaine de l'apprentissage artificiel. Ces définitions ne sont qu'une synthèse des concepts fondamentaux de ce domaine, une étude plus détaillée de ces concepts étant proposé par [Cornuéjols et Miclet, 2011].

Notion d'apprentissage artificiel Elle est définie par [Cornuéjols et Miclet, 2011] comme *toute méthode permettant de construire un modèle de la réalité à partir de données, soit en améliorant un modèle partiel ou moins général, soit en créant complètement le modèle*. Pour notre étude, nous conservons cette définition générale, mais en employant le terme d'"environnement" au lieu du terme "réalité". Un environnement est la sous-partie de la réalité perceptible par un système. Un algorithme d'apprentissage exploite donc les données perçues de son environnement pour se construire un modèle de cet environnement. Un système utilisant un algorithme d'apprentissage pour réaliser un processus d'apprentissage est appelé "système apprenant".

Il est important de préciser que ce modèle n'est pas un modèle "dans l'absolu". Le modèle produit est une représentation particulière de l'environnement construite pour répondre à un objectif. Cet objectif dépend quant à lui de la tâche à réaliser. Un système apprenant le comportement d'un utilisateur dans un environnement ambiant ne va pas construire un modèle absolu de tout ce qu'il perçoit (il ne va pas chercher à modéliser la diffusion lumineuse entre les pièces quand bien même il possèderait les données pour cela), mais plutôt un modèle plus spécifique du comportement de l'utilisateur.

Une hypothèse inhérente à l'application d'un algorithme d'apprentissage sur un jeu de données est que celui-ci permet de se construire une représentation du monde (un modèle) suffisante pour en extraire l'objectif à apprendre.

Notion de tâche Un processus d'apprentissage dépend fortement de la tâche à réaliser, c'est-à-dire du résultat attendu. Sans être exhaustif, nous présentons un certain nombre de tâches requérant l'application d'algorithmes d'apprentissage.

La tâche la plus classique concerne le problème d'*identification*. Un algorithme perçoit des données servant d'exemples et cherche à en déduire une règle générale. Par exemple, supposons un algorithme qui perçoit un ensemble d'animaux caractérisés par différents attributs (tels que leurs tailles et leurs couleurs), ainsi que la race de chaque animal. L'algorithme généralise alors les exemples perçus afin de déterminer ce qui fait qu'un animal appartient à telle ou telle race. Par la suite, face à un nouvel exemple d'animal, il devient capable de dire de quelle race il est.

Un autre exemple de tâche concerne l'*approximation*. À partir d'un jeu de données, un algorithme peut apprendre à approximer une fonction inconnue. Ainsi, face à un nouveau jeu de données, l'algorithme pourra estimer une valeur de sortie probable de cette fonction. De façon plus générale, les tâches d'*identification* et d'*approximation* peuvent être vues comme des tâches de *généralisation*.

L'apprentissage peut aussi servir à réaliser une tâche de *prédiction*. À partir d'un historique d'exemples, un algorithme apprend à prédire l'état futur d'une donnée. Ce type d'apprentissage est notamment appliqué dans le domaine de la finance, pour prédire l'état futur des marchés.

Un dernier exemple d'apprentissage concerne l'*amélioration* de connaissances. Un joueur d'échec virtuel peut connaître les règles des échecs, et par conséquent savoir jouer, et réaliser un apprentissage pour améliorer sa stratégie de jeu de parties en parties. Cela ne change pas sa connaissance sur les règles du jeu, mais plutôt ses performances dans la réalisation de sa tâche.

Notion de performances L'évaluation de la performance d'un processus d'apprentissage peut porter sur plusieurs critères tels que :

- le taux d'erreur : si le système apprenant est capable de détecter ses erreurs (soit par des capacités d'introspection qui lui permettent de déterminer ses erreurs par lui-même, soit par l'intervention d'un élément extérieur qui possède la connaissance à apprendre), il est possible de déterminer un taux d'erreurs qui exprime la justesse de l'apprentissage réalisé (le nombre d'erreurs par rapport aux nombre de réponses proposés par exemple).
- les conséquences en cas d'erreurs : celles-ci varient en fonction de la tâche que doit réaliser le système apprenant. Un système ayant un taux d'erreur faible mais fournissant toujours des résultats incorrects dans les cas les plus critiques peut être considéré comme moins performant qu'un système ayant un taux d'erreur plus élevé mais fournissant des résultats erronés uniquement sur des cas peu critiques.
- le coût de traitement : l'exploitation de données pour réaliser un apprentissage im-

plique un coût en temps et en processus computationnel, soit au niveau de l'apprentissage lui-même (construction du modèle), soit au niveau de l'exploitation de ce modèle. En fonction de la tâche à réaliser et des ressources disponibles pour la réaliser, ce coût peut avoir une influence plus ou moins forte sur l'évaluation de la performance de l'algorithme d'apprentissage. Par exemple, quand bien même un algorithme d'apprentissage atteint un taux d'erreur nul, si ce résultat nécessite plusieurs heures de traitement alors qu'il est censé être appliqué à une tâche en temps réel, sa performance sera probablement jugée très faible.

– l'intelligibilité : un algorithme d'apprentissage doit être capable de "justifier" le résultat de son apprentissage. Un exemple classique est celui d'un système ayant appris à fournir un diagnostic à un patient en fonction des symptômes exprimés par ce dernier : pour être jugé crédible par le médecin, le résultat proposé ne peut se limiter à un diagnostic Il doit nécessairement s'accompagner du raisonnement ayant mené à cette conclusion.

Nous pouvons remarquer que l'évaluation du résultat est liée à la tâche à réaliser. Une comparaison des performances de plusieurs algorithmes d'apprentissage dans l'absolu, par comparaison du taux d'erreurs par exemple, n'a donc pas de sens. Il est nécessaire d'établir un processus d'évaluation tenant compte de la tâche à réaliser, et donc du niveau d'importance accordé au respect de chacun des critères de performance.

Notion de protocole Un algorithme d'apprentissage se caractérise par le *protocole* qui décrit les interactions qu'il entretient avec son environnement. Le protocole inclut la fréquence d'interaction avec l'environnement, le type de données que l'algorithme prend en entrée, et le type de données qu'il retourne en sortie.

Dans le cas d'un apprentissage *hors ligne*, l'algorithme reçoit la totalité des données et les utilise pour construire son modèle sans interagir avec son environnement. En règle générale, il parcourt séquentiellement toutes les entrées pour construire son modèle. Une fois la liste des entrées parcourue, l'apprentissage est terminé, et le modèle est alors construit. Un apprentissage *hors ligne* fonctionne donc en deux phases : la première phase où il reçoit les données depuis l'environnement qu'il utilise pour construire son modèle, et la seconde phase où il applique son modèle déjà construit en fonction des demandes de l'environnement.

Dans le cas d'un apprentissage *en ligne*, les données arrivent séquentiellement et l'algorithme propose une réponse à chaque réception de nouvelle(s) donnée(s), puis reçoit une réponse de l'environnement. Ensuite, il utilise cette réponse pour faire évoluer son modèle. L'apprentissage se fait donc au travers de l'interaction entre l'apprenant et son environnement. Il n'y a donc pas deux phases distinctes comme pour l'apprentissage *hors ligne*, l'algorithme enrichissant son modèle au fur et à mesure de ses interactions.

L'identification est un exemple de tâche utilisant généralement un apprentissage hors ligne, tandis que la prédiction est une tâche pour lequel l'apprentissage en ligne est davantage applicable. La figure 2.1 illustre les interactions entre un système apprenant et son environnement. À gauche, pour l'apprentissage hors ligne, nous pouvons voir l'apprenant percevant chaque donnée et construisant son modèle répondant aux différentes données au

fur et à mesure qu'il les traite (seule la première phase, celle de l'apprentissage, est illustrée). À droite, pour l'apprentissage en ligne, l'apprenant propose un retour pour chaque entrée perçue, et reçoit ensuite un retour (*feedback*) de l'environnement lui confirmant la justesse de sa proposition (ici, un *feedback* booléen).

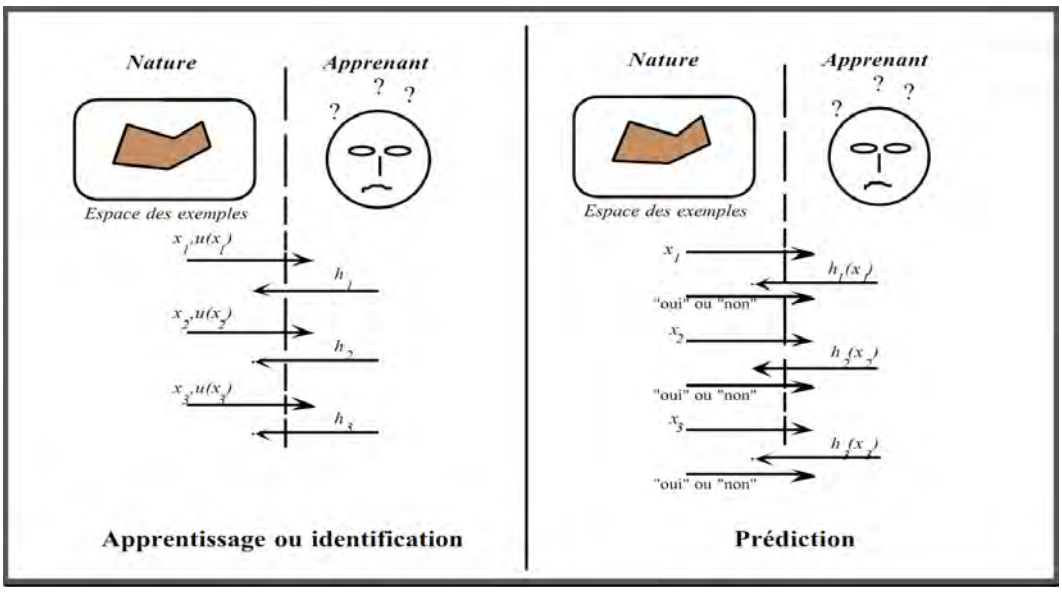

Figure 2.1 – Illustration du fonctionnement d'un algorithme d'apprentissage hors ligne (à gauche) et d'un algorithme d'apprentissage en ligne (à droite) [Cornuéjols et Miclet, 2011]

Notion de surapprentissage Si nous fournissons un jeu de données à un algorithme d'apprentissage, celui-ci gagnera d'abord en performances au fur et à mesure qu'il construira son modèle, puis commencera à perdre en performances à force de trop vouloir optimiser son modèle : il y a surapprentissage. La figure 2.2 illustre ce phénomène. L'algorithme d'apprentissage cherche à séparer deux classes à partir d'un ensemble d'exemples (carrés et ronds). L'image centrale montre le résultat produit par un apprentissage correct, la majorité des exemples étant correctement catégorisée par l'algorithme. Un sous-apprentissage entraîne évidemment un plus grand nombre d'erreurs, comme le montre l'image de gauche. En revanche, l'image de droite montre les effets du surapprentissage, où l'algorithme perd en capacité de généralisation en s'efforçant de produire une parfaite catégorisation pour la totalité des exemples.

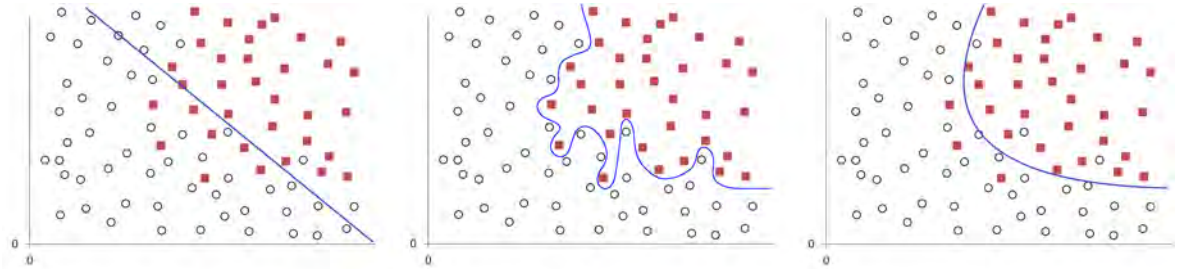

Figure 2.2 – Illustration de l'effet de surapprentissage

La majorité des applications réelles ne présentent pas de séparation clairement définie entre les classes à apprendre. En tentant de produire un résultat correct pour tous les exemples en entrée, un algorithme d'apprentissage finit donc par dégrader ses performances. Pour éviter ce phénomène, plusieurs stratégies ont été proposées. La plus classique, dans le cas d'un apprentissage *hors ligne*, consiste à séparer l'ensemble des données en entrée en deux sous-groupes : le premier sert à la réalisation de l'apprentissage, et le second sert de jeu de tests. Ainsi, au fur et à mesure que l'algorithme construit son modèle, il vérifie ses performances sur le jeu de test. Dès lors qu'il observe une dégradation de ses performances, il considère qu'il a atteint ses performances optimales. Une telle stratégie requiert bien sûr d'avoir, non seulement assez de données pour réaliser son apprentissage, mais aussi assez de données pour l'évaluer.

2.2 Principaux algorithmes d'apprentissage

Dans cette section, nous présentons les principaux algorithmes d'apprentissage. Ceux-ci sont généralement regroupés en trois grandes familles : les algorithmes d'apprentissage supervisé, non-supervisé et par renforcement. Nous commençons donc par présenter les principaux algorithmes d'apprentissage supervisé, puis par renforcement. En revanche, nous n'abordons pas ici les algorithmes d'apprentissage non supervisé. En effet, de tels algorithmes permettent de diviser des données en différentes catégories, sans association de classe spécifique à ces différentes catégories. Or, l'application qui nous intéresse ici est l'apprentissage d'un comportement à attribuer à un environnement ambiant en fonction du contexte perçu, c'est-à-dire l'association à toute situation perçue d'une action à réaliser. Les algorithmes d'apprentissage non supervisés ne sont donc pas appropriés à une telle problématique.

Tout au long de la présentation de ces différents algorithmes, nous nous intéresserons à la capacité de ces algorithmes à être appliqués à l'apprentissage du comportement utilisateur en environnement ambiant. Cela concerne leur capacité à déterminer pour chaque situation, quelle action aurait été réalisée par les utilisateurs, ce qui rend notamment l'application de classifieurs appropriée. Mais il est aussi important qu'ils soient capables de fonctionner en prenant en compte :

- les changements de comportements des utilisateurs, et donc être capable de modifier le comportement déjà appris ;
- les changements survenant dans l'environnement, et donc être capable de réaliser leurs apprentissages en considérant l'ajout et la suppression de données en cours de fonctionnement.

Pour un apprentissage hors ligne, nous supposons que l'algorithme est relancé plusieurs fois au cours du temps afin de tenir compte de l'arrivée de nouvelles données, et éventuellement de changements en cours de fonctionnement. Qu'il s'agisse d'un apprentissage en ligne ou hors ligne, il s'agit d'évaluer la capacité d'un algorithme à apprendre le bon comportement (en sachant que les exemples les plus anciens peuvent contredire les plus récents), ainsi que sa capacité à réaliser ses traitements en sachant que le nombre de données dans les différents exemples peut varier.

2.2.1 Algorithmes d'apprentissage supervisé

Le principe de l'apprentissage supervisé se base sur l'existence d'une entité, communément appelé *oracle*, qui possède la connaissance à apprendre. Cet oracle peut être vu comme un expert enseignant sa connaissance à l'apprenant au travers d'exemples. Dans un algorithme d'apprentissage supervisé, l'apprenant reçoit un ensemble d'entrées $\mathcal{E}$. Chaque entrée e_i est composée de n attributs. En fonction des capacités de l'algorithme, ces attributs peuvent prendre des valeurs discrètes ou continues. De plus, l'oracle associe à chacune des entrées une classe c parmi l'ensemble fini $\mathcal{C}$ de classes possibles. Si l'ensemble $\mathcal{C}$ ne contient que deux classes possibles, il s'agit d'un apprentissage de *concept*. En revanche, il l'ensemble $\mathcal{C}$ contient un plus grand nombre de classes, il s'agit d'un apprentissage de *classification*.

Un algorithme d'apprentissage supervisé dispose donc d'un ensemble d'exemples illustrant l'objectif à apprendre. A partir de ces exemples, il tente de construire un modèle qui lui permette d'attribuer à chacune des entrées la même étiquette que lui avait attribuée l'oracle. En d'autres termes, l'algorithme d'apprentissage considère que si le modèle qu'il a construit permet d'attribuer la bonne classe pour chaque exemple, ce modèle se rapproche suffisamment de la réalité, et donc lors d'une nouvelle entrée, il sera capable de lui associer la même sa classe que l'oracle.

2.2.1.1 Les k-plus proches voisins

L'algorithme des k-plus proches voisins, introduit par [Duda *et al.*, 1973], consiste à utiliser directement les données de chaque exemple perçu en entrée pour classifier tout nouvel exemple. Il va en particulier rechercher, pour tout nouvel exemple, les k données voisines les plus proches. Le voisinage est généralement mesuré en distance euclidienne entre deux exemples en se basant sur les données qui les caractérisent et qui représentent les dimensions de ces exemples. Pour déterminer la classe d'un nouvel exemple, l'algorithme comptabilise parmi les k données voisines le nombre d'exemples appartenant à chacune des classes, et associe au nouvel exemple en entrée la classe ayant comptabilisé le plus de points. La figure 2.3 donne un exemple d'application de cette méthode, où la recherche de la classe pour un exemple inconnu se fait par l'observation des $k = 8$ plus proches voisins ; avec 6 exemples étiquetés comme carrés et 2 exemples étiquetés comme ronds, l'algorithme étiquète le nouvel exemple comme un carré.

Le paramétrage d'un tel algorithme se focalise sur la valeur de k. Plus ce nombre est grand, plus les probabilités d'obtention d'un résultat correct augmentent. Mais si k devient trop important, le résultat inverse est obtenu. En effet, avec un paramètre k trop important, l'algorithme des k-plus proches voisins considère des exemples si éloignés de l'exemple à classifier que la probabilité que ces exemples appartiennent à la même classe que l'exemple à classifier diminue.

Il existe d'autres versions de l'algorithme des k-plus proches voisins qui limitent les traitements à réaliser. En effet, avec un grand nombre d'exemples de dimensions élevées, les traitements pour classifier un nouvel exemple peuvent être coûteux. Certains algorithmes réalisent alors des prétraitements sur les exemples en entrée pour élaguer les exemples isolés qui n'influenceraient pas le résultat obtenu, ou pour au contraire élaguer les exemples

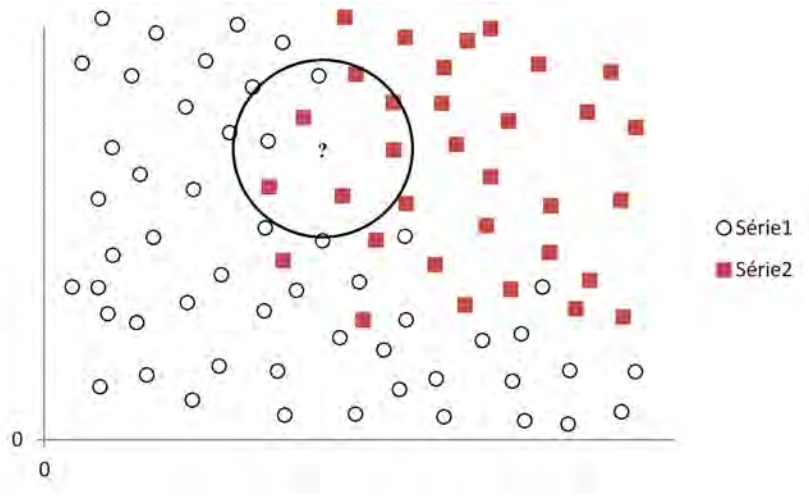

Figure 2.3 – Recherche de la classe d'un exemple par méthode des k-plus proches voisins, avec $k = 8$

suffisamment proches d'exemples de même classe pour ne pas être nécessaires à l'obtention du résultat. Ces différentes versions sont présentées notamment par [Belaïd et Belaïd, 1992].

L'algorithme des k-plus proches voisins a clairement l'avantage de la simplicité, tant dans son fonctionnement que dans sa mise en place. Cependant, ses performances sont fortement dépendantes du choix du paramètre k ainsi que de la fonction d'évaluation des distances. De plus, en disposant d'un grand nombre d'exemples et d'un paramètre k assez grand, cet algorithme admet une bonne tolérance aux exemples incorrects, mais le nombre de données dans ces exemples est fixé et ne peut évoluer.

2.2.1.2 Les arbres de décision

Les arbres de décision font partie des méthodes d'apprentissage classiques les plus faciles à utiliser. Leur fonctionnement se base sur la construction d'un arbre dont le parcours permet d'attribuer la bonne classe aux différentes entrées. L'exemple le plus classique de l'apprentissage par arbre de décision provient de l'étude de [Quinlan, 1993], où l'algorithme cherche à prédire si une équipe de sportifs ira jouer ou pas en fonction de la météo. Le tableau 2.1 montre un ensemble de 14 entrées, chaque entrée représentant le niveau d'ensoleillement (soleil, pluie ou couvert), la température (en °K), l'humidité (en %) et la présence ou pas de vent. Chaque entrée est associée à une étiquette {oui/non} permettant de savoir si l'équipe a joué ou pas.

A partir de ces données, l'algorithme d'apprentissage cherche à construire un arbre de décision permettant de séparer les différentes entrées en plusieurs sous-groupes d'entrées (les feuilles) appartenant à la même classe. L'arbre produit à partir de ces données est représenté dans la figure 2.4. Chaque nœud représente une condition sur les attributs, et pour toute nouvelle entrée perçue, le parcours de l'arbre se fait en fonction du résultat obtenu en appliquant les conditions des nœuds sur ses attributs. La classe de cette nouvelle entrée est alors établie en fonction de la feuille d'arrivée. Dans chaque nœud, le pourcentage d'exemples appartenant à chaque classe est représenté ; au niveau des feuilles de l'arbre, ce

Ensoleillement	Température	Humidité	Vent	Jouer
soleil	75	70	oui	oui
soleil	80	90	oui	non
soleil	85	85	non	non
soleil	72	95	non	non
soleil	69	70	non	oui
couvert	71	90	oui	oui
couvert	83	78	non	oui
couvert	64	65	oui	oui
couvert	71	75	non	oui
pluie	71	80	oui	non
pluie	65	70	oui	non
pluie	75	80	non	oui
pluie	68	80	non	oui
pluie	70	96	non	oui

Tableau 2.1 – Exemple d'un jeu de données pour la construction d'un arbre de décision [Quinlan, 1993]

pourcentage est donc de 100% pour une des classes.

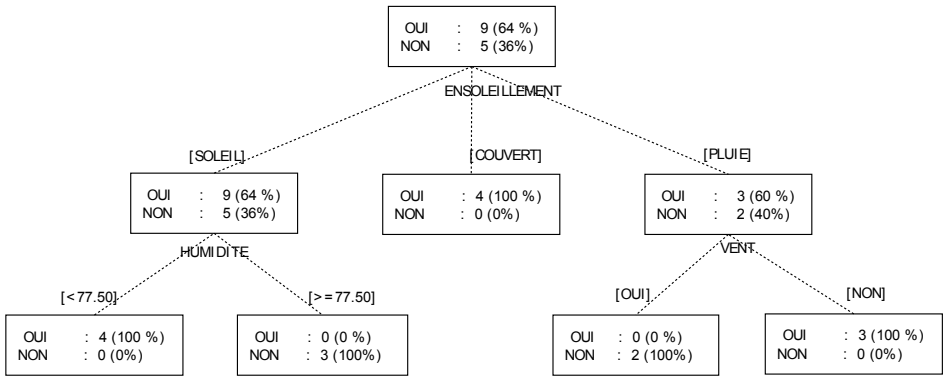

Figure 2.4 – Arbre de décision produit à partir du tableau 2.1

La construction de l'arbre à partir des données en entrée se fait en partant du nœud principal et en construisant progressivement l'arbre, en déterminant la condition de chaque nœud qui permettra de séparer au mieux les entrées de classes différentes. Dans l'exemple, l'algorithme a considéré que la valeur de l'ensoleillement était un bon critère, car il permet d'obtenir une feuille "pure" (c'est-à-dire, pour laquelle toutes les données appartiennent à la même classe). Puis, pour chaque sous-arbre créé, l'algorithme réitère le même traitement. En théorie, l'algorithme termine son traitement une fois que l'arbre n'est constitué que de feuilles pures.

Les performances d'un algorithme d'apprentissage par arbre de décision sont principalement liées à sa capacité à trouver la meilleure condition à attribuer à chaque nœud. En effet, plus la condition est optimale, plus l'arbre créé sera minimaliste, et donc plus son utilisation pour déterminer la classe d'une nouvelle entrée sera rapide. La performance de l'algorithme d'apprentissage sera aussi liée à sa capacité à prévenir le surapprentissage. Cela se fait soit en évitant de construire un sous-arbre pour un nœud presque pur (prétraitement), soit par

une construction complète de l'arbre suivi d'un élagage des sous-arbres jugés trop spécialisés (post-traitement). Ces algorithmes restent néanmoins assez limités dans leurs prises en compte des évolutions rencontrées.

2.2.1.3 Les réseaux de neurones

Les réseaux de neurones artificiels sont des modèles inspirés de la physiologie des cerveaux, composés de neurones interconnectés entre eux. En biologie, un neurone est la cellule composant le système nerveux d'un animal. Cette cellule se caractérise notamment par son excitabilité : elle perçoit des stimulations bioélectriques (ou signaux synaptiques) au niveau de ses dentrites qui, s'ils atteignent un certain seuil, entraînent une impulsion nerveuse en sortie au niveau de son axone. Connectés les uns aux autres au travers de leurs synapses, les neurones composant un système nerveux forment un réseau très complexe. Partant de l'idée que si le cerveau (humain en particulier) est capable d'apprendre alors qu'il n'est constitué que d'éléments primaires comme les neurones, une branche de la recherche en intelligence artificielle a mené à l'étude et à la modélisation des neurones pour concevoir des réseaux de neurones artificiels capables d'imiter les capacités cognitives du cerveau, notamment en terme de capacité d'apprentissage. C'est ainsi que [Rosenblatt, 1958] proposa le modèle des réseaux de neurones, ou *perceptron*.

Un réseau de neurone est usuellement un modèle composé de neurones connectés les uns aux autres selon une topologie spécifique. Un apprentissage basé sur les réseaux de neurone se fait *hors ligne*, le réseau de neurone étant construit à partir d'une base d'apprentissage composée d'exemples. Chaque neurone est composé de n entrées, n correspondant au nombre d'attributs composant les exemples de la base d'apprentissage. Chaque entrée d'un neurone possède un poids, les valeurs des exemples perçus par chaque neurone étant pondérées par ces poids. De plus, un neurone possède une fonction d'activation calculant la valeur en sortie du neurone en fonction de la somme des valeurs pondérées en entrée. Plusieurs fonctions existent, les plus classiques étant la fonction de Heaviside retournant exclusivement une valeur égale à 0 ou 1, ou la fonction sigmoïde retournant une valeur réelle entre 0 et 1. Enfin, un neurone possède une valeur de seuil, qui fixe la valeur minimum de la somme pondérée des entrées requise pour que le neurone s'active, c'est-à-dire pour qu'il renvoie une valeur supérieure à 0.5 en sortie (nécessairement 1 avec la fonction de Heaviside). La figure 2.5 représente l'état interne d'un neurone artificiel tel que nous venons de le décrire.

L'utilisation d'un neurone artificiel permet de séparer deux classes sur une base d'exemples sous condition que ces deux classes soient dites linéairement séparables [1]. Dans le cas de l'apprentissage de plusieurs classes (apprentissage de classification plutôt que de concept), plusieurs neurones sont nécessaires, un neurone étant associé à chaque classe, et la valeur en sortie du neurone représentant le niveau de probabilité d'appartenance à cette classe.

Les performances de l'apprentissage des réseaux de neurones se sont fortement amélio-

1. Un ensemble d'exemples est dit linéairement séparable s'il existe un hyperplan qui permet de séparer l'ensemble des exemples en fonction de leurs classes.

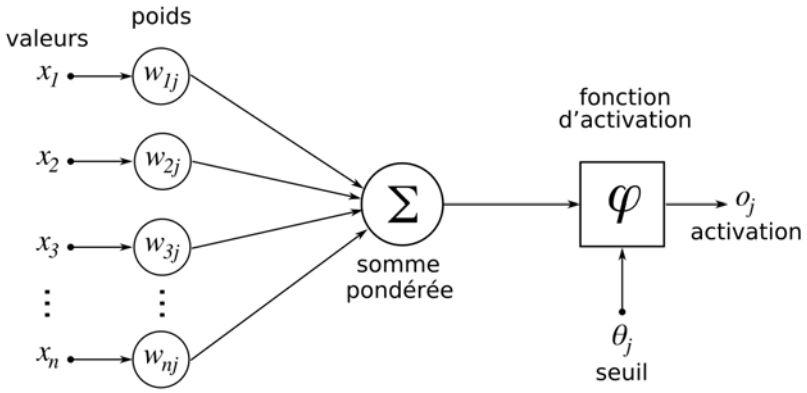

Figure 2.5 – Représentation de l'état interne d'un neurone artificiel

rées par l'intégration de réseaux de neurones multi-couches, introduit par [Hopfield, 1982], où des sorties de neurones sont connectés aux entrées d'autres neurones. Le nombre de couches représente le nombre de neurones entre l'entrée et la sortie du réseau. Ces réseaux de neurones permettent de représenter des fonctions, deux couches étant suffisantes pour représenter des fonctions continues, tandis que trois couches permettent de représenter toute fonction arbitraire.

Le nombre de neurones pour chaque couche, la fonction d'activation et le seuil de chaque neurone sont autant de paramètres qu'il faut choisir pour utiliser un réseau de neurone. Un nombre insuffisant de neurones pour apprendre une fonction trop complexe, ou un nombre trop important de neurones pour apprendre une fonction simple, peuvent fortement dégrader les performances des réseaux de neurones. Plusieurs études proposent des stratégies pour aider un concepteur à choisir ces paramètres, mais il n'existe pas de solution générale. De façon générale, les réseaux de neurones proposent de bonnes performances pour des tâches de classification, mais ne sont pas de base capables de tenir compte des évolutions du nombre de données, ni de la tâche à apprendre.

2.2.1.4 Les algorithmes génétiques

Le principe de fonctionnement des algorithmes génétiques, proposés par [Holland, 1975], se base sur le phénomène de *sélection naturelle*. Les différentes espèces dans le monde évoluent de génération en génération, la pérennité d'une espèce se faisant sur la base des principes darwinistes (en simplifié, la capacité pour un être vivant d'assurer la pérennité de son espèce). Cependant, au travers de l'agriculture et l'élevage, l'homme intervient directement dans ce processus, favorisant l'évolution de certaines espèces sur des critères différents. Il ne s'agit plus de sélection naturelle, mais plutôt de *sélection artificielle*. Le principe inhérent aux algorithmes génétiques est basé sur ce phénomène.

Le génotype d'un individu représente l'information génétique associée à cet individu, et est modélisée sous la forme d'un chromosome. Un algorithme génétique génère donc des individus, chacun d'eux possédant son propre génotype, et fait évoluer cet ensemble d'individus pour produire, par un mécanisme basé sur la sélection artificielle, des individus considérés comme plus performants.

Comme nous l'avons souligné en section 2.1, un algorithme d'apprentissage a pour objectif de construire un modèle de son environnement à partir d'exemples. [Cornuéjols et Miclet, 2011] parle d'apprentissage à deux espaces : l'*espace des exemples* d'où proviennent les exemples en entrées de l'algorithme, et l'*espace des hypothèses* qui constitue l'ensemble des modèles pouvant décrire l'environnement. Les algorithmes présentés jusqu'à présent fonctionnaient dans le but de construire progressivement ce modèle en s'enrichissant progressivement des exemples. Autrement dit, ils parcourent l'espace des hypothèses pour trouver le modèle décrivant au mieux les exemples perçus. Dans le cas des algorithmes génétiques, il s'agit d'un apprentissage à trois espaces : à l'espace des hypothèses est associé l'*espace "génotypique"*, chaque modèle étant associé à une expression génotypique, ou génotype. Pour appliquer un algorithme génétique, il suffit alors d'associer le format du modèle à celui du génotype. Un algorithme génétique ne cherche donc pas à trouver un modèle satisfaisant en parcourant l'espace des hypothèses, mais plutôt à faire évoluer des individus associés à différents modèles jusqu'à l'obtention d'un individu satisfaisant (autrement dit, un individu dont le génotype est associé à un modèle satisfaisant).

La représentation la plus classique pour un génome est une chaîne de bits, tout modèle étant alors transformable en chaîne de bits, et vice-et-versa. Bien sûr, le choix du nombre de bits, ou de façon générale la complexité d'un génotype, influence fortement les performances d'un algorithme génétique : un génotype trop simple limitera l'espace des hypothèses que pourra parcourir l'algorithme, tandis qu'un génotype trop complexe augmentera fortement le coût du parcours de l'espace des hypothèses.

En général, un algorithme génétique commence son traitement en générant un ensemble d'individus aléatoirement. Il entre ensuite dans un cycle à quatre étapes qu'il poursuit jusqu'à l'obtention d'un modèle satisfaisant, les critères de satisfaction pouvant être l'observation un gain inexistant entre deux générations successives, ou l'atteinte d'une limite en termes de ressources calculatoires.

1. **Évaluation** : Les modèles associés aux génotypes de chacun des génomes sont évalués vis-à-vis de la base d'exemples en entrée grâce à une fonction d'évaluation des performances, communément appelé fonction de fitness (*fitness function*). Cette étape d'évaluation affiche clairement l'appartenance des algorithmes génétiques à la famille des algorithmes d'apprentissage supervisé. C'est aussi cette étape qui est généralement la plus coûteuse, car l'évaluation des performances d'un modèle, à plus forte raison d'un ensemble de modèles, vis-à-vis d'un ensemble d'exemples s'avère généralement coûteuse en ressources.

2. **Sélection** : Cette étape consiste donc à choisir les individus qui mèneront à la création de nouveaux individus, ceux qui se limiteront à survivre et ceux qui disparaîtront. Conformément au processus de sélection naturelle, les individus considérés comme les "meilleurs" ont une probabilité plus forte d'être sélectionnés pour la survie et à la procréation.

3. **Création** : Les individus sélectionnés peuvent être employés pour créer de nouveaux individus dotés de nouveaux génotypes. Les deux principales méthodes sont les *croisements* et les *mutations* de génotypes. Dans le cas du croisement, deux génotypes sont combinés dans l'espoir d'obtenir des individus possédant les meilleures caractéris-

tiques des deux individus parents. Pour cela, un ou plusieurs points des génotypes des deux individus parents sont sélectionnés, et la suite des génotypes est intervertie. Dans le cas de la mutation, une partie (généralement plutôt petite) du génotype d'un individu est changé aléatoirement.

4. **Remplacement** : Cette étape vise à remplacer l'ancienne génération d'individus par la nouvelle génération. Une fois encore, le choix du nombre d'anciens individus conservés et supprimés est un paramètre de l'algorithme fixé par le concepteur. Plusieurs stratégies existent, mais la plus commune consiste à garder un nombre d'individus fixe en supprimant une partie des anciens individus. Un choix possible est alors de prendre l'ensemble des individus, anciens et nouveaux, de les évaluer et de conserver uniquement les plus performants.

Les algorithmes génétiques nécessitent un traitement assez important dans leurs mises en place, les choix de modélisation des génotypes ayant un impact très important sur les performances de l'algorithme. Cependant, cette modélisation peut théoriquement inclure la possibilité que des données soient ajoutées ou supprimées. De plus, les algorithmes génétiques sont capables de tenir compte d'une évolution de la fonction à apprendre. Cela ne les rend cependant pas forcément applicables à l'apprentissage du comportement des utilisateurs en environnement ambiant. En effet, le traitement des algorithmes génétiques est généralement assez long, car il est nécessaire d'évaluer chaque modèle produit pour pouvoir converger vers des individus plus performants.

2.2.1.5 Les machines à vecteurs de support

Les machines à vecteurs de support, ou SVM (*Support Vector Machine*), aussi appelées séparateurs à vaste marge en français, constituent une autre famille de classifieurs, fortement consolidés par des théories mathématiques. Ils ont été pour la première fois introduits par [Cortes et Vapnik, 1995].

Initialement, les SVM ne permettent que la séparation de deux classes linéairement séparables. La figure 2.6 illustre leur fonctionnement. Il existe en théorie une infinité de droites permettant de séparer les deux classes, mais l'objectif des SVM est de trouver la droite optimale séparant ces deux classes, cette droite maximisant la distance entre elle et chacun des exemples. De façon générale, l'objectif des SVM est de trouver, pour un problème de dimension D (autrement dit, un problème incluant D données différentes), l'hyperplan de dimension $D-1$ optimal séparant les exemples connus.

Une méthode ne permettant que la séparation de classes linéairement séparables n'est applicable qu'à une catégorie très limitée de problèmes. Cependant, il est possible d'appliquer les SVM à des problèmes non-linéairement séparables en étendant les dimensions du problème. Il s'agit alors de prendre une fonction, généralement nommé ϕ, qui transforme les exemples de dimensions D en exemples de dimensions D' plus importantes. L'intérêt de cette opération est illustré dans la figure 2.7 : à gauche, nous pouvons voir les exemples de dimension 2, tandis qu'à droite, nous pouvons voir ces mêmes exemples en dimension 3 après avoir été transformés par une fonction ϕ . La séparation entre ces exemples se fait alors grâce à un plan.

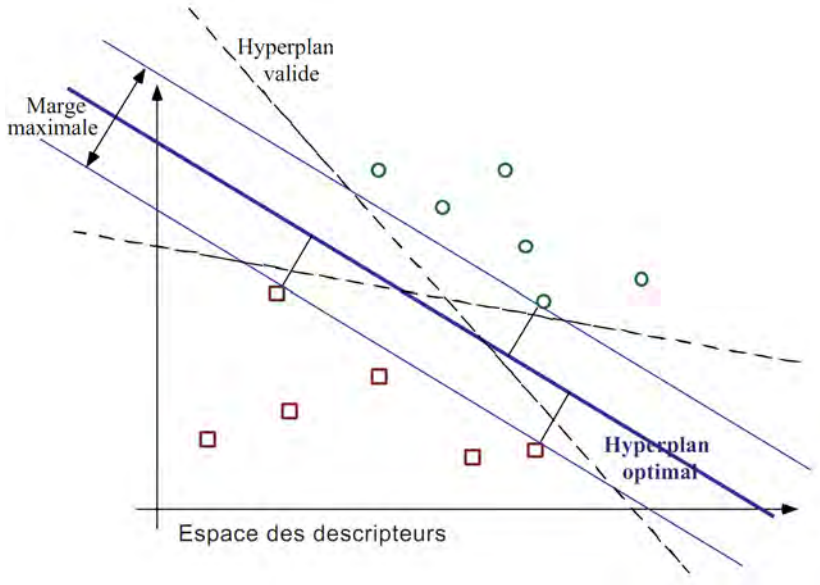

Figure 2.6 – Illustration de la droite de séparation optimale recherchée par les SVM dans le cas d'un problème à deux dimensions, d'après [Cornuéjols et Miclet, 2011]

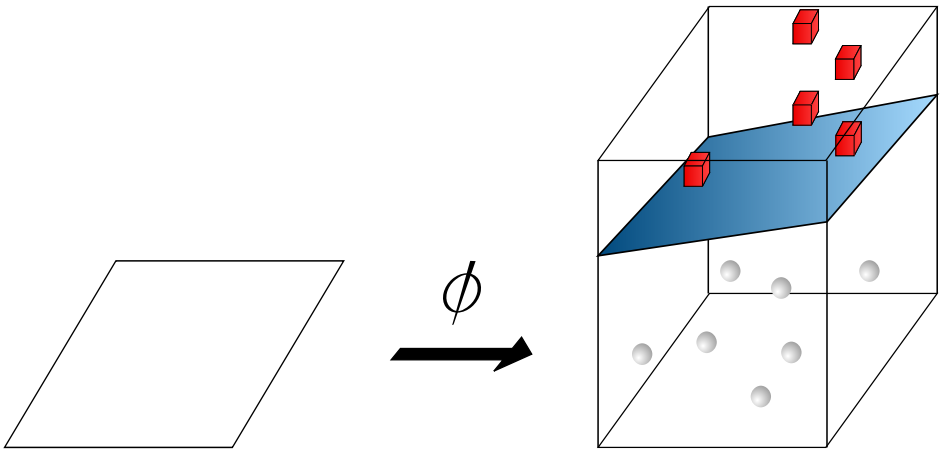

Figure 2.7 – Illustration de l'effet de l'ajout de dimension supplémentaire dans la recherche d'un hyperplan optimal de séparation de classes

Dans le cas présent, la fonction ϕ est idéale, puisqu'elle retourne directement des exemples linéairement séparables. Il faut cependant comprendre que, même en prenant une fonction au hasard, nous tendons nécessairement vers une meilleure séparation. En effet, dans le meilleur des cas l'ajout d'une dimension aura amélioré la séparation entre les deux classes, et dans le pire des cas elle ne l'aura pas améliorée mais ne l'aura pas non plus dégradée. Ainsi, une transformation amenant les exemples en entrée vers une dimension tendant vers l'infini impliquera nécessairement l'obtention d'un hyperplan séparant les deux classes.

La recherche de cet hyperplan passe par le calcul d'un produit scalaire entre les vecteurs des exemples en entrée. Augmenter les dimensions de ces exemples implique donc une augmentation de la complexité du calcul, et donc des temps de calculs. Il est néanmoins possible

d'éviter cela grâce à l'utilisation de fonctions noyaux. Si nous prenons la fonction de transformation ϕ transformant une donnée de dimension D en dimension D', alors la fonction noyau associée à ϕ est une fonction qui, prenant en entrée une donnée de dimension D, renvoie le même résultat que le produit scalaire de cette donnée en dimension D'. Ainsi, en choisissant judicieusement la fonction de transformation ϕ , il est possible de disposer d'une fonction noyau n'augmentant pas, ou peu, la complexité des calculs tout en utilisant une fonction ϕ permettant d'augmenter fortement la dimension des données. En général, il s'agit plutôt de trouver une fonction noyau de complexité identique ou légèrement supérieure au problème initial, et d'en déduire une fonction ϕ vers un espace de dimension plus élevée, voir idéalement de dimension tendant vers l'infini.

De multiples mécanismes peuvent ensuite compléter l'utilisation des SVM. L'ajout de variables "ressorts" permet d'assouplir la répartition des exemples, afin de tolérer quelques erreurs de classements en échange d'un résultat moins spécifique. De plus, comme tous les algorithmes de classification binaire, autrement dit ne renvoyant que deux classes possibles, il est possible de les généraliser à des problèmes de classifications non binaires. Cela peut se faire en attribuant un SVM par classe en sortie, la classification d'un nouvel exemple en entrée étant déterminé par le SVM obtenant la plus grande probabilité d'appartenance, ou bien en attribuant un SVM pour chaque paire de classe en sortie, la classe attribuée le plus souvent à un nouvel exemple en entrée étant alors considérée comme la bonne.

Pour conclure, les SVM fournissent non seulement de bonnes performances, mais consolident leurs résultats sur des théories mathématiques. Ils ne sont cependant pas capables de gérer l'évolution de la fonction à apprendre, tout comme ils ne sont pas capables de gérer des exemples de dimensions variables. De plus, les performances des SVM sont très fortement liées aux fonctions noyaux employées, ce qui complexifie leur mise en application en impliquant une étude préliminaire à l'utilisation des SVM pour trouver des fonctions noyaux adaptées.

2.2.1.6 Les réseaux bayésiens

Les réseaux bayésiens, présentés par [Jensen, 1996], ne sont pas en soi des algorithmes d'apprentissage, mais plutôt une représentation de connaissances exprimant des liens de causalité entre des données, et les probabilités associées aux valeurs de ces données. L'ensemble des données est représenté dans un graphe orienté acyclique, où chaque nœud représente une donnée à laquelle est associée une probabilité pour chaque valeur qu'elle peut prendre. Cette probabilité dépend aussi des valeurs des nœuds parents.

La méthode d'utilisation de base d'un réseau bayésien consiste à laisser un expert construire le réseau bayésien et paramétrer les probabilités pour chaque donnée en fonction des valeurs des données-parents. Ce réseau bayésien est alors utilisé pour réaliser des traitements d'inférence de données :

- L'inférence descendante (ou inférence causale) permet d'établir les probabilités de valeurs pour une donnée sachant une ou plusieurs valeurs de ses données parents.
- L'inférence ascendante (ou diagnostic) permet d'établir les probabilités de valeurs pour une données sachant une ou plusieurs valeurs de ses données filles.
- L'explication est une combinaison des deux : connaître par exemple la probabilité de

valeurs pour une donnée sachant, non seulement la valeur d'une de ses données filles, mais aussi la valeur d'une autre donnée parente de cette donnée fille.

La création et le paramétrage d'un réseau bayésien reste cependant une tâche très lourde dès qu'il s'agit de l'appliquer à un problème complexe. Il est cependant possible d'automatiser le paramétrage du réseau à partir d'une liste d'exemples. Établir les probabilités associées aux différentes données de façon à se rapprocher des exemples en entrée est une tâche relativement simple.

En revanche, la tâche devient beaucoup plus complexe dès lors que nous cherchons à déterminer aussi la structure du réseau bayésien à partir des exemples, mais nous pouvons alors réellement parler d'apprentissage. Il existe deux approches générales pour l'apprentissage de la structure et des paramètres d'un réseau bayésien :

- la première méthode consiste à déterminer les dépendances et indépendances entre les données, puis les utiliser pour construire le graphe. Cette méthode est néanmoins sensible aux erreurs dans les tests d'indépendance.
- la seconde méthode consiste à établir une fonction de score pour évaluer la pertinence d'un graphe vis-à-vis des exemples en entrée, et de chercher à construire le graphe obtenant le meilleur score. Parmi les différentes techniques d'exploration heuristiques des graphes candidats, la plus simple est la descente de graduent, qui part d'un graphe vide et applique à chaque itération l'opérateur (ajout, suppression ou inversement de l'orientation d'un arc) qui maximise le résultat obtenu par la fonction de score.

Une étude comparative des algorithmes d'apprentissage de structures de réseaux bayésiens est proposée notamment par [François et Leray, 2003]. Ces algorithmes ne sont cependant pas appropriés pour l'apprentissage d'une fonction susceptible d'évoluer, avec des données qui apparaissent ou disparaissent. Ils fournissent en revanche, une fois l'apprentissage réalisé, un résultat intéressant en terme de compréhension par l'utilisateur.

2.2.1.7 Case-Based Reasoning

Le *Case-Based Reasoning* (CBR), ou raisonnement à partir de cas, est une méthode de résolution de problèmes à travers l'utilisation d'une base de cas. Une introduction à cette méthode est notamment proposée par [Kolodner, 1992]. Le principe général de cette méthode est illustré par la figure 2.8. Un système basé sur le CBR possède une base de connaissances constituée d'un ensemble de cas, chaque cas étant constitué d'une représentation d'une situation donnée d'une part, et de la représentation de la solution adaptée à cette situation de l'autre. Les représentations utilisées dépendent du cadre d'applications, mais une situation est généralement représentée sous la forme simple d'une liste de clés-valeurs.

Le fonctionnement du CBR est divisé en 4 phases :

1. La recherche de cas similaire : le CBR percevant un cas en entrée, il recherche dans sa base de connaissance le cas le plus similaire. La fonction de similarité constitue la première fonction à instancier dans la conception d'un CBR. La solution associée au cas similaire est alors sélectionnée ;
2. L'application de la solution : dans les systèmes les plus simples, la solution trouvée dans la phase précédente peut être appliquée directement au nouveau cas. Cependant, pour des systèmes plus complexes, il convient la plupart du temps d'appliquer un prétrai-

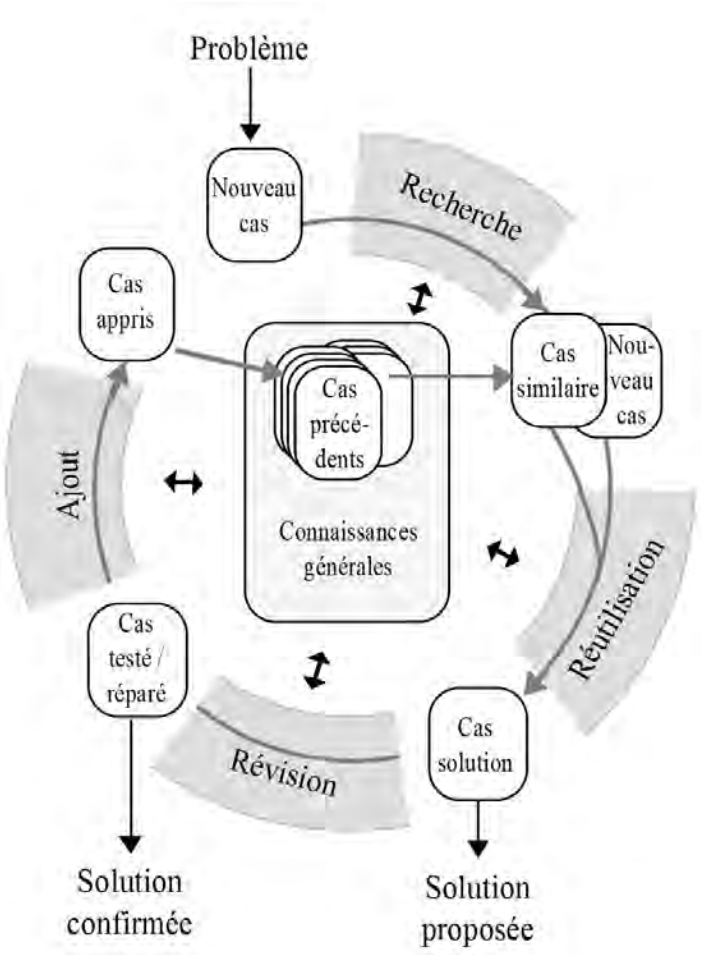

Figure 2.8 – Illustration du fonctionnement du *Case-Based Reasoning*, extrait de [Aamodt et Plaza, 1994]

tement sur cette solution. Cette fonction d'adaptation d'une solution à un cas similaire constitue la seconde fonction du CBR à instancier. La solution produite est alors appliquée ;

3. Révision de solution proposée : Une fois que la solution a été mise en oeuvre, le CBR observe si elle a été efficace ou pas. Cette observation peut faire intervenir un expert confirmant la justesse de la nouvelle solution, ou un système extérieur capable d'évaluer lui-même si la solution proposée est correcte. Si la solution proposée n'est pas satisfaisante, le CBR va tenter de comprendre les raisons de cet échec, et éventuellement de modifier son traitement (au niveau de sa fonction de détection de similarité ou de celle d'adaptation d'une solution, par exemple) pour ne pas réitérer son erreur. Ces capacités de compréhension et/ou de réparation d'une erreur constituent la troisième fonction à instancier durant la conception d'un CBR ;

4. Apprentissage : Dans le cas où la solution proposée en phase 2 a été validée durant la phase 3, elle est associée au nouveau cas en entrée. Ce cas est alors ajouté à la base de connaissances du CBR. Cet ajout, qui constitue la quatrième fonction à instancier durant la conception du CBR, ne se limite pas à un simple ajout direct. En effet, plusieurs

traitements doivent être réalisés pour éviter les connaissances redondantes (et ainsi éviter d'exploser la taille de la base de connaissances), ou contradictoires (ce qui implique de déterminer si un cas doit être conservé ou supprimé).

Le CBR constitue une approche générale de conception d'un système capable d'enrichir ses connaissances en cours de fonctionnement. Il ne s'agit donc pas d'un algorithme d'apprentissage en soi, mais plutôt d'une méthode pour la conception d'un système utilisant des algorithmes d'apprentissage. Les performances d'un CBR dépendent donc de l'instanciation faite des fonctions de chacune des quatre étapes de son fonctionnement. En particulier, l'évaluation du résultat obtenu lorsqu'une solution est retenue permet de mettre à jour un apprentissage réalisé préalablement, et donc de gérer l'aspect évolutif de la fonctionnalité à apprendre. Concernant la prise en compte des variations dans les données perceptibles, elle est dépendante de la modélisation des cas, et surtout de l'instanciation de la fonction de similarité.

2.2.2 L'apprentissage par renforcement

Un algorithme d'apprentissage par renforcement est un apprentissage *en ligne*. A chaque cycle, il perçoit l'état courant de son environnement ainsi que la liste des actions qu'il peut réaliser. Il choisit alors une de ces actions, puis au cycle suivant, il perçoit le nouvel état de l'environnement ainsi qu'une valeur de récompense lui permettant d'évaluer la qualité du choix de son action à l'état précédent.

L'objectif d'un algorithme d'apprentissage par renforcement est alors de maximiser le gain au cours du temps. Ce gain peut être évalué de plusieurs façons : gain cumulé, gain en moyenne, etc. Cependant, l'algorithme n'ayant pas de connaissance *a priori* sur les effets de ses actions, il peut déterminer la récompense qu'il recevra en retour qu'en essayant une action pour un état donné. Cette récompense n'étant pas forcément la même à chaque fois, la connaissance sur la meilleure action à réaliser pour un état donné requiert une phase d'exploration, où l'algorithme doit tenter chaque action un certain nombre de fois. En phase d'exploitation, l'algorithme exploite les connaissances acquises pour choisir l'action ayant la meilleur espérance de gain. Trouver le juste milieu entre la phase d'exploration et la phase d'exploitation est une des problématiques principales des algorithmes de renforcement.

Le modèle construit par un algorithme d'apprentissage par renforcement est appelé *politique*. Il détermine quelle action réaliser en fonction de l'état courant. L'algorithme ne recherche cependant pas le gain immédiat le plus élevé, mais bien le gain le plus important à long terme. La valeur locale associée à chaque couple d'état-action représente l'espérance de gain à long terme en suivant une certaine politique ; elle est appelée *utilité*. La fonction qui retourne la valeur de l'utilité pour chaque couple d'état action en suivant la même politique est appelée *fonction d'utilité*. La politique optimale est celle qui permet, quelque soit l'état de l'environnement, de déterminer l'action dont l'espérance de gain est la plus élevée.

L'utilisation d'un algorithme d'apprentissage par renforcement peut alors se faire dans trois cas différents :

1. L'algorithme possède un modèle de son environnement, et possède aussi la fonction d'utilité de la politique optimale. L'algorithme n'a alors plus qu'à utiliser le modèle de

l'environnement pour observer, pour l'état courant, quel sera l'état suivant (plusieurs états possibles en cas d'environnement non déterministe, avec une probabilité pour chaque état), et utiliser ensuite la fonction d'utilité pour choisir l'action qui mènera à l'état possédant la meilleure utilité. Dans ce premier cas, la politique apprise tend à devenir très vite la politique optimale, car toutes les connaissances nécessaires (l'environnement et la fonction d'utilité) sont déjà possédées par l'algorithme.

2. L'algorithme possède un modèle de son environnement qui lui permet de connaître les effets de chaque action possible pour un état donné, mais n'a pas de fonction d'utilité l'informant du gain espéré à long terme. L'algorithme cherche donc à déterminer une fonction d'utilité pour sa politique, tout en essayant d'améliorer cette politique. Ce double traitement est complexe, car l'évolution de la politique entraîne nécessairement un changement dans la fonction d'utilité.
3. L'algorithme doit fonctionner sans connaître ni l'environnement, ni la fonction d'utilité. Dans ce cas, l'algorithme doit aussi estimer empiriquement l'espérance de gain de chaque action en fonction de chaque état. Dans le cas d'un environnement non déterministe, cette estimation s'améliore à chaque itération. C'est dans ce troisième cas que l'alternance entre la phase d'exploration et la phase d'exploitation prend toute son importance.

Un des intérêts des algorithmes par renforcement pour l'apprentissage du comportement de l'utilisateur en environnement ambiant est qu'il s'agit d'un apprentissage en ligne, ce qui les rend capables d'apprendre en continu en fonction des observations réalisées. Le Q-learning, présenté dans la section suivante, a d'ailleurs déjà été utilisé dans cet optique par [Zaidenberg, 2009], dans le système présenté en section 1.2.2.11. Malgré des résultats intéressants, ces algorithmes se heurtent à la limite du nombre de variables qui ne peut évoluer en cours de fonctionnement, ce qui limite leur utilisation dans le cadre d'environnement ambiant.

2.2.2.1 Q-learning

L'algorithme du Q-learning ([Watkins, 1989]) se base sur une méthode *hors politique*, car l'apprentissage de la fonction d'utilité est réalisé indépendamment de la politique courante π. Cette fonction d'utilité Q est évaluée par itération : chaque fois que l'algorithme réalise une action a dans un état s, il met à jour la qualité estimée $Q(s,a)$ de l'action a à l'état s en fonction de la valeur courante de $Q(s,a)$ et de la récompense obtenue r. Cette mise à jour est basée sur la formule suivante :

$$Q(s,a) = (1-\alpha).Q(s,a) + \alpha.(r + \gamma.\max_{x \in \mathcal{A}} Q(s,x))$$

où $Q(s,a)$ la qualité estimée de l'action a à l'état s quelle que soit la politique suivie, α est le taux d'apprentissage représentant l'impact de la nouvelle valeur sur la valeur retournée par $Q(s,a)$, r le gain reçu suite à la réalisation de l'action a, γ le taux de diminution des renforcements, et $\max_{x \in \mathcal{A}} Q(s,x)$ la qualité la plus élevée qu'il est possible d'obtenir à l'état s grâce à une des actions réalisables.

La nouvelle valeur attribuée est influencée par le gain reçu et par l'estimation de la meilleure qualité qui peut être obtenue à l'état suivant. En effet, ce n'est pas le gain immédiat qui représente la qualité d'une action dans un état, mais bien le gain espéré à long terme. Par conséquent, la qualité de la future action dans le futur état influence aussi la qualité de l'action courante à l'état courant.

L'alternance entre la phase d'exploration et la phase d'exploitation est réalisée par une procédure dite ε-gloutonne : l'action sélectionnée par l'algorithme est généralement celle qui propose la meilleure qualité, sauf lorsqu'avec une probabilité de ε l'algorithme décide de sélectionner au hasard une autre action. Ce paramètre ε peut être fixé, ou varier en fonction d'autres paramètres (une valeur classique est $\varepsilon = 1/t$, qui diminue les probabilités d'exploration au cours du temps).

2.2.2.2 SARSA

Contrairement à l'algorithme du Q-learning, l'algorithme SARSA de [Sutton et Barto, 1998] se base sur une méthode *sur politique*, car la politique courante π de l'algorithme influence en continu l'apprentissage de la fonction d'utilité. Cette fonction est aussi évaluée par itération, mais suivant cette fois la formule suivante :

$$Q^{\pi}(s,a) = (1-\alpha).Q^{\pi}(s,a) + \alpha.(r + \gamma.Q^{\pi}(s,a))$$

où π représente la politique courante, $Q^{\pi}(s,a)$ la qualité estimée de l'action a à l'état s suivant la politique π, α est le taux d'apprentissage représentant l'impact de la nouvelle valeur sur la valeur retournée par $Q^{\pi}(s,a)$, r le gain reçu suite à la réalisation de l'action a, et γ le taux de diminution des renforcements.

La méthode employée est très similaire à celle du Q-learning, l'alternance entre exploration et exploitation étant aussi gérée par une procédure ε-gloutonne.

2.3 Filtrage de données

Les performances d'un algorithme d'apprentissage est fortement dépendant du nombre de variables en entrées. De façon générale, plus le nombre de variables en entrée est grand, plus les traitements des algorithmes d'apprentissage sont conséquents. Tous les algorithmes d'apprentissage ne sont pas égaux face à ce phénomène : les SVM par exemple, sont beaucoup moins sensibles à l'augmentation des données que les réseaux de neurones.

Ce phénomène devient un problème lorsque, du fait du domaine d'application, un algorithme se retrouve à avoir plus de variables en entrée que nécessaire pour son traitement. Si nous considérons l'ensemble des données D perçu par un algorithme d'apprentissage, nous pouvons alors voir apparaître deux cas de figures :

1. Il existe une donnée $d \in D$ dont la suppression n'entraînerait aucune diminution des performances de l'algorithme d'apprentissage, voire permettrait de les améliorer. d est donc une variable *inutile*.

2. Il existe deux données $d_1, d_2 \in D$ pour lesquelles il est possible d'en supprimer une sans que cela n'entraîne de diminution des performances de l'algorithme d'apprentissage, alors que la suppression simultanée des deux données diminuerait les performances de l'algorithme. d_1 et d_2 sont donc des variables utiles mais redondantes.

Une illustration très explicite des effets de l'inutilité d'une donnée sur un algorithme d'apprentissage est donnée par [Delalleau, 2008]. La figure 2.9 montre un exemple où l'algorithme d'apprentissage tente d'apprendre à classifier des informations en deux classes. Si l'algorithme d'apprentissage observe les points A et B en utilisant uniquement la donnée x_1 (représentés par leurs projection A' et B' sur l'axe x_1), il trouvera correctement les deux classes. La séparation entre ces deux classes est représentée par la ligne verticale en pointillé. Cependant, s'il considère la donnée inutile x_2, il ne réussira pas à séparer les deux classes, sa séparation incorrecte étant représentée par la ligne oblique continue. Ainsi, prendre en compte une donnée inutile dans un processus d'apprentissage implique une augmentation du nombre d'exemples nécessaires à fournir à l'algorithme d'apprentissage pour surmonter ce problème.

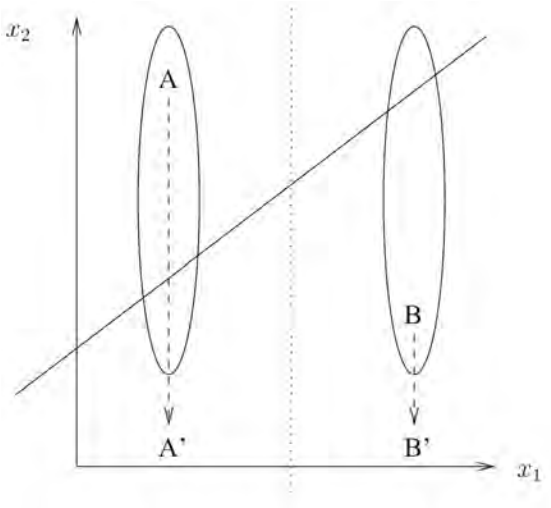

Figure 2.9 – Illustration de l'effet des données inutiles sur un algorithme d'apprentissage (de [Delalleau, 2008])

Pour résoudre ce problème, il devient nécessaire de filtrer les données inutiles ou redondantes, soit en intégrant les données au fur et à mesure pour vérifier que leur ajout augmente les performances (sélection des données utiles et non redondantes), soit en les supprimant au fur et à mesure pour vérifier que leur suppression ne diminue pas les performances (filtrage des données inutiles ou redondantes). Dans les deux cas, le problème reste globalement le même : sur quel critère estimer la pertinence d'une donnée pour un algorithme d'apprentissage ? Comment évaluer le choix de filtrer ou pas une donnée, tout en s'assurant que ce processus d'évaluation ne soit pas lui-même plus lourd à réaliser que le gain qu'il fournit ?

Plusieurs approches ont été proposées dans la littérature pour résoudre ce problème. Sans entrer dans les détails de leur fonctionnement, nous présentons ici les quatre catégories principales existantes, à savoir les méthodes d'ordonnancement de variables, les méthodes

filter, les méthodes *wrapper* et les méthodes *embedded*. Selon l'approche proposée, nous pourrons parler aussi bien de sélection de données utiles que de filtrage de données inutiles, le résultat produit étant au final le même, c'est-à-dire l'ensemble des données utiles au processus d'apprentissage.

2.3.1 Ordonnancement de variables

Les méthodes basées sur le classement des variables essaient d'attribuer un score pour chaque donnée. Ce score représente l'utilité de cette donnée vis-à-vis de l'objectif à apprendre. L'évaluation de cette utilité est réalisée pour chaque donnée indépendamment des autres ; les données obtenant un score trop faible sont éliminées. Dans le cas d'un algorithme d'apprentissage appliqué à la classification, c'est la capacité à permettre de prédire la valeur d'une donnée en sortie en se basant uniquement sur cette donnée en entrée qui sert à évaluer l'utilité de cette donnée en entrée.

[Guyon et Elisseeff, 2003] énoncent plusieurs méthodes basées sur l'ordonnancement de variables. Une de ces méthodes se base sur le coefficient de corrélation de Pearson. Si nous considérons X_i la variable aléatoire correspondant au i^{me} composant du vecteur des données en entrée x et Y la variable aléatoire correspondant à la sortie y, le coefficient de corrélation de Pearson est défini pour la i^{me} donnée par la formule suivante :

$$R(i) = \frac{cov(X_i, Y)}{\sqrt{var(X_i) \cdot var(Y)}}$$

La covariance entre deux variables aléatoires permet de quantifier l'indépendance entre ces deux variables. En divisant cette covariance par la racine carrée du produit des variances de ces deux données, nous obtenons alors un coefficient de corrélation normé entre 0 et 1. Cependant, cette formule s'applique à des variables aléatoires dont nous connaissons les lois de distribution. Pour l'appliquer à une base d'exemples, il faut utiliser une estimation empirique du coefficient de corrélation de Pearson qui, étant donnés m le nombre d'exemples, $x_{k,i}$ la valeur de la i^{eme} donnée du k^{eme} exemple, y_k la valeur de la sortie associée au k^{eme} exemple, $\bar{x_i}$ la moyenne de la i^{eme} donnée sur l'ensemble des m exemples, et $\bar{y}$ la valeur moyenne des données en sortie associées à l'ensemble des m exemples, est calculée par la formule suivante :

$$\hat{R}(i) = \frac{\sum_{k=1}^{m}(x_{k,i} - \bar{x_i})(y_k - \bar{y})}{\sqrt{\sum_{k=1}^{m}(x_{k,i} - \bar{x_i})^2 (y_k - \bar{y})^2}}$$

Cette méthode fournit donc une estimation numérique du degré d'indépendance entre une donnée en entrée et la donnée en sortie. Plus cette valeur est élevée, plus les variables sont dépendantes. Il reste alors à ordonner les différents scores calculés pour chacune des données afin d'éliminer les données ayant un score trop faible en ne conservant que les données présentant les meilleurs scores.

Malgré leur simplicité, les méthodes basées sur le classement de variables présentent de gros inconvénients. En effet, même si pour certains problèmes, une évaluation indépendante

Algorithme 2.1 : L'algorithme Relief de [Kira et Rendell, 1992]

Pré-conditions : S : la liste des exemples, chaque exemple $x \in S$ étant composé de p données, $n \in E$: valeur entière constante, $\tau \in [0,1]$: seuil de pertinence
Initialiser $W_i = 0, \forall\, i \in [0;p]$
Pour $i = 1..n$ **Faire**
Sélectionner au hasard $X \in S$
Sélectionner au hasard $Near - hit$, un des plus proches exemples de X appartenant à la même classe
Sélectionner au hasard $Near - miss$, un des plus proches exemples de X n'appartenant pas à la même classe
Pour $d = 1..p$ **Faire**
$W_d = W_d - diff(x_d, Near - hit_d)^2 + diff(x_d, Near - miss_d)^2$
Fin Pour
Fin Pour
Pour $i = 1..p$ **Faire**
$Pertinence_i = \frac{1}{n} \cdot W_i$
Si $Pertinence_i \geq \tau$ **Alors**
d_i est une donnée utile
Sinon
d_i est une donnée inutile
Fin Si
Fin Pour

de l'utilité de chaque donnée peut être suffisante, une telle solution s'avère souvent inadaptée pour des domaines plus complexes. Par exemple, il est possible de se retrouver dans une configuration où deux données sont considérées comme inutiles si elles sont évaluées séparément, alors qu'elles sont utiles si nous les considérons toutes les deux. Par exemple, pour apprendre la fonction booléenne OU EXCLUSIF à partir de deux entrées, la capacité de connaître l'état de la sortie en fonction d'une seule entrée est nulle ; il est nécessaire de tenir compte des deux entrées à la fois pour en juger l'utilité. Un autre problème se pose avec les variables redondantes qui peuvent fournir chacune une capacité à prédire l'état d'une donnée en sortie suffisante pour être jugées utiles, alors qu'une seule de ces données aurait été suffisante pour fournir la même prédiction.

Nous nous intéressons par la suite à d'autres méthodes de filtrage de données inutiles capables de tenir compte des corrélations entre les différentes données.

2.3.2 Méthodes *filter*

Les méthodes de type *filter* s'intéressent à la sélection de données utiles indépendamment de l'algorithme d'apprentissage lui-même. Ces méthodes sont vues comme un prétraitement des données utilisées en entrée du processus d'apprentissage, et ne permettent donc pas d'interagir avec celui-ci. Une des méthodes existantes de type *filter* utilise l'algorithme *Relief* de [Kira et Rendell, 1992].

L'algorithme *Relief* consiste à réitérer n fois (le paramètre n étant fixé par le concepteur) le traitement suivant : il prend au hasard un exemple X, ainsi que deux exemples proches de X, la proximité entre deux exemple étant calculée par la distance euclidienne entre les exemples ; le premier exemple $Near - hit$ appartient à la même classe que X, alors que le second exemple $Near - miss$ appartient à une classe différente. L'algorithme évalue alors l'utilité de chacune des données d comme la moyenne des valeurs W_d calculée aux cours des n itérations, chaque valeur W_d représentant la distance entre la valeur de cette donnée pour le premier exemple x_d et celle de l'exemple de classe différente $Near - miss_d$ moins la distance entre la même valeur x_d et celle de l'exemple de même classe $Near - hit_d$.

Lorsqu'une donnée d est utile, il est attendu que la moyenne des valeurs W_d tende à augmenter au cours des différentes itérations, car les valeurs d'une donnée tendent à être davantage similaires entre deux exemples de même classe qu'entre deux exemples de classes différentes. Par contre, si la donnée d n'est pas utile, les différences de valeurs pour cette donnée tendent à être globalement les mêmes que les deux exemples soient de même classe ou non, faisant alors tendre la moyenne des valeurs W_d vers zéro.

En se basant sur un seuil τ fournie par le concepteur, l'algorithme termine son traitement en comparant les valeurs calculées avec ce seuil, considérant toute donnée au dessus de ce seuil comme pertinente, toute valeur en dessous du seuil comme non pertinente. Ainsi, ce traitement permet d'évaluer la pertinence des différentes données indépendamment de l'algorithme utilisé, sans besoin d'intégrer de fortes connaissances *a priori* (à l'exception du seuil fixé à la main). Enfin, sa complexité est linéaire ($\Theta(p+n)$, p étant le nombre de donnée en entrée d'un exemple, et n le nombre d'exemples de S intervenant dans la recherche des exemples les plus proches, dans la même classe ou pas).

Comme l'illustre cet exemple, les méthodes de type *filter* permettent généralement de réaliser un filtrage des données inutiles sans impliquer de calculs de forte complexité. Elles sont de plus indépendantes de l'algorithme d'apprentissage, ce qui les rend plus facilement applicables, mais diminue les chances d'obtenir le sous-ensemble optimal des données pertinentes.

2.3.3 Méthodes *wrapper*

Les méthodes de type *wrapper* interagissent avec l'algorithme d'apprentissage afin d'évaluer l'utilité de sous-ensembles de données. Pour cela, l'algorithme de sélection de données considère l'algorithme d'apprentissage comme une boîte noire, qu'il entraîne avec des sous-ensembles de données, pour évaluer les performances de l'algorithme d'apprentissage avec chacun de ces sous-ensembles, et ainsi déterminer le sous-ensemble permettant d'obtenir les meilleures performances. Un tel algorithme permet donc d'interagir avec l'algorithme d'apprentissage pour améliorer ses performances, sans s'intéresser pour autant au choix de l'algorithme d'apprentissage.

Bien sûr, évaluer les performances de l'algorithme d'apprentissage pour la totalité des sous-ensembles de données assure d'obtenir le sous-ensemble optimal, mais un tel processus serait alors excessivement lourd dès lors que le nombre de données devient conséquent. Différentes stratégies peuvent cependant être utilisées pour rechercher plus efficacement le

sous-ensemble des données optimales.

L'algorithme d'OBLIVION proposé par [Langley et Sage, 1994] appartient à la catégorie des méthodes de sélection des données pertinentes de type *wrapper*. Il consiste à parcourir un arbre où chaque nœud contient un ensemble de données associées aux performances produites par l'algorithme d'apprentissage à partir des exemples limités à ces données. Le passage d'un nœud père à un de ses nœuds fils se fait par la suppression d'une des données.

Initialement, le nœud racine contient la totalité des données présentes. OBLIVION applique alors l'algorithme d'apprentissage sur les exemples pour déterminer les performances de cet algorithme. Cette évaluation utilise la méthode de la validation croisée *k-way* : l'ensemble des exemples en entrée de l'algorithme d'apprentissage est séparé en k sous-ensembles, puis un des sous-ensembles est sélectionné comme ensemble de validation, et les (k-1) autres sous-ensembles sont utilisés pour réaliser le processus d'apprentissage. Le résultat de l'apprentissage est alors évalué sur le sous-ensemble de validation. Ce traitement est réalisé k fois afin que chaque sous-ensemble soit utilisé une fois comme sous-ensemble d'évaluation. La moyenne du nombre d'erreurs pour chaque évaluation permet alors d'estimer les performances globales de l'algorithme d'apprentissage.

Une fois les performances de l'algorithme d'apprentissage évalué pour l'ensemble des données du nœud racine, l'algorithme OBLIVION construit les nœuds fils : chaque nœud contient l'ensemble des données du père moins une. Il y a donc autant de nœud fils générés que de données dans le nœud père. OBLIVION applique alors le processus d'évaluation de l'algorithme d'apprentissage pour chaque nœud en considérant le sous-ensemble des données proposé par chaque nœud. Dès lors, le nœud fournissant la meilleure performance devient le nœud courant. OBLIVION réitère alors son traitement de génération de nœuds fils, d'évaluation des nœuds fils générés et de sélection du meilleur nœud comme le nouveau nœud courant. Ce traitement se poursuit jusqu'à ce que même le meilleur nœud retourne une performance inférieure à celle du nœud père. L'ensemble des données fournies par ce dernier est alors considéré comme l'ensemble des données utiles optimales.

Cette méthode de parcours des sous-ensemble fournit bien sûr de bien meilleurs résultats qu'une recherche exhaustive des sous-ensembles possibles, et assure de retourner un résultat performant. En revanche, la critique principale portée aux méthodes de type *wrapper* est qu'elles impliquent un fort coût computationnel. Ici, notamment, si nous prenons d comme le nombre de données en entrée de chaque exemple, et k la valeur fixée pour la validation croisée *k-way*, le processus peut requérir une évaluation de l'algorithme d'apprentissage jusqu'à $k \cdot \frac{d \cdot (d+1)}{2}$ fois.

2.3.4 Méthodes *embedded*

La dernière catégorie de méthode de filtrage de données est celle des méthodes *embedded*. Alors que les méthodes de type *filter* réalisent leurs traitements en amont du traitement d'apprentissage, et que les méthodes de type *wrapper* interagissent avec l'algorithme d'apprentissage, les méthodes de type *embedded* sont directement incluses dans l'algorithme d'apprentissage, et sont appliquées en même temps que le processus d'apprentissage lui-même. Par conséquent, il n'y a pas une méthode générale de filtrage *embedded*, car chacune

d'elles est fortement dépendante du fonctionnement de l'algorithme d'apprentissage.

Nous présentons ici, à titre d'exemple, une méthode de filtrage proposée par [Weston *et al.*, 2003] et appliquée aux SVMs (voir section 2.2.1.5) pour l'apprentissage de problèmes linéairement séparables (c'est-à-dire, ceux ne requérant pas d'être étendus à des dimensions supérieures pour devenir linéairement séparables). Dans le processus d'apprentissage des SVM, l'objectif est de déterminer, pour un problème de dimension D, l'hyperplan de dimension (D-1) optimal pour classifier correctement l'ensemble des exemples. Cet hyperplan s'écrit alors, pour tout $d_i \in D$ sous la forme $w_0 + \sum_{i=1}^{D-1} w_i \cdot d_i = 0$, le vecteur $W = (w_1, ..., w_{D-1})$ représentant alors le vecteur directeur de l'hyperplan.

La méthode de filtrage consiste donc à calculer cet hyperplan, puis à recalculer l'ensemble des exemples en les multipliant par les valeurs absolues des composantes du vecteur W. A force de réitérer ce processus, les composantes ayant le moins d'impact dans le calcul des hyperplans tendent progressivement vers zéro. Ce processus se termine lorsque le nombre de données à filtrer, autrement dit le nombre de données dont les composantes associées dans le vecteur directeur de l'hyperplan solution sont à zéro, est en dessous d'une valeur fixée. Cette méthode est donc capable de fournir une solution combinant une séparation de classes la plus juste possible tout en minimisant le nombre de données en entrée des exemples.

Il est évident que cette méthode ne peut s'appliquer que pour ce cas précis d'algorithme d'apprentissage. Les méthodes *embedded* s'enrichissent donc des spécificités de l'algorithme d'apprentissage pour lequel elles ont été conçues, mais ne sont en revanche pas du tout applicables à d'autres.

2.3.5 Synthèse

Dans cette section, nous avons présenté les quatre catégories principales du filtrage de données : ordonnancement de variables, *filter*, *wrapper* et *embedded*. Les méthodes d'ordonnancement et les méthodes *filter* établissent la pertinence des différentes données *a priori*, en prétraitement de l'algorithme d'apprentissage, en observant uniquement les exemples et sans interaction avec l'algorithme d'apprentissage. Les méthodes *filter* considèrent l'ensemble des données pour tenter d'établir le sous-ensemble optimal de données pertinentes au lieu d'étudier la pertinence des données une à une. De telles méthodes peuvent fournir de bons résultats sans nécessiter pour autant de traitements trop importants. Cependant, ces méthodes ne sont pas toujours applicables, par exemple dans le cas d'apprentissage en ligne, et leurs résultats ne sont généralement pas optimaux.

Les méthodes *wrapper*, en revanche, tendent à fournir de meilleurs résultats, voire des résultats optimaux, grâce à leurs interactions directes avec l'algorithme d'apprentissage utilisé. De plus, le choix de l'algorithme d'apprentissage peut généralement se faire indépendamment du traitement de filtrage, ce qui rend ces méthodes facilement applicables à de nombreux problèmes. Néanmoins, comme ces méthodes étudient les performances de l'algorithme d'apprentissage en fonction de différents sous-ensembles de données à évaluer , elles impliquent des traitements importants. De plus, cette exploitation continue de l'algorithme d'apprentissage rend difficile l'utilisation de ces méthodes de filtrage dans le cas des

algorithmes d'apprentissage en ligne.

Il est en revanche plus difficile d'évaluer les performances des méthodes *embedded*, car celles-ci sont très liées au fonctionnement de l'algorithme d'apprentissage dans lequel elles sont intégrées. Cependant, dans le cadre de la conception d'un algorithme d'apprentissage, il est intéressant d'envisager également la conception une méthode *embedded* adapté à ce nouvel algorithme. En effet, enrichir un algorithme d'apprentissage avec une méthode *embedded* peut permettre d'obtenir un filtrage de données supérieur à celui de méthodes plus génériques.

2.4 Discussion

Dans cette seconde partie de l'état de l'art, nous nous sommes intéressés aux différents algorithmes d'apprentissage existants afin d'étudier si l'un d'eux semblait directement applicable à notre problématique, à savoir l'apprentissage du comportement à attribuer à un système ambiant à partir de l'observation des actions des utilisateurs. En particulier, un algorithme d'apprentissage devrait fonctionner avec des données perceptibles qui apparaissent et disparaissent en cours d'activité. Il doit aussi être capable d'ajuster la fonctionnalité apprise en cas de changement de préférences des utilisateurs.

Deux catégories d'algorithmes peuvent être distinguées : les algorithmes d'apprentissage *en ligne* qui interagissent en continu avec leur environnement respectif, et les algorithmes d'apprentissage *hors ligne* qui requièrent une liste d'exemples sur laquelle réaliser leur apprentissage. Étant données les caractéristiques des systèmes ambiants, un apprentissage *hors ligne* (qui implique une phase d'apprentissage puis une phase d'application) nous semble inadapté. En effet, pour tenir compte des changements dans l'environnement, il est au moins nécessaire de relancer régulièrement l'apprentissage pour tenir compte des nouveautés (enregistrées sous forme de nouveaux exemples). Se pose alors un certain nombre de questions : à quelle fréquence relancer ce traitement ? A fréquence régulière ou juste en cas de contradiction d'un utilisateur ? Comment détecter les contradictions ? De plus, avec le temps, le nombre d'exemples augmente, ce qui implique un traitement de plus en plus lourd pour l'algorithme d'apprentissage. Il est donc nécessaire d'élaguer les exemples enregistrés, mais sur quels critères ? Les plus anciens, au risque d'effacer des règles spécifiques à certaines situations non rencontrées récemment mais néanmoins pertinentes ? Tous ces choix peuvent avoir une influence très forte sur les performances de l'algorithme d'apprentissage, et sont difficilement appréhendables de façon automatique.

L'avantage des algorithmes *en ligne* est qu'ils ne nécessitent pas d'enregistrer l'ensemble des exemples rencontrés, ni de déterminer quand relancer l'apprentissage, celui-ci étant réalisé en continu. Les algorithmes par renforcement notamment s'avèrent très intéressants pour notre problématique, ce qui explique qu'ils aient déjà été appliqués précédemment par [Zaidenberg, 2009] (voir section 1.2.2.11) dans le cadre des systèmes ambiants. Cependant, cette approche présente selon nous deux inconvénients. Le premier est l'incapacité pour la plupart de ces algorithmes de continuer à fonctionner si des changements dans les données perceptibles apparaissent (apparition ou disparition d'un dispositif par exemple, dans le cadre d'une application en système ambiant). Le second est que, contrairement à l'ap-

prentissage supervisé qui imite les utilisateurs à partir d'exemples passés, l'apprentissage par renforcement se base sur un enchaînement d'essais/erreurs, ce qui à notre sens risque d'entraîner une gêne pour les utilisateurs le temps que le bon comportement soit trouvé.

Il nous semble que réaliser des actions incorrectes est plus gênant encore que de ne pas agir du tout. C'est pourquoi, nous préférons une approche basée sur l'observation du comportement de l'utilisateur plutôt que sur un enchaînement d'essais/erreurs pour déterminer quelles actions réaliser dans les différentes situations qui se présentent. Un apprentissage *en ligne* capable d'observer et d'apprendre le comportement des utilisateurs pour progressivement prendre la main sur le système et réaliser les actions à la place des utilisateurs serait donc idéal.

Étant donnés nos besoins et contraintes très précis, il reste peu d'algorithmes d'apprentissage pertinents. L'approche du CBR pourrait nous fournir le résultat recherché, car il possède bien un fonctionnement supervisé *en ligne*. Cette approche a déjà été appliquée dans le cadre des systèmes ambiants, à travers le système de [Tapia *et al.*, 2008] présenté en section 1.2.2.10. Il s'agit d'une approche générique ; pour être appliquée au domaine de l'ambiant, et respecter les critères attendus, un important travail doit être réalisé pour instancier les fonctions des différentes étapes du CBR. Nous recherchons donc une alternative à ces algorithmes afin de concevoir un système d'apprentissage du comportement utilisateur adapté aux spécificités des environnements ambiants.

Deuxième partie

Contribution à l'apprentissage du comportement des utilisateurs en système ambiant par approche AMAS

3 Amadeus : système multi-agent adaptatif pour l'apprentissage contextuel de comportement

L'OBJECTIF de notre étude porte sur la conception d'un système capable de rendre en temps réel un environnement ambiant sensible au contexte. Comme nous l'avons montré dans la section précédente, un tel système doit être ouvert et distribué, intégrer un modèle de données générique et expressif, posséder un comportement proactif et explicable, et assurer la confidentialité des données perçues.

La littérature du domaine propose un certain nombre de gestionnaires de contexte capables d'assurer une partie de ces traitements. De tels systèmes ont pour objectif de collecter des informations de contexte, d'en assurer la gestion et la mise à disposition à des applications, mais ne prennent pas en compte l'aspect proactivité de la sensibilité au contexte. Ils sont en revanche capables de gérer la distribution des données perçues depuis des dispositifs vers l'ensemble des dispositifs de l'environnement intéressés par ces données. De plus, certains de ces gestionnaires sont capables d'assurer la confidentialité des données perçues vis-à-vis des différents utilisateurs, de modéliser différents types de données avec une forte expressivité sémantique, et d'assurer l'intégration ou la disparition de dispositifs en cours de fonctionnement. De tels outils fournissent donc une base exploitable pour atteindre notre objectif.

Nous nous intéressons donc à la conception d'un système capable d'être couplé à un gestionnaire de contexte pour que le système ambiant adopte un comportement proactif satisfaisant pour les utilisateurs. Pour cela, ce système sensible au contexte devra faire évoluer son comportement en continu au travers de l'apprentissage du comportement des utilisateurs.

Les algorithmes d'apprentissages existants dans la littérature ont été appliqués à de nombreux domaines d'application pour lesquels ils ont fourni de très bons résultats. Malheureusement, la plupart de ces algorithmes sont inadaptés à notre problématique (voir chapitre 2), du fait de leur incapacité à :

- intégrer les évolutions de la fonctionnalité à apprendre (du fait des changements de préférences des utilisateurs) ;
- prendre en compte la dynamique de l'environnement, qui entraîne une évolution en cours de fonctionnement du nombre de variables présentes en entrée et en sortie de

l'algorithme d'apprentissage ;
– limiter les perturbations vis-à-vis des utilisateurs, notamment par la réalisation d'actions inappropriées.

Pour répondre à ce constat, trois possibilités s'offrent à nous :

1. Améliorer un des algorithmes d'apprentissage existants pour le rendre plus adapté aux contraintes spécifiques des systèmes ambiants ;
2. Intégrer un algorithme d'apprentissage dans un système en charge de gérer les évolutions de l'environnement afin de relancer l'apprentissage en cas de besoin. Cette solution est assimilée à la conception d'un CBR (voir section 2.2.1.7). En effet, la représentation des cas, la recherche du cas le plus proche, et surtout la révision des cas appris passent souvent par un algorithme d'apprentissage qui est intégré au CBR. Ainsi, bien que normalement un arbre de décision soit incapable d'évoluer au cours du temps, il peut être réappris régulièrement par un CBR qui, par observation des retours de son environnement, considère qu'un réapprentissage est nécessaire ;
3. Concevoir intégralement un nouveau système capable d'apprentissage répondant aux contraintes particulières des systèmes ambiants.

Chacune de ces possibilités est susceptible de fournir de bons résultats, et il n'y a pas *a priori* de choix meilleur que les autres. Cependant, dotés d'une certaine expertise vis-à-vis de l'approche par AMAS, nous avons jugé intéressant de tenter de concevoir un nouveau système d'apprentissage au travers de cette approche. Cette solution est originale puisqu'elle propose une démarche nouvelle pour les systèmes ambiants.

Dans ce chapitre, après avoir présenté les grands principes de l'approche par AMAS, nous détaillons son instanciation au problème de l'apprentissage du comportement des utilisateurs dans les systèmes ambiants. Cette instanciation a donné lieu à la définition et à la conception du système multi-agent adaptatif *Amadeus*.

3.1 Les Systèmes Multi-Agent Adaptatifs

L'approche par *Adaptive Multi-Agent Systems* ([Georgé *et al.*, 2011]), ou approche par AMAS, permet la conception de systèmes multi-agents dont l'objectif est de résoudre des problèmes complexes, incomplètement spécifiés, et pour lesquels il n'existe pas de solution algorithmique connue *a priori*. Selon cette approche, le concepteur définit le comportement local de chacun des agents composant le système jusqu'à l'obtention d'une organisation globale de ces agents produisant une fonction collective adéquate, autrement dit une fonction considérée comme satisfaisante par un observateur extérieur. Cette organisation entre les agents résulte des interactions entre le système multi-agent et son environnement, et constitue une réponse adéquate aux événements imprévus. L'approche par AMAS est une approche organisationnelle permettant la construction de systèmes multi-agents qui s'auto-adaptent continuellement et localement à la dynamique de leur environnement.

Dans ce chapitre, nous présentons les différentes notions liées aux systèmes multi-agents à fonctionnalité "émergente", c'est-à-dire les notions d'agents, d'environnement et d'émergence. Nous présentons ensuite plus en détail l'approche par AMAS.

3.1.1 Définitions

3.1.1.1 Agent et multi-agent

Une définition classique du concept d'*agent* est donnée par [Ferber, 1995], qui le décrit comme une "*entité virtuelle ou réelle qui est capable d'agir sur son environnement, qui possède des moyens de perception et de représentation partielle de son environnement, qui est capable de communiquer avec d'autres agents et qui est autonome dans sa prise de décision*".

Un agent se caractérise donc par les propriétés suivantes :
- il évolue dans un environnement (physique ou virtuel) depuis lequel il perçoit des informations et sur lequel il est capable d'agir ;
- il est autonome, ce qui signifie qu'il possède une représentation interne de son environnement auquel les autres agents n'ont pas accès, grâce à laquelle il décide seul de ses actions sans l'intervention d'un utilisateur extérieur ou d'un autre agent ;
- il est capable d'interagir avec d'autres agents, soit directement, soit par envoi de messages, soit par inscription dans l'environnement.

Un système multi-agent est alors composé d'un ensemble d'agents interagissant les uns avec les autres, chaque agent possédant son propre objectif et ses propres connaissances et compétences. En général, aucun agent ne possède à lui tout seul l'ensemble des connaissances et compétences lui permettant de résoudre à lui seul l'objectif global du système, ces connaissances et compétences étant réparties entre les différents agents. Le comportement du système multi-agent résulte des comportements locaux de chacun des agents qui le composent et de leurs interactions. L'observation d'un système multi-agent peut alors se faire selon deux niveaux : (i) le niveau *macro* où l'observateur extérieur au système perçoit la totalité du système en ignorant les interactions internes au système ; (ii) le niveau *micro* où l'observateur perçoit les parties internes du système et leurs interactions. La fonctionnalité globale du système, observée au niveau *macro*, dépend du comportement local des agents qui composent ce système au niveau *micro*, mais aussi de l'organisation de ces agents et de leurs interactions.

3.1.1.2 Environnement

L'environnement d'une entité représente tout ce qui n'est pas cette entité. Du point de vue du système, l'environnement représente donc tout ce qui est extérieur au système. Concernant l'environnement d'un agent au sein d'un système, il inclut l'environnement du système auquel se rajoute le reste du système (incluant les autres agents et leurs connexions). [Russell et Norvig, 1995] proposent quatre caractéristiques qui permettent de décrire l'environnement d'un agent :

- **Accessible / inaccessible** : un environnement est accessible pour un agent si celui-ci est capable de percevoir dans cet environnement la totalité des informations pertinentes pour la réalisation de sa tâche ;
- **Discret / continu** : un environnement discret possède un nombre fini d'états distincts ;
- **Déterministe / non-déterministe** : un environnement est déterministe si son état suite à la réalisation d'une action ne dépend que de l'état courant et de l'action elle-même ;

– **Statique / dynamique** : si malgré l'inactivité d'un agent, son environnement évolue, alors il s'agit d'un environnement dynamique.

3.1.1.3 Émergence

Un phénomène émergent est caractérisé comme étant *plus que la somme de ses parties*, autrement dit le résultat produit par un ensemble d'entités en interaction est davantage que la somme des résultats produits par les différentes entités. [Lewes, 1877] fait notamment la distinction entre le résultant pour lequel la séquence d'étapes qui produisent un phénomène est traçable, et l'émergent ou cela n'est pas possible.

Bien que la notion d'émergence ne possède pas de définition consensuelle, la littérature propose un certain nombre de définitions de ce phénomène. En particulier, pour [De Wolf et Holvoet, 2005], un système produit un phénomène émergent lorsqu'il y a des propriétés cohérentes au niveau macro qui surviennent des interactions entre les parties au niveau micro, et que ces propriétés sont nouvelles au regard des parties individuelles du système. [Georgé, 2003] caractérise plus précisément un phénomène émergent comme :

– un phénomène ostensible : il s'impose à l'observateur au niveau *macro* ;
– un phénomène radicalement nouveau : il est imprévisible à partir de l'observation du niveau *micro* ;
– un phénomène cohérent et corrélé : il a une identité propre mais liée aux parties de niveau *micro* ;
– un phénomène produisant une dynamique particulière : le phénomène n'est pas prédonné, il s'auto-maintient.

La connaissance de ce type de phénomène n'est pas nouveau, comme le montre notamment l'étude de [Goldstein, 1999] qui recense des études ayant trait aux phénomènes émergents remontant jusqu'à la Grèce antique. Pour autant, l'idée d'étudier de tels phénomènes pour tenter de générer volontairement d'autres phénomènes émergents est plus récente. En informatique notamment, il s'agit de rechercher des mécanismes et des méthodes permettant de concevoir des systèmes produisant une fonctionnalité émergente. La résolution d'un problème par une fonctionnalité émergente est susceptible d'apporter des solutions nouvelles par rapport aux solutions d'informatiques classiques.

Les caractéristiques des systèmes multi-agents (distribution physique et/ou fonctionnelle des connaissances et des compétences, possibilité d'évolutions dans l'organisation globale des agents, fonction globale résultant des comportements des agents et de leurs interactions les uns avec les autres, etc.) les rendent adaptés à la conception de systèmes à fonctionnalité émergente. L'apparition d'un phénomène émergent dans un système multi-agent est illustrée par la figure 3.1, tirée de [Picard, 2004]. Au niveau *micro*, nous pouvons observer un ensemble d'agents $\{a_1, a_2, ..., a_n\}$, chaque agent a_i produisant une fonctionnalité partielle f_{p_i}, tandis qu'au niveau *macro*, les interactions entre ces agents produisent la fonctionnalité globale f_s du système. Soit $\ominus$ l'opérateur de combinaison entre deux fonctions partielles. La fonction globale du système est la combinaison de toutes les fonctions partielles de ses n agents $f_s = f_{p_1} \ominus f_{p_2} \ominus ... \ominus f_{p_n}$. La fonction globale obtenue dépend donc :

1. de la fonctionnalité partielle f_{p_i} de chaque agent a_i, une modification de cette fonctionnalité entraînant une modification de la fonction globale ;

2. de l'organisation des différents agents. En effet, considérant deux agents a_i et a_j possédant respectivement les fonctionnalités partielles f_{p_i} et f_{p_j}, la fonctionnalité $f_{p_i} \ominus f_{p_j}$ est généralement différente de $f_{p_j} \ominus f_{p_i}$, c'est pourquoi un changement dans l'organisation entre les agents entraîne un changement dans la combinaison des fonctions partielles, et par conséquent dans la fonction globale ;
3. des agents présents dans le système. L'ajout ou la suppression de l'agent a_i dans le système entraîne l'ajout ou la suppression de sa fonction partielle f_{p_i}, modifiant alors la fonction globale.

Il est donc nécessaire de pouvoir définir quand et comment les agents doivent interagir de sorte à construire et maintenir une organisation produisant la fonctionnalité adéquate. C'est dans ce but qu'a été proposée l'approche par AMAS.

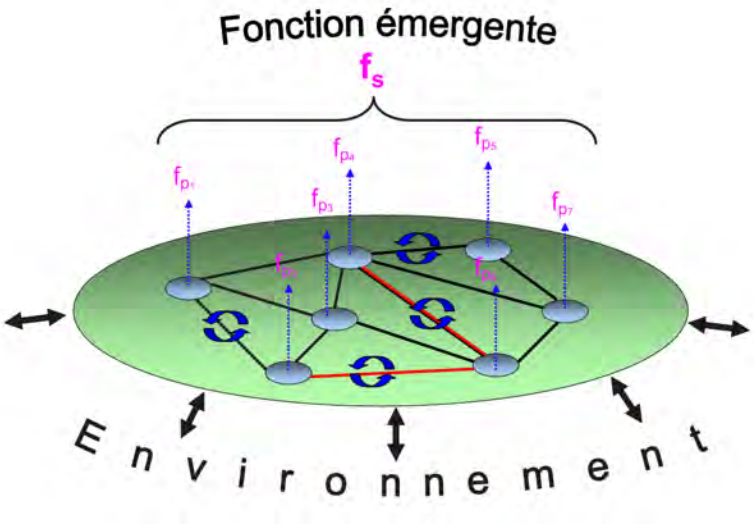

Figure 3.1 – Émergence d'une fonction globale dans un système multi-agent, extrait de [Picard, 2004]

3.1.2 L'approche par AMAS

Nous présentons maintenant l'approche par AMAS. Nous commençons par en faire une présentation générale, puis nous nous intéressons à la notion de coopération particulière à cette approche. Enfin, nous détaillons les règles générales guidant le comportement coopératif d'un agent au sein d'un AMAS.

3.1.2.1 Présentation générale

Face à un nouveau problème à résoudre, l'approche la plus classique consiste à définir clairement le problème et à le diviser en différents sous-problèmes, puis à réitérer ce processus jusqu'à l'obtention de sous-problèmes faciles à résoudre. Cette approche est qualifiée de *top-down*. Cependant, cette approche peut s'avérer inadaptée pour résoudre certains problèmes, notamment lorsque ces problèmes sont incomplètement spécifiés.

A l'opposé de l'approche *top-down* classique, l'approche par AMAS (dont une description complète est notamment fournie par [Georgé *et al.*, 2011]) est une approche *bottom-up*, où les différentes sous-parties du système sont définies dans le but d'obtenir un comportement global répondant à un problème donné. La fonctionnalité recherchée n'est pas spécifiée *a priori* ; il s'agit plutôt d'attribuer aux différents agents un comportement local assurant que la fonctionnalité globale obtenue soit satisfaisante pour répondre au problème. Dans ce cas là, le système est dit en adéquation fonctionnelle avec son environnement.

Pour obtenir ce résultat, l'approche par AMAS se base sur le théorème de l'adéquation fonctionnelle énoncé par [Camps, 1998]. Ce théorème stipule que, pour tout système fonctionnellement adéquat, il existe au moins un système à milieu intérieur coopératif qui réalise une fonction équivalente dans le même environnement. Un AMAS est donc composé d'agents possédant leur propre objectif, et interagissant entre eux de façon coopérative. Les interactions coopératives entre les différents agents mènent alors le système vers l'adéquation fonctionnelle.

3.1.2.2 Coopération

Une définition générale de la coopération est donnée par [Picard, 2004], qui la caractérise comme le juste milieu entre l'altruisme (propension à favoriser les objectifs des autres avant les siens) et l'égoïsme (propension à favoriser ses propres intérêts avant ceux des autres). Un agent coopératif cherche donc à atteindre aussi bien ses objectifs qu'à aider les autres agents à atteindre les leurs, de façon aussi égalitaire que possible.

Plus précisément, la théorie des AMAS établit qu'un agent est en situation coopérative dès lors qu'il respecte les trois méta-règles suivantes :

- tout signal perçu par l'agent est compris par lui sans ambiguïté ;
- toute information provenant de ses perceptions est utile à son raisonnement ;
- son raisonnement lui permet d'effectuer des actions utiles aux autres et à l'environnement.

Dans le cas où un agent se trouve face à une situation pour laquelle une de ces méta-règles n'est pas respectée, cette situation est appelée "Situation Non Coopérative", ou SNC. L'agent doit alors soit changer sa fonction partielle, soit changer de place dans l'organisation actuelle (voir section 3.1.2.4).

Sept types génériques de SNC que peut rencontrer un agent (soit vis-à-vis d'un autre agent, soit vis-à-vis de son environnement) ont été mises en évidence à partir de la définition de la coopération [Georgé *et al.*, 2011] :

1. *Incompréhension* : l'agent perçoit un signal qu'il ne comprend pas ;
2. *Ambiguïté* : l'agent attribue plusieurs interprétations possibles pour un signal perçu ;
3. *Incompétence* : l'agent n'est pas capable d'exploiter le signal perçu lors de son raisonnement ;
4. *Improductivité* : l'agent ne peut tirer aucune conclusion à partir du signal perçu ;
5. *Concurrence* : l'agent croit que son action et celle d'un autre agent vont aboutir au même résultat ;

6. *Conflit* : l'agent croit que la réalisation de son action est incompatible et avec celle d'un autre agent (elle empêche cette autre agent de réaliser sa propre action) ;
7. *Inutilité* : l'agent croit que son action ne va ni le rapprocher de son but, ni aider un autre agent.

[Bonjean *et al.*, 2009] définit le comportement d'un agent coopératif comme constitué de deux comportements distincts : un comportement nominal, décrivant sa fonctionnalité au sein du système et les actions qu'il réalise pour atteindre ses objectifs indépendamment des SNC qu'il est susceptible de rencontrer, et un comportement coopératif qui lui permet de prévenir, détecter et résoudre les SNC qu'il est susceptible de rencontrer. Le comportement coopératif des agents est à l'origine de l'auto-adaptation d'un AMAS : au fur et à mesure que les différentes SNC sont résolues ou évitées, le système tend vers l'adéquation fonctionnelle.

3.1.2.3 Notion de criticité

Pour établir un comportement coopératif, l'approche par AMAS intègre la notion de criticité. Selon [Lemouzy, 2011], la criticité correspond à "*la notion de distance entre la situation courante et le but local de l'agent*". Ainsi, "*plus l'agent est éloigné de son but, plus il estime sa situation courante comme critique*". Un agent est coopératif s'il agit de manière à aider l'agent de son voisinage qu'il considère être le plus critique. S'il est lui même l'agent le plus critique, il cherche alors à réduire sa propre criticité.

En tant qu'agents coopératifs, tous les agents au sein d'un AMAS tentent en continu de réduire la criticité de l'agent le plus critique (éventuellement lui-même), tout en évitant que le processus visant à aider cet agent n'amène un autre agent à devenir encore plus critique. Éventuellement, si un agent s'avère ne pas être en mesure d'aider l'agent le plus critique de son voisinage, son attitude coopérative l'amène à aider d'autres agents moins critiques, dans l'espoir que ces derniers soient en mesure d'aider l'agent le plus critique grâce à la réduction de leur propre criticité. Cette notion de criticité est notamment employée dans la résolution de certaines SNC, pour déterminer quelle SNC résoudre en premier par exemple.

3.1.2.4 Mécanismes coopératifs

Lorsqu'un agent coopératif se retrouve face à une SNC, son objectif est d'une part de réparer cette SNC (pour revenir à un état coopératif), et d'autre part d'éviter dans la mesure du possible de se retrouver à en générer de nouvelles (nous parlons alors de prévention de SNC). Pour cela, il réalise des actions dites "coopératives". [Capera, 2005] a décomposé ces actions coopératives d'agents AMAS en trois types de mécanismes :

1. Ajustement (ou *tuning*) : l'agent réalise des ajustements limités à son état interne afin de modifier sa fonctionnalité ;
2. Réorganisation : si les mécanismes d'ajustement ne sont pas suffisants, l'agent modifie ses liens d'interaction avec les agents de son voisinage ;
3. Évolution : si les mécanismes d'ajustement et de réorganisation ne sont pas suffisants, l'agent utilise un mécanisme d'évolution, autrement dit de création ou de suppression

d'agents. La création d'un agent se produit quand aucun des agents existants n'est capable de produire une fonctionnalité jugée nécessaire par un des agents existants, ou lorsqu'un agent est surchargé. La suppression d'un agent se produit lorsque la fonctionnalité produite par cet agent devient inutile pour son environnement.

La principale différence entre ces trois catégories de mécanismes réside dans l'impact que leur application entraîne au niveau du système multi-agent. En effet, les mécanismes de tuning n'ont qu'un impact léger sur le système car les modifications apportées se limitent à l'état interne de l'agent appliquant ce mécanisme. En revanche, les mécanismes d'auto-organisation, et à plus forte raison d'évolution, ont un impact bien plus important sur le système car ils modifient sa topologie. Afin de ne pas perturber inutilement les autres agents du système, un agent coopératif cherche toujours à minimiser l'impact que son comportement coopératif aura sur le système. C'est pourquoi, si plusieurs mécanismes permettent de résoudre une SNC, un agent coopératif choisit de préférence un mécanisme de tuning, puis d'auto-organisation, et enfin seulement d'évolution.

3.1.2.5 Méthode ADELFE

L'approche par AMAS fournit donc une approche générale pour la conception de systèmes auto-adaptatifs, qu'il convient d'instancier en fonction du problème à résoudre. Cette approche n'a encore jamais été appliquée à l'apprentissage du comportement de l'utilisateur en environnement ambiant. En revanche, elle a été appliquée à des domaines partageant certaines caractéristiques avec l'ambiant, notamment en terme de dynamisme (apparition/disparition d'entités) et de distribution. C'est par exemple le cas du contrôle de bioprocédés ([Videau *et al.*, 2010]), du contrôle manufacturier ([Kaddoum, 2011]) et du contrôle de moteurs thermiques ([Boes *et al.*, 2013]). Cependant, en dépit des performances obtenues par l'application des AMAS à différents problèmes, la conception d'un tel système présente de nombreuses difficultés.

Plusieurs méthodes ont été proposées pour aider à la conception d'un système multi-agent, telles que :

- Tropos de [Bresciani *et al.*, 2004], une méthode de développement orientée agent qui se base sur une approche orientée buts, et qui utilise la notation i* pour définir les agents sous la forme d'acteurs. Principalement, Tropos utilise deux types de diagrammes : les diagrammes *acteur* qui décrivent les dépendances entre acteurs, et les diagrammes *but* qui illustrent l'analyse des buts et des plans du point de vue acteur ;
- INGENIAS de [Pavón, 2006], une méthode dont l'objectif est d'aider au développement de systèmes multi-agents en divisant le problème en plusieurs aspects plus concrets, qui forment les différentes perspectives d'un système multi-agent (perspectives Agent, Organisation, Environnement, Buts/Tâches, et Interaction). Cette méthode est donc fondée sur la description de ces différentes perspectives (sous la forme de méta-modèles) afin de permettre aux concepteurs de définir itérativement l'architecture et les fonctionnalités de leurs systèmes multi-agents ;
- PASSI de [Cossentino *et al.*, 2008], une méthode de développement couvrant tout le cycle de développement d'un SMA depuis la spécification des besoins jusqu'à l'implémentation, en se basant sur 5 modèles : le modèle de collecte des besoins, le modèle

de société d'agent, le modèle d'implémentation d'agent, le modèle de code, et enfin le modèle de déploiement.

Plus spécifique au développement des AMAS, la méthode ADELFE (Atelier de DEveloppement de Logiciel à Fonctionnalité Emergente) a été proposée par [Picard, 2004], puis étendue par [Rougemaille, 2008] et [Bonjean *et al.*, 2013]. Elle s'appuie sur le processus RUP (*Rational Unified Process*) pour guider le concepteur durant les 5 phases de la conception d'un AMAS :

1. Les besoins préliminaires qui permettent d'obtenir une description précise et consensuelle du problème, sans nécessiter de langage de modélisation spécifique ;
2. Les besoins finals qui correspondent à l'étude de l'environnement du système à concevoir et des interactions avec les utilisateurs ;
3. L'analyse qui permet d'identifier les différentes entités du système à concevoir, d'étudier et évaluer la pertinence de l'approche par AMAS pour résoudre le problème, puis d'identifier les agents et leurs interactions ;
4. La conception qui permet de spécifier le langage d'interaction entre agents, ainsi que les comportements des différents agents ;
5. Le codage durant lequel le système est implémenté.

Chaque phase est divisée en plusieurs activités, elles-mêmes divisées en plusieurs étapes. Les activités et étapes en gras dans la figure 3.2 sont spécifiques à la conception d'un AMAS.

3.2 Spécification et conception d'*Amadeus*

Dans ce chapitre, nous détaillons les cinq phases la conception du système multi-agent *Amadeus*, en nous focalisant en particulier sur les étapes spécifiques à la conception d'un AMAS.

3.2.1 Besoins préliminaires et finals

L'objectif du système *Amadeus* est d'assurer la sensibilisation au contexte d'un environnement ambiant pour lui permettre d'adapter son comportement et ainsi améliorer le confort de ses utilisateurs. Au travers des différents capteurs de l'environnement, *Amadeus* doit acquérir assez d'informations pour anticiper les actions des utilisateurs sur les différents effecteurs afin de réaliser ces actions à leur place.

Nous définissons une liste des mot-clés utiles à la compréhension du problème et de son domaine. Les définitions associées à chaque mot-clé ne sont pas forcément consensuelles, mais nous nous baserons sur celles-ci pour le reste de notre étude.

– Environnement ambiant : espace physique composé d'un ensemble de dispositifs interconnectés. Un environnement ambiant se caractérise notamment par sa forte dynamique, certains dispositifs pouvant apparaître ou disparaître au cours du temps, tout comme les connections entre eux. Cette dynamique rend notamment difficile, voire impossible, de délimiter nettement les frontières d'un environnement ambiant ;

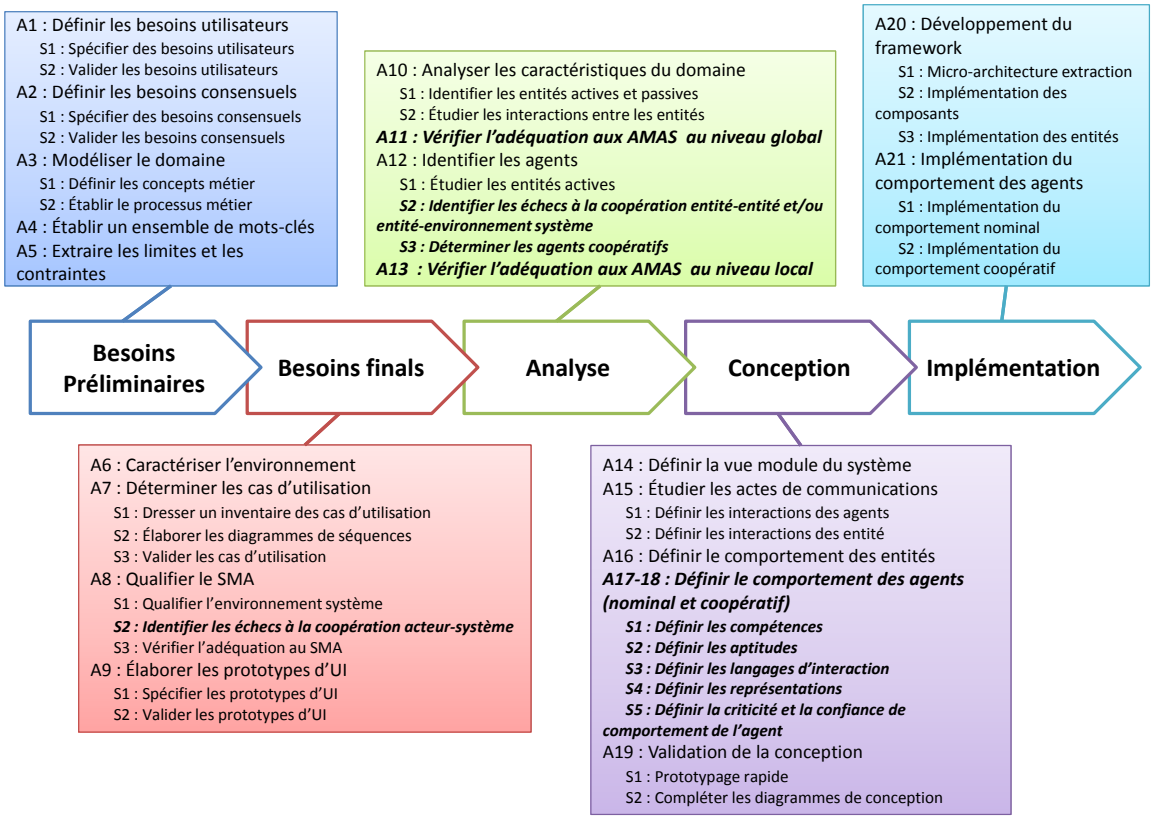

Figure 3.2 – Méthode ADELFE ([Bonjean *et al.*, 2013])

- Système ambiant : environnement ambiant doté d'une couche logicielle reliant les différents dispositifs ;
- Dispositif : entité matérielle présente dans l'environnement ambiant, et possédant au minimum un capteur et/ou un effecteur. De plus, il possède la capacité computationnelle de supporter une application logicielle, ou au moins d'être connecté à un autre dispositif qui la supportera pour lui ;
- Capteur : élément associé à un dispositif qui permet de percevoir des informations depuis l'environnement (exemple : capteur de luminosité, de présence, etc.) ;
- Effecteur : élément associé à un dispositif lui permettant d'agir dans l'environnement, et par conséquent d'en modifier l'état. Un effecteur intègre aussi un capteur, car son propre état fait partie des informations de l'environnement (exemple : lampe, store électrique, radiateur) ;
- Action : affectation de l'état d'un effecteur, comme par exemple allumer une lampe, ou fermer un store. Le maintien de l'effecteur dans son état courant peut aussi être vu comme une action. Une action peut être faite par un utilisateur ou par une application logicielle ;
- Variables contextuelles : ensemble des variables perceptibles par des capteurs, et pouvant avoir un impact sur l'exécution d'une tâche par une entité (utilisateur ou entité logicielle). Chaque variable génère des données au cours du temps. Une variable est

considérée comme utile vis-à-vis d'une action si la connaissance de cette variable a un impact sur les prises de décisions quant à la réalisation de cette action par un utilisateur ou par une application logicielle (par exemple, la luminosité d'une pièce est une des variables utiles pour décider quand agir sur la lampe de cette même pièce) ;
- Situation : état (valeur) de l'ensemble des variables contextuelles à un moment donné.

Les besoins fonctionnels désignent les contraintes liées au comportement du système :
- L'adaptation d'*Amadeus* doit être dynamique : il doit être capable de s'adapter aux situations non prévues, comme les comportements changeants des utilisateurs ;
- L'apprentissage d'*Amadeus* doit à la fois être rapide et produire des actions correctes et opportunes. Même s'il ne l'aide pas immédiatement, l'action d'un dispositif ne doit pas gêner l'utilisateur. L'apprentissage des actions doit être suffisamment rapide pour ne pas paraître inutile à l'utilisateur. Par exemple, si après plusieurs jours, le système est toujours en train d'apprendre et n'agit toujours pas, l'utilisateur risque de le considérer comme inutile ;
- *Amadeus* agit en temps contraint : le raisonnement du système doit être assez rapide pour lui permettre d'agir dans un temps acceptable par l'utilisateur. Par exemple, nous pouvons considérer que si le système met 5 minutes à décider d'allumer la lampe quand l'utilisateur entre dans la pièce, celui-ci le jugera inefficace ;
- La gêne occasionnée par les mauvaises actions du système doit être sensiblement inférieure au bénéfice procuré par ses actions correctes. Nous pouvons considérer qu'un utilisateur sera tolérant à quelques mauvaises actions du système s'il agit correctement le reste du temps.

Les besoins non fonctionnels désignent les contraintes liées à la mise en application du système :
- *Amadeus* doit être ouvert : de nouveaux capteurs ou effecteurs doivent pouvoir apparaître ou disparaître en cours de fonctionnement ;
- *Amadeus* doit pouvoir fonctionner sur des systèmes embarqués avec peu de mémoire et de puissance de calcul.
- *Amadeus* doit disposer des capteurs suffisant pour lui fournir l'ensemble minimal des données requises à son apprentissage.

La phase des besoins préliminaires doit aussi établir les limites du système à concevoir, autrement dit ce qui n'appartient pas à ses objectifs. Dans notre cas, l'objectif d'*Amadeus* se limite à anticiper les actions régulières de l'utilisateur ; il n'a pas pour objectif d'effectuer des actions sur des évènements particuliers et à risque, en cas d'accidents par exemple. Il n'a pas non plus pour objectif d'assurer un niveau de certitude dans ses actions suffisant pour un cadre d'utilisation impliquant la sécurité des utilisateurs.

L'environnement d'*Amadeus* se compose d'un ensemble d'entités actives et passives. Parmi celles-ci , nous distinguons :

- Entités actives : les *utilisateurs* qui peuvent agir sur les effecteurs, et qui ont un comportement autonome.
- Entités passives :les *dispositifs*, composés de *capteurs* qui perçoivent et transmettent les données de l'environnement sans agir dessus, et d'*effecteurs* qui changent de valeur uniquement sous l'action des utilisateurs ou du système.

En se basant sur les définitions de [Russell et Norvig, 1995], l'environnement d'*Amadeus* possède les propriétés suivantes :

Inaccessible : les caractéristiques d'un système ambiant font qu'il est impossible d'espérer percevoir la totalité du système à tout moment, les informations perçues pouvant facilement être soumises à des délais, des bruits, voire même disparaître ;

Continu : la forte dynamique d'un système ambiant fait que le nombre d'états dans lequel il peut être n'est pas limité ;

Non déterministe : par nature, les effets d'une action dans le monde physique réel ne peut être prédit de façon certaine (par exemple, l'acceptation d'une action par un utilisateur ne peut-être certaine à 100%) ;

Dynamique : un système ambiant est en constante évolution, changeant au cours du temps et sous les actions des utilisateurs.

Les entités actives et passives dans l'environnement interagissent avec le système. En particulier :

- Les capteurs envoient au système les données perçues ;
- Les utilisateurs peuvent modifier l'état des effecteurs ;
- Les effecteurs signalent au système tout changement d'état réalisé par un utilisateur ;
- Le système peut modifier l'état d'un effecteur.

3.2.2 Analyse

Cette phase d'analyse permet de vérifier l'adéquation des AMAS pour résoudre notre problème. Le cas échéant, elle propose d'identifier les agents de notre système. Ces agents pouvant être eux-mêmes des AMAS, il est nécessaire de vérifier aussi l'adéquation aux AMAS de ces agents. Ce processus est répété jusqu'à ce que la granularité des agents obtenus soit suffisamment faible pour ne pas nécessiter d'être conçu comme un AMAS.

Dans cette section, nous présentons une première itération de la phase d'analyse menant jusqu'à la définition des agents *dispositif*. Puis, nous présentons une seconde itération de cette phase d'analyse où l'agent *dispositif* est lui-même analysé comme un AMAS. Enfin, nous présentons l'architecture de l'AMAS d'AMAS *Amadeus* obtenu à la fin de la phase Analyse.

3.2.2.1 Première itération au niveau système

La méthode ADELFE nous fourni un outil basé sur un questionnaire permettant d'établir la pertinence de l'approche des AMAS pour notre problème. A partir de ces réponses, l'outil évalue la pertinence de l'application des AMAS à notre problème. Cet outil est représenté par la figure 3.3.

Cet outil confirme sans ambiguïté la pertinence de l'approche des AMAS pour la conception de notre système. En effet, la quasi-totalité des propriétés qui rendent l'application des AMAS pertinente sont présentes pour notre problème : l'absence d'algorithme répondant de

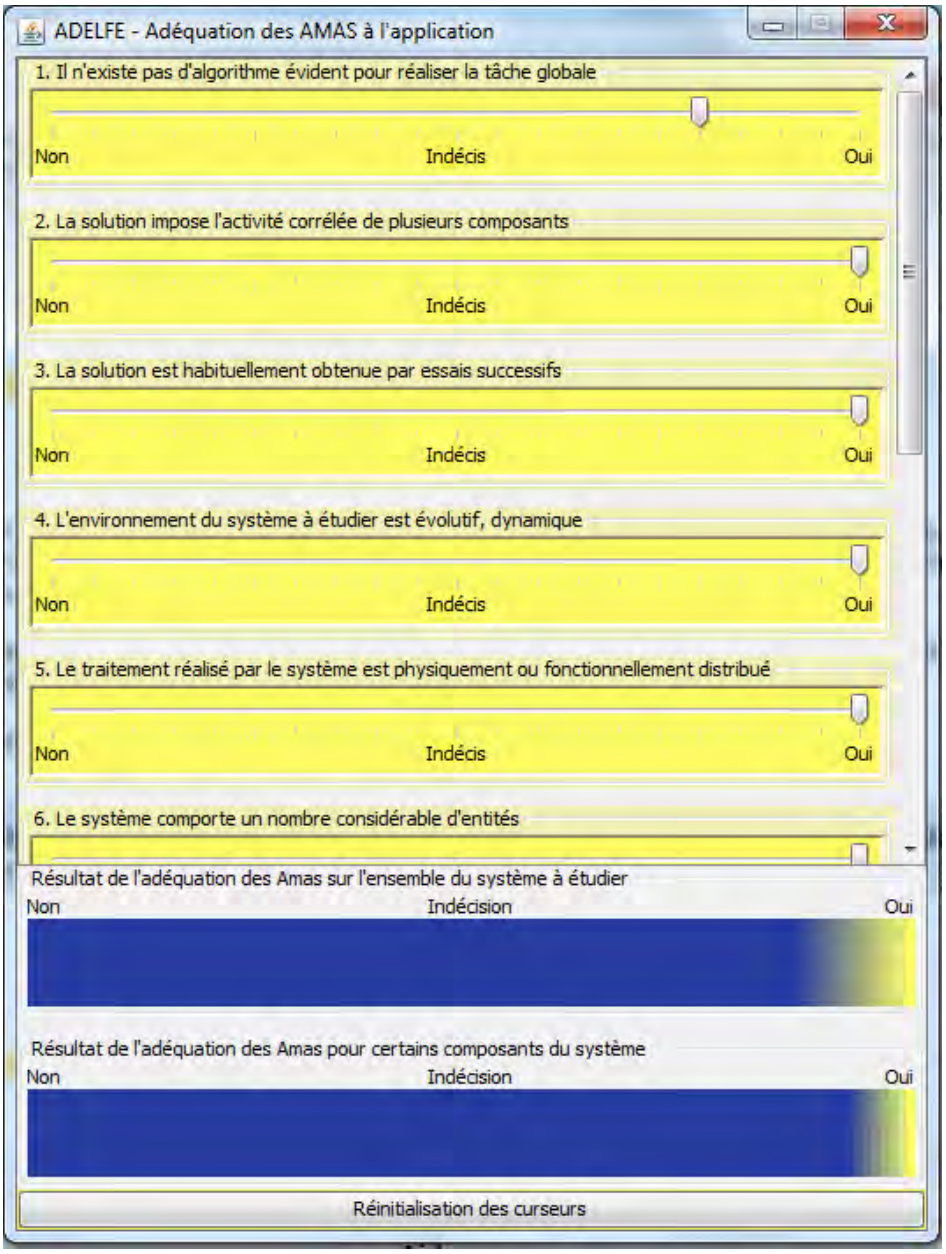

Figure 3.3 – Outil d'ADELFE pour l'évaluation de la pertinence de l'application des AMAS à notre problème

façon évidente au problème, la présence de grand nombre de composants corrélés, un environnement évolutif, dynamique, ouvert et partiellement perceptible, et la forte probabilité pour notre système de se retrouver dans des situations non prévues.

Les agents de notre système sont alors déterminés à partir des entités actives et passives identifiées lors de la caractérisation de l'environnement d'*Amadeus*, ainsi que des mots-clés employés pour caractériser cet environnement. Dans cette première itération, nous avons considéré qu'un environnement est composé d'utilisateurs réalisant des actions sur des dispositifs. Notre système ayant pour objectif d'apprendre quel comportement attribuer à ces dispositifs, nous avons jugé judicieux d'agentifier ces dispositifs.

Un agent *dispositif* est alors associé à chacun des dispositifs de l'environnement ambiant, chacun de ces agents ayant alors pour objectif d'assurer un contrôle correct de son dispositif, tout en coopérant avec les autres agents *dispositif* pour s'échanger les informations nécessaires à leur traitement respectif. Ce contrôle passe par l'observation et la mémorisation des actions des utilisateurs en fonction du contexte dans lequel il se trouve, la prise de décision de quelle action réaliser en fonction du contexte courant, et l'échange des données perçues localement avec les autres agents *dispositif*.

3.2.2.2 Seconde itération au niveau des parties du système

Le test de l'adéquation aux AMAS proposé par la méthode ADELFE nous a permis de constater la pertinence de concevoir les agents *dispositif* comme étant eux-mêmes des AMAS, du fait de certaines de leurs propriétés (similaires à celles d'*Amadeus*) : l'absence d'algorithme répondant de façon évidente au problème, un environnement évolutif, dynamique, ouvert et partiellement perceptible, et la forte probabilité pour notre système de se retrouver dans des situations non prévues.

Nous avons donc réitéré les traitements de la phase d'analyse de la méthode ADELFE afin de déterminer les agents composant un AMAS *dispositif*. Pour cela, nous nous basons sur la description de l'environnement d'un AMAS *dispositif*, c'est-à-dire l'environnement d'*Amadeus* décrit précédemment auquel se rajoute le reste du système (autrement dit, les autres AMAS *dispositif* ainsi que les relations entre ces différents AMAS).

Un AMAS *dispositif* est associé à un dispositif, lui-même étant composé de capteurs et d'effecteurs. Nous commençons alors par définir les agents *capteur* et les agents *effecteur*, un *capteur* étant associé à chaque capteur d'un dispositif, et un agent *effecteur* étant associé à chaque effecteur d'un dispositif. De plus, les différents AMAS *dispositif* ayant besoin des données en provenance des autres AMAS *dispositif*, nous définissons les agents *donnée*, un agent *donnée* étant associé à chaque variable perçue et représentant les données issues de cette variable. Nous avons donc dans chaque AMAS *dispositif* un agent *donnée* par capteur interne, et un agent *donnée* par capteur appartenant à un autre AMAS *dispositif*. Enfin, un AMAS *dispositif* ayant pour objectif d'assurer le contrôler de son dispositif en fonction de son contexte, nous définissons les agents *contrôleur* et les agents *contexte*, un agent *contrôleur* assurant le contrôle pour un effecteur associé, et un agent *contexte* représentant une possibilité d'action réalisable dans un contexte particulier.

Au final, nous avons réifié l'agent *dispositif* sous la forme d'un AMAS *dispositif* composé de cinq types d'agents :

1. Un agent *capteur* qui est en charge de la propagation pertinente des données en provenance de son capteur et à destination des différents AMAS *dispositif* ;
2. Un agent *effecteur* qui réalise le même traitement que l'agent *capteur* (propager les données représentant l'état de l'effecteur) mais est aussi capable de modifier l'état de l'effecteur qui lui est associé ;
3. Un agent *donnée* est en charge de la propagation d'une donnée auprès des agents *contexte* de son AMAS *dispositif*, ainsi que de l'évaluation de l'utilité de cette donnée pour ces agents *contexte* ;

4. Un agent *contrôleur* qui décide quelle action réaliser sur son effecteur en fonction du contexte courant ;
5. Un agent *contexte* qui propose une action particulière à réaliser dans un contexte spécifique.

Pour appliquer *Amadeus* dans un système ambiant, nous associons un AMAS *dispositif* à chaque dispositif, l'ensemble des AMAS *dispositif* répartis à travers l'environnement ambiant formant alors un AMAS de plus haut niveau. La figure 3.4 représente le système *Amadeus* appliqué à un système ambiant.

Au niveau global, *Amadeus* est donc constitué de plusieurs AMAS *dispositif* coopérant entre eux pour atteindre leur objectif (le contrôle de leur dispositif respectif). Si nous nous situons à un niveau plus local, *Amadeus* est constitué des cinq types d'agents énoncés précédemment. Les interactions entre les AMAS *dispositif* sont réalisées via les agents *capteur* et *effecteur* d'une part et les agents *donnée* d'autre part. La figure 3.5 montre un exemple d'AMAS *dispositif* appliqué à un dispositif comprenant deux effecteurs et un capteur, et percevant un autre capteur provenant d'un autre AMAS *dispositif*. Notons que chaque agent *contrôleur* est associé à un unique agent *effecteur* et à un ensemble d'agents *contexte*. Les agents *donnée* sont associés à au moins un groupe d'agents *contexte*, mais pas nécessairement à tous les agents *contexte* de l'AMAS *dispositif*.

Dans la suite du document, lorsque nous parlerons d'agents d'*Amadeus* en général, nous ferons référence aux agents *capteur*, *effecteur*, *donnée*, *contexte* et *contrôleur*. En revanche, lorsque nous nous intéresserons à l'ensemble des agents d'*Amadeus* associés à un dispositif particulier, nous utiliserons le terme d'AMAS *dispositif*. Concernant les agents d'*Amadeus*, l'application du test de l'adéquation aux AMAS nous a permis d'établir la non-pertinence de concevoir ces agents comme étant eux-mêmes des AMAS, leur granularité étant suffisamment faible pour être conçus directement comme des agents.

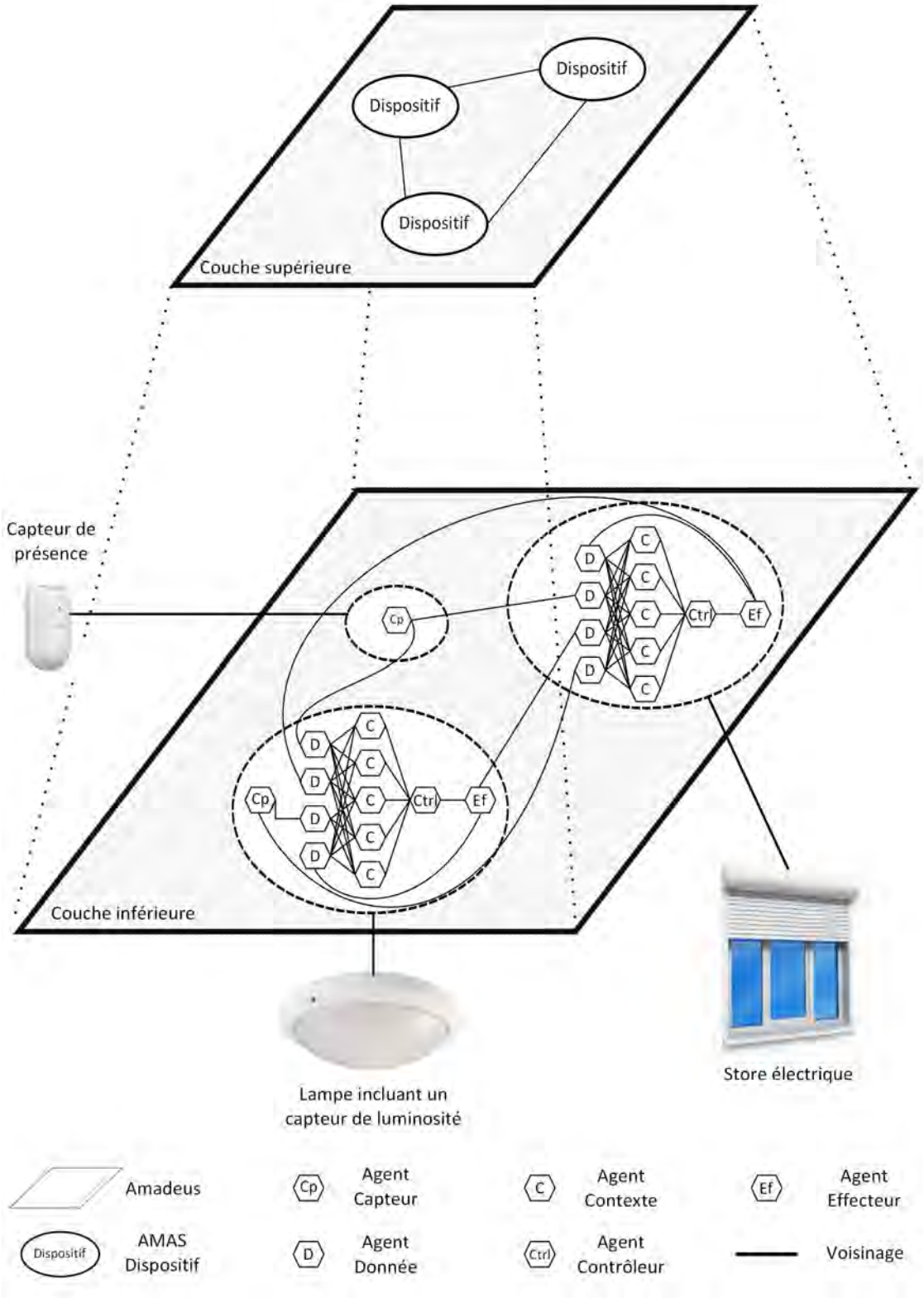

Figure 3.4 – *Amadeus* pour un système ambiant constitué d'une lampe (intégrant un capteur de luminosité), un capteur de présence et un store électrique. *Amadeus* est un AMAS (couche supérieure) lui-même composé d'AMAS (couche inférieure), formant alors un AMAS d'AMAS.

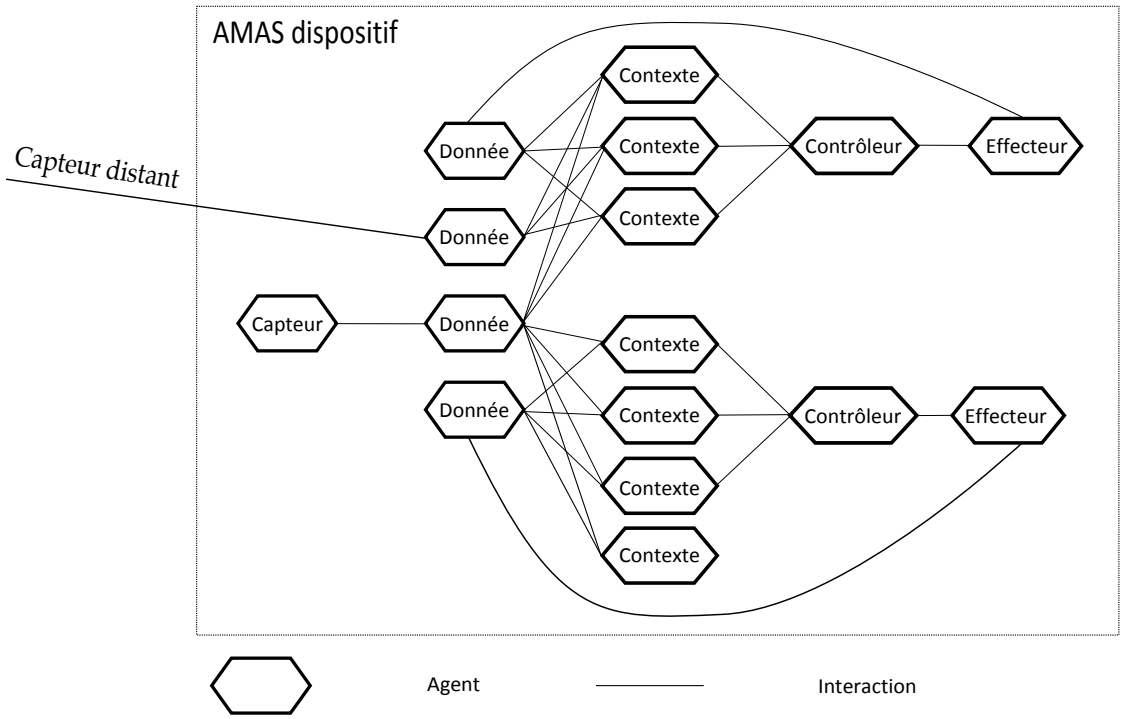

Figure 3.5 – AMAS *dispositif* associé à un dispositif composé d'un capteur et de deux effecteurs

3.2.3 Conception : comportement nominal des agents

Durant cette phase, nous détaillons les différents agents du système en suivant le modèle définit par [Bernon *et al.*, 2005]. Nous définissons leur objectif local, leur représentation du monde (composé du système et de son environnement), leurs aptitudes, leurs compétences et leurs langages d'interactions (l'aspect coopératif sera étudié dans la section suivante) :

- L'objectif d'un agent est le ou les états du monde qu'un agent s'engage à atteindre/maintenir ;
- La représentation du monde inclut ses connaissances sur son environnement et sur son état interne, ainsi que ses croyances vis-à-vis des autres agents (son environnement social) ;
- Les aptitudes d'un agent désignent sa capacité à raisonner à la fois sur ses connaissances et sur ses croyances. Il s'agit plus précisément de connaissances opératoires qui peuvent consister, par exemple, à interpréter un signal en provenance d'un autre agent ou de l'environnement ;
- Les compétences d'un agent désignent ses connaissances et son savoir faire sur le domaine. Elles sont spécifiques à son objectif ;
- Les langages d'interactions, qui se font ici par envoi de messages quelque soit le type d'agent défini dans *Amadeus*.

Pour faciliter la compréhension des objectifs et du fonctionnement des différents agents, nous présentons le fonctionnement global simplifié d'un AMAS *dispositif*, de la mise à jour d'une valeur perçue depuis un capteur, jusqu'à la décision de l'action à réaliser. Ce fonctionnement est illustré figure 3.6 à l'aide d'un exemple du contrôle d'un radiateur (par un AMAS *dispositif* associé à ce radiateur) ; nous décrivons ici les différentes étapes :

1. L'agent *capteur* perçoit un changement de valeur depuis le capteur physique (exemple : depuis un capteur de température) ;
2. Cet agent *capteur* envoie la nouvelle donnée à l'agent *donnée* qui lui est associé ;
3. L'agent *donnée* envoie sa nouvelle valeur à l'ensemble des agents *contexte* locaux (supposons qu'il y en ait quatre dans cet exemple) pour qui cette donnée est utile ;
4. (a) Un agent *contexte*, devenu valide grâce à la mise à jour, propose de réaliser une action (par exemple, il propose de mettre le radiateur à 1) ;

   (b) Un autre agent *contexte*, devenu lui aussi valide grâce à la mise à jour, propose de réaliser une autre action (par exemple, il propose de mettre le radiateur à 2) ;
5. (a) L'agent *contrôleur*, ayant choisi la première proposition d'action, sélectionne le premier agent *contexte*.

   (b) L'agent *contrôleur* déselectionne en même temps l'agent *contexte* sélectionné précédent (qui proposait par exemple de maintenir le radiateur à 0) ;
6. L'agent *contrôleur* envoie la proposition d'action de l'agent sélectionné à l'agent *effecteur* (associé au radiateur dans notre exemple) ;
7. L'agent *effecteur* réalise l'action demandée par l'agent *contrôleur* (dans l'exemple, mettre le radiateur à 1).

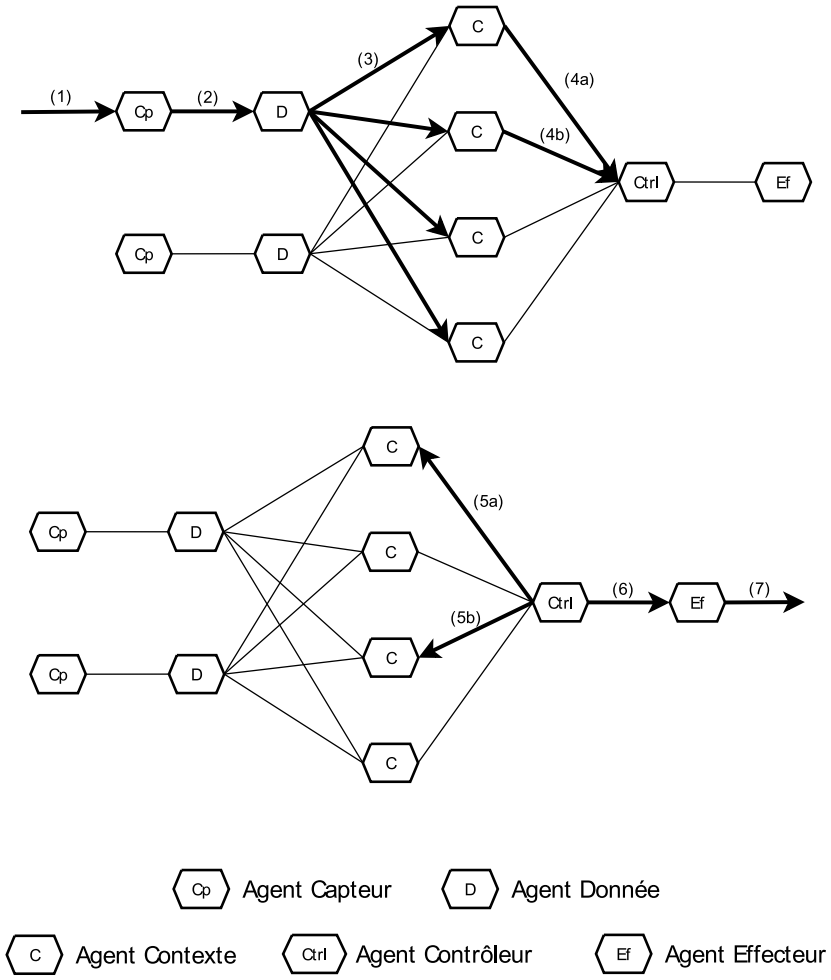

Figure 3.6 – Illustration du fonctionnement d'*Amadeus*

Avant de présenter en détails les différents agents d'*Amadeus*, il est nécessaire de revenir sur un point important concernant la méthode ADELFE. En effet, bien que les différentes étapes de cette méthode soient présentées de façon séquentielle, leur application réelle nécessite généralement de réitérer plusieurs fois différentes étapes jusqu'à obtenir un résultat satisfaisant. C'est pourquoi la présence de certains éléments dans la présentation des agents (au niveau de leurs compétences, aptitudes et représentation du monde notamment) ne sera expliqué que plus tard, lors de la présentation du comportement coopératif des agents. De plus, le comportement des différentes agents se composant de leur comportement nominal et de leur comportement coopératif, il est nécessaire de lire la description de ces deux comportements pour comprendre l'intégralité du fonctionnement des différents agents d'*Amadeus*.

3.2.3.1 Agent *capteur*

Objectif Au sein de chaque AMAS *dispositif* déployé sur les différents dispositifs du système ambiant, un agent *capteur* est associé à chacun des capteurs du dispositif associé. Cet agent est donc associé à la variable représentant l'état de ce capteur. Pour rappel, une va-

riable génère un ensemble de données au cours du temps. L'objectif d'un agent *capteur* est alors d'assurer la transmission de ces données à chaque agent *donnée* qui lui est associé, qu'il appartienne au même AMAS *dispositif* que l'agent *capteur* ou qu'il appartienne à un autre AMAS *dispositif* présent dans le système ambiant. En effet, les variables perceptibles localement dans un dispositif ne sont généralement pas suffisantes pour assurer un contrôle efficace de ce dispositif, il est donc nécessaire que les données perçues soient transmises entre les AMAS *dispositif*.

Représentation du monde Un agent *capteur* reçoit directement les valeurs de la variable perçue depuis le capteur associé. Il connaît donc en continu une donnée issue de l'environnement (par exemple, la luminosité ou la température d'une pièce, ou bien l'état ouvert/fermé d'une porte). L'agent *capteur* connaît aussi la valeur qu'avait la variable la dernière fois qu'il l'a transmise.

Aptitude Un agent *capteur* est capable aussi bien d'envoyer des messages à des agents appartenant au même AMAS *dispositif* que lui qu'à des agents appartenant à d'autres AMAS *dispositif* présents dans le système ambiant (dans la pratique, il passe par le gestionnaire de contexte). Il est aussi capable de comparer la valeur courante de sa variable avec la valeur qu'avait cette variable lorsqu'il a envoyé une mise à jour aux différents agents *donnée* pour la dernière fois.

Compétence Si la transmission des mises à jour entre un agent *capteur* et les agents *donnée* associés était réalisé à travers une topologie client/serveur (l'agent *capteur* ne transmet une mise à jour qu'à la demande d'un agent *donnée*), cela mettrait ces agents face à différentes SNC, présentées dans les sections 3.2.4.6 et 3.2.4.7. La résolution de ces SNC nous a mené à doter l'agent *capteur* de la capacité à déterminer s'il est pertinent d'envoyer une mise à jour de la variable à chaque agent *donnée* qui lui est associé (local ou distant).

Cette compétence est basée sur le principe de la communication spontanée présenté par [Camps, 1998] : lorsque l'agent *capteur* considère qu'il est pertinent d'envoyer une mise à jour aux agents *donnée* qui lui sont associés, autrement dit lorsqu'il détecte une différence significative entre la valeur courante de sa variable et la valeur de cette variable lors de la dernière mise à jour, il envoie spontanément une mise à jour à ces agents *donnée*.

Comportement nominal Le comportement nominal de l'agent *capteur* pour un cycle de fonctionnement, réitéré en continu pendant toute la durée de vie de l'agent, est illustré par l'algorithme 3.1. Il consiste à percevoir les valeurs envoyées par le capteur qui lui est associé, et à évaluer si une mise à jour est pertinente, autrement dit à vérifier que la valeur perçue a significativement changé. Si c'est le cas, il diffuse la mise à jour de sa variable.

3.2.3.2 Agent *effecteur*

Objectif Un agent *effecteur* est associé à chacun des effecteurs d'un dispositif. Son objectif est le même que celui d'un agent *capteur*, à savoir assurer la diffusion de la donnée repré-

Algorithme 3.1 : Comportement nominal de l'agent *capteur* au cours d'un cycle de fonctionnement

```
1: NouvelleValeurPerçue ← Mise_à_Jour()
2: Si Evaluation_pertinence_mise_à_jour(DernièreValeurEnvoyée,
   NouvelleValeurPerçue) = Vrai Alors
3:    Diffusion_vers_agents_donnée_associés(NouvelleValeurPerçue)
4:    DernièreValeurEnvoyée ← NouvelleValeurPerçue
5: Fin Si
```

sentant l'état de son effecteur aux agents *donnée* qui lui sont associés et appartenant aux différents AMAS *dispositif* présents dans le système ambiant (incluant l'agent *donnée* local). Cependant, il possède un second objectif, qui est de répondre aux sollicitations de l'agent *contrôleur* qui lui est associé lorsque celui-ci lui demande d'affecter un nouvel état à son effecteur.

Représentation du monde L'agent *effecteur* est directement relié à son effecteur dont il connaît l'état courant. Il mémorise aussi l'état de l'effecteur au moment où il a envoyé sa mise à jour pour la dernière fois.

Aptitude L'agent *effecteur* possède les mêmes aptitudes que l'agent *capteur*.

Compétence L'agent *effecteur* possède aussi les mêmes compétences que l'agent *capteur*. Il est aussi capable d'interagir avec l'effecteur pour modifier son état.

Comportement nominal Il n'appartient pas à l'agent *effecteur* de décider de modifier l'état de son effecteur, cette décision appartenant à l'agent *contrôleur* qui lui est associé. Comme un seul agent *contrôleur* est associé à l'agent *effecteur*, il aurait été possible de regrouper ces deux agents, afin que la décision de modifier l'état de l'effecteur soit aussi attribuée à l'agent *effecteur*, mais nous avons décidé de séparer le rôle de ces deux agents dans un souci de clarté. Le comportement nominal de l'agent *effecteur* est donc quasiment le même que celui d'un agent *capteur*, auquel se rajoute le fait de modifier l'état de son effecteur si l'agent *contrôleur* associé le lui demande. Il prévient aussi l'agent *contrôleur* qui lui est associé en cas de changement d'état de l'effecteur. Le comportement nominal de l'agent *effecteur* pour un cycle de fonctionnement, réitéré en continu pendant toute la durée de vie de l'agent, est illustré par l'algorithme 3.2.

3.2.3.3 Agent *donnée*

Objectif Un agent *donnée* est responsable d'une donnée au sein d'un AMAS *dispositif*. Cette donnée peut provenir d'un agent *capteur* (ou *effecteur*) local au dispositif ou d'un agent *capteur* (ou *effecteur*) appartenant à un autre AMAS *dispositif*. L'objectif d'un agent *donnée* est de gérer la diffusion locale de sa donnée au sein de l'AMAS *dispositif* auquel il appartient en assurant la transparence vis-à-vis de la provenance de cette donnée (les agents traitent les

Algorithme 3.2 : Comportement nominal de l'agent *effecteur* au cours d'un cycle de fonctionnement

```
1: Si Réception_demande_modification_envoyé_par_agent_contrôleur(Nouvel_état) Alors
2:    Modification_effecteur(Nouvel_état)
3: Fin Si
4: NouvelleValeurPerçue ← Mise_à_Jour()
5: Si Evaluation_pertinence_mise_à_jour(DernièreValeurEnvoyée,
   NouvelleValeurPerçue) = Vrai Alors
6:    Diffusion_vers_agents_donnée_associés(NouvelleValeurPerçue)
7:    Envoi_vers_agent_contrôleur_associés(NouvelleValeurPerçue)
8:    DernièreValeurEnvoyée ← NouvelleValeurPerçue
9: Fin Si
```

mises à jour des agents *donnée* indifféremment de la localisation du capteur ou de l'effecteur que chaque agent *donnée* représente). Cette diffusion est destinée aux agents *contexte* pour qui la variable d'où provient cette donnée est jugée utile, c'est-à-dire aux agents pour qui la connaissance de cette donnée a un impact sur leur fonctionnement.

L'utilité (ou l'inutilité) d'une donnée est relative aux différentes actions réalisables, ces actions étant proposées par différents agents *contexte*. Par conséquent, pour un agent *donnée*, l'évaluation de l'utilité de sa donnée est la même pour tous les agents *contexte* associés à un même agent *contrôleur* et proposant la même action.

Représentation du monde Un agent *donnée* connaît la valeur de la donnée qui lui est transmise depuis un agent *capteur* ou un agent *effecteur*. Il considère que, tant qu'il n'y a pas de mise à jour depuis cet agent, sa connaissance de la valeur de cette donnée est correcte. L'agent *donnée* possède aussi une croyance concernant les agents *contexte* locaux, qui indique si sa donnée leur est utile ou pas.

Aptitude Chaque agent *donnée* possède l'aptitude de diffuser un message à l'ensemble des agents *contexte* locaux en fonction de l'effecteur qui leur est associé et de l'action proposée par chaque agent *contexte* (voir section 3.2.3.4).

Compétence Chaque agent *donnée* possède la compétence de déterminer si sa variable est utile aux différents agents *contexte* appartenant au même AMAS *dispositif* que lui. Cette capacité découle de la résolution de la SNC présentée section 3.2.4.8.

Comportement nominal Une variable est considérée comme utile vis-à-vis d'une action spécifique sur un effecteur donné si les décisions des utilisateurs de réaliser cette action sont dépendantes de la valeur des données issues de cette variable. Un agent *donnée* reçoit donc une donnée perçue depuis un agent *capteur* ou *effecteur* local ou distant, et son objectif est de fournir cette donnée aux agents *contexte* pour qui elle est utile. Un agent *donnée* possède alors une liste de paires <agents *contrôleur*/action> pour lequel il se considère utile (voir

section 3.2.4.8 pour la gestion de cette liste). Lorsqu'il reçoit une mise à jour de sa donnée, il la transmet donc à tous les agents *contexte* locaux proposant une des actions appartenant à cette liste. Le comportement nominal de l'agent *donnée* pour un cycle de fonctionnement, réitéré en continu pendant toute la durée de vie de l'agent, est illustré par l'algorithme 3.3.

Algorithme 3.3 : Comportement nominal de l'agent *donnée* au cours d'un cycle de fonctionnement

1: **Si** Réception_mise_à_jour **Alors**
2: ValeurCourante $\leftarrow$ Mise_à_Jour()
3: **Pour tout** $c \in$ ListeContexteLocaux tel que
$< c.AgentContrleurAssoci, c.ActionPropose > \in$ ListeUutilité **Faire**
4: envoie_mise_à_jour(ValeurCourante, c)
5: **Fin Pour**
6: **Fin Si**

3.2.3.4 Agent *contexte*

Objectif Lorsqu'un utilisateur réalise une action dans une certaine situation, nous pouvons considérer que cette action est pertinente pour cette situation spécifique. Par conséquent, une des hypothèses de notre étude est que, si cette même situation se présente à nouveau, alors la même action sera à nouveau pertinente. Cette connaissance est représentée au travers des agents *contexte*.

Un agent *contexte* associe une situation spécifique, dite "situation de validité", à une action à réaliser sur une effecteur donné. Son objectif est alors de déterminer si la situation courante telle qu'il la perçoit est similaire à sa situation de validité, auquel cas il considère comme pertinent que son action soit réalisée. Ne pouvant pas réaliser lui-même cette action, il la propose à l'agent *contrôleur* en charge du contrôle de l'effecteur sur lequel s'effectue son action.

Représentation du monde Une situation de référence représente la situation dans laquelle un utilisateur a réalisé une action. Chaque agent *contexte* prend en charge une connaissance portant sur la réalisation de cette action dans cette situation de référence. La proposition d'un agent *contexte* envoyée à un agent *contrôleur* peut alors être traduite comme ceci : "la situation courante est similaire à la situation de référence durant laquelle je sais qu'un utilisateur a réalisé telle action, aussi je pense que si tu réalises la même action maintenant, cela va le satisfaire". L'action elle-même est une valeur numérique représentant l'état à affecter à l'effecteur. Une situation pourrait être représentée comme l'ensemble des états des variables qui la caractérisent, mais se pose alors le problème d'évaluer la similarité entre deux situations. En effet, l'égalité parfaite des valeurs de chaque variable serait inapplicable avec des capteurs réels. Considérons un thermomètre retournant une température avec une précision de trois décimales après la virgule. Il est évident que si les valeurs caractérisant la situation courante sont identiques aux valeurs caractérisant la situation de référence à l'exception de la température, si la différence entre ces deux valeurs de température se joue à

le troisième décimale après la virgule, les deux situations peuvent être considérées comme étant similaires.

Au lieu d'une représentation basée sur des valeurs exactes, nous proposons d'associer à chaque variable perçue une plage de valeurs représentant l'ensemble des valeurs pour lesquelles l'agent *contexte* considère que la valeur courante est similaire à celle de la situation de référence. Ces plages de valeurs sont appelées *plages de validité*, chacune d'elles se considérant alors comme valide si la valeur courante de la donnée associée se situe entre les bornes de sa plage de valeurs. Chaque plage de validité est initialisée à $[v_d - \tau, v_d + \tau]$, v_d représentant la valeur de la donnée d dans la situation de référence, et τ représentant 2% de l'intervalle $[d_{MIN}, d_{MAX}]$ que peut prendre la donnée d (autrement dit, $\tau = 0.02 \cdot (d_{MIN} - d_{MAX})$). Cette valeur de τ a été obtenue au travers de divers tests empiriques.

Chaque plage de validité est modélisée sous la forme d'un AVRT, un outil que nous avons développé et présenté au chapitre 4.1. Il s'agit d'un outil logiciel permettant de représenter et manipuler une plage de valeurs dont les bornes évoluent en fonction des *feedbacks* perçues depuis son environnement. L'intégration (ou l'exclusion) d'une valeur au sein de la plage de valeurs se fait progressivement en fonction de la distance entre cette valeur et la borne la plus proche ; ainsi, l'intégration d'une valeur proche d'une des bornes nécessitera moins de *feedbacks* que l'intégration d'une valeur très éloignée des bornes de l'AVRT.

Voici donc la représentation du monde d'un agent *contexte* :

- une représentation de sa situation de validité sous la forme d'une liste de plages de validité (une par donnée perçue). L'agent *contexte* est lui-même valide si la totalité de ses plages de validité sont valides, et invalide dans le cas contraire ;
- une connaissance sur l'action à réaliser sur l'effecteur, autrement dit l'état à lui attribuer, lorsqu'il est valide. Cet état peut être différent de l'état courant de l'effecteur (changement d'état), ou être identique (maintient de l'état courant) ;
- un gain estimé, qui est une valeur numérique représentant une croyance sur l'intérêt de réaliser son action plutôt que celle d'un autre agent. Cet estimateur est implémenté grâce aux mécanismes de l'agent *self-ordered*, dont hérite l'agent *contexte* (voir section 4.2). Ces mécanismes fonctionnent grâce à une liste de croyances sur des agents *contexte* qu'un agent *contexte* considère supérieurs (agents supérieurs), et une autre liste sur les agents *contexte* qu'il considère inférieurs (agents inférieurs). Ces mécanismes permettent alors de générer une valeur numérique représentative de la hiérarchie entre les agents *contexte* : si un agent *contexte* c_1 est supérieur à un autre agent *contexte* c_2, alors le gain estimé de c_1 est supérieur à celui de c_2. Les mécanismes issus de l'agent *self-ordered* assurent aussi la consistance entre les gains estimés des différents agents *contexte* reliés par une relation d'ordre (directement ou à par transitivité avec d'autres agents *contexte*) ;
- un niveau de confiance qui représente la confiance qu'il accorde à sa proposition d'action lorsqu'il l'envoie à l'agent *contrôleur* ;
- une connaissance sur le fait d'être actuellement sélectionné par l'agent *contrôleur* ou pas.

Aptitude Un agent *contexte* est capable d'envoyer des messages aux agents de son voisinage, en l'occurrence son agent *contrôleur* et les agents *donnée* locaux. Il est aussi capable d'ajouter des agents *contexte* dans sa liste d'agents supérieurs ou inférieurs, ce qui modifie automatiquement la valeur de son gain estimé. L'utilisation d'AVRT pour modéliser ses plages de validité lui donne l'aptitude de détecter s'il est valide dans la situation courante, ainsi que l'aptitude de modifier les situations où il est valide, en intégrant de nouvelles situations, ou en excluant des situations précédemment valides. Enfin, il est capable d'augmenter ou diminuer la valeur de sa confiance.

Compétence Un agent *contexte* possède la compétence de déterminer la pertinence d'envoyer sa proposition d'action à son agent *contrôleur*, en fonction de son état de validité. Il est aussi capable d'estimer l'utilité de certaines variables perçues dans certaines situations (voir section 3.2.4.8).

Comportement nominal Le comportement nominal de l'agent *contexte* pour un cycle de fonctionnement, réitéré en continu pendant toute la durée de vie de l'agent, est illustré par l'algorithme 3.4. Celui-ci commence son cycle en mettant à jour sa situation de validité en fonction des mises à jours reçues depuis les agents *donnée*. Si l'ensemble des données sont situées dans l'intervalle de leur plage de validité respective, alors l'agent *contexte* devient valide à son tour. Dans ce cas, il envoie sa proposition d'action à l'agent *contrôleur*. Si sa proposition d'action est acceptée, l'agent *contexte* est sélectionné. Cela signifie que l'action qu'il a proposée va être appliquée à l'effecteur associé. Dans ce cas, il attendra jusqu'à ne plus être sélectionné, soit par déselection de l'agent *contrôleur*, soit par son invalidité (au moins une donnée est hors de sa plage de validité, ce qui amène l'agent *contexte* à annuler sa proposition d'action).

3.2.3.5 Agent *contrôleur*

Objectif L'objectif de l'agent *contrôleur* est de décider quelle action réaliser sur l'effecteur qui lui est associé de façon à ce que l'état de cet effecteur satisfasse les utilisateurs.

Représentation du monde Un agent *contrôleur* connaît un ensemble d'agents *contexte* qui lui sont associés et qui, en fonction de la situation courante, lui proposent différentes actions possibles. Il connait aussi un unique agent *effecteur*, à qui il transmet ses décisions d'actions.

Aptitude Un agent *contrôleur* est capable d'envoyer des messages aux agents *contexte* qui lui sont associés (afin de les (dé)sélectionner, notamment), ainsi qu'à son agent *effecteur* (afin de lui envoyer des demandes d'actions).

Compétence Un agent *contrôleur* possède la compétence de déterminer quelle est la meilleure action à réaliser, parmi un ensemble de propositions d'actions. Cette décision se base notamment sur le gain estimé et la confiance qui accompagnent chaque proposition d'action. Plus de détails sont présentés aux paragraphes 3.2.4.3 et 3.2.4.5.

Algorithme 3.4 : Comportement nominal de l'agent *contexte* au cours d'un cycle de fonctionnement

```
AncienEtatValiditéContexte ← EtatValiditéContexte
PlagesDeValidité ← Mise_à_Jour(PlagesDeValidité,EtatCourantDonnées)
Si (∀d ∈ PlagesDeValidité, d.état = valide) Alors
    EtatValiditéContexte ← valide
Sinon
    EtatValiditéContexte ← invalide
Fin Si
Si (AncienEtatValiditéContexte = invalide ET EtatValiditéContexte = valide Alors
    envoie_proposition_au_contrôleur(PropositionAction)
Fin Si
Si (AncienEtatValiditéContexte = valide ET EtatValiditéContexte = invalide Alors
    annule_proposition_au_contrôleur(PropositionAction)
    EtatSélectionContexte ← désélectionné
Fin Si
Si Sélection par le contrôleur Alors
    EtatSélectionContexte ← sélectionné
Fin Si
Si Désélection par le contrôleur Alors
    EtatSélectionContexte ← désélectionné
Fin Si
```

Comportement nominal L'agent *contrôleur* commence son cycle de vie en percevant les propositions d'actions envoyées par les agents *contexte*. Il intègre alors les nouvelles propositions d'actions à sa liste de propositions d'action, et supprime celles des agents *contexte* devenus invalides. Ensuite, l'agent *contrôleur* décide quelle proposition est la plus intéressante. Si l'agent *contexte* qui a fait cette proposition est différent de l'agent précédemment sélectionné, alors cet agent *contexte* précédemment sélectionné est déselectionné, tandis que l'agent *contexte* ayant fait la meilleure proposition d'action est sélectionné. Sa proposition d'action est alors envoyé à l'agent *effecteur* pour qu'il réalise cette action.

Le comportement nominal de l'agent *contrôleur* pour un cycle de fonctionnement, réitéré en continu pendant toute la durée de vie de l'agent, est illustré par l'algorithme 3.5.

3.2.4 Conception : comportement coopératif des agents

Nous avons distingué deux parties dans le comportement d'un agent (section 3.1.2.2) :

- son comportement **nominal** qui l'amène à atteindre son objectif tout en aidant les autres agents à atteindre le leur ;
- son comportement **coopératif** qui consiste à détecter ou prévenir les SNC et à ajuster son comportement (ou inciter les autres agents à le faire) pour résoudre les SNC rencontrées.

Algorithme 3.5 : Comportement nominal de l'agent *contrôleur* au cours d'un cycle de fonctionnement

1: Mise à jour ListeContextePropositions
2: *MeilleureProposition* ← NULL
3: **Pour tout** $p \in$ ListeContextePropositions **Faire**
4: **Si** p est une meilleure proposition d'action que *MeilleureProposition* **Alors**
5: *MeilleureProposition* ← p
6: **Fin Si**
7: **Fin Pour**
8: **Si** *MeilleureProposition*.Contexte $\neq$ *ContexteSélectionné* **Alors**
9: Envoie *Déselection* à *ContexteSélectionné*
10: *ContexteSélectionné* ← *MeilleureProposition*.Contexte
11: Envoie *Sélection* à *ContexteSélectionné*
12: Envoie *MeilleureProposition* à *EffecteurAssocié*
13: **Fin Si**

Selon l'approche par AMAS ([Georgé *et al.*, 2011]), le comportement global du système converge vers un comportement fonctionnellement adéquat au fur et à mesure que les SNC diminuent ou sont correctement anticipées.

La conception d'agents coopératifs passe donc par l'analyse des SNC qu'ils sont susceptibles de rencontrer. Pour chacune d'elles, le concepteur de l'AMAS commence par établir de quel type de SNC il s'agit. Le concepteur détermine qui est impliqué dans chacune de ces SNC, sachant qu'une SNC peut concerner un agent seul, ou faire intervenir plusieurs agents de même type ou de types différents, ou même concerner un agent avec l'environnement du système. Enfin, le concepteur détermine, pour chacune de ces SNC, les agents capables de la détecter et ceux qui sont capables de la résoudre. En effet, il est parfois possible que des agents se retrouvent impliqués dans une SNC, mais que seuls d'autres agents soient capables de la détecter et/ou de la résoudre. La résolution de cette SNC peut être réalisée, soit par un **ajustement** d'un paramètre interne à un agent, soit par une **réorganisation** entre les agents, soit par une **évolution** du système multi-agent (création ou suppression d'agents).

Nous établissons donc ici la liste des SNC que peuvent rencontrer les agents au sein d'*Amadeus*. Les comportements coopératifs découlant de la résolution de ces SNC viennent s'ajouter aux comportements nominaux définis dans la section précédente, et forment ainsi les comportements finaux des agents.

3.2.4.1 Réalisation d'une action par un utilisateur

Problématique L'objectif de l'agent *contrôleur* est de réaliser des actions à la place des utilisateurs. Si un utilisateur réalise une action alors que cette action ne faisait pas partie des propositions d'actions perçues par l'agent *contrôleur*, l'agent *contrôleur* est en SNC d'incompétence : il n'est pas capable de réaliser les actions à la place des utilisateurs au vu des propositions d'actions qu'il perçoit.

Détection Cette SNC est détectée par l'agent *contrôleur*, au moment où il perçoit une action de l'utilisateur qui ne fait pas partie des propositions d'actions envoyées par les agents *contexte* valides.

Résolution L'agent *contrôleur* qui a détecté cette SNC va créer un agent *contexte* en lui fournissant la description de la situation dans laquelle l'utilisateur a réalisé son action (autrement dit, la liste de l'état de chaque donnée perçu par l'AMAS *dispositif*), ainsi que la description de l'action elle-même. Nous nous intéressons plus spécifiquement au mécanisme de création d'un agent *contexte* qui survient lors de la résolution de cette SNC.

Pour rappel, un agent *contexte* se compose de :
- une plage de valeurs pour chaque variable perçue représentant la situation dans laquelle il est valide (autrement dit la situation dans laquelle il doit envoyer sa proposition d'action) ;
- une représentation de ladite proposition d'action ;
- un niveau de confiance ;
- un gain estimé, hérité de l'agent *self-ordered* (voir section 4.2) ;
- un statut de sélection.

Ces différents éléments sont initialisés comme suit :
- Les plages de valeurs, modélisées sous la forme d'*Adaptive Value Range Tracker* (section 4.1), sont initialisées à partir des valeurs des différentes variables au moment de la réalisation de l'action par l'utilisateur ;
- L'action est représentée par une valeur numérique représentant l'état à attribuer à l'effecteur ;
- Le niveau de confiance est initialisé à 0.5, soit un niveau de confiance moyen ;
- Le gain estimé est initialisé d'après le processus décrit en section 4.2 ;
- Le statut de sélection est initialisé à "non sélectionné".

Nous pouvons donc nous attendre à ce qu'au fur et à mesure que les utilisateurs réalisent des actions, le nombre d'agents *contexte* augmente, et enrichisse ainsi les connaissances d'*Amadeus* vis-à-vis du comportement des utilisateurs.

3.2.4.2 Aucun agent *contexte* valide

Problématique Lorsque l'intégralité des agents *contexte* qui sont associés à l'agent *contrôleur* sont invalides, l'agent *contrôleur* ne dispose d'aucune proposition d'action. Il se retrouve alors face à une SNC d'**improductivité**.

Détection Cette SNC est détectée par l'agent *contrôleur* lorsque la liste de propositions d'actions envoyées par les agents *contexte* est vide.

Résolution Pour résoudre cette SNC, deux solutions sont possibles : la première consiste en un ajustement au niveau des agents *contexte* tandis que la seconde entraîne une évolution du système multi-agent au travers de la création d'un nouvel agent *contexte*. Comme précisé dans la section 3.1.2.4, le comportement coopératif d'ajustement étant prioritaire sur celui

d'évolution, les agents impliqués dans la résolution de cette SNC choisiront la première solution dans la mesure du possible.

Première solution Elle consiste à étendre légèrement les plages de validité d'un agent *contexte*, en considérant que si la situation courante est proche de celle dans laquelle il est valide, alors sa proposition d'action lorsqu'il est valide à des chances d'être appropriée pour la situation courante.

Une plage de validité est dite "validable" si la valeur courante est extérieure aux bornes de la plage de validité, mais qu'un unique ajustement d'une de ses deux bornes lui permettrait de devenir valide. Cette information est obtenue grâce à l'AVRT utilisé pour modéliser chaque plages de validité, qui sait déterminer si, pour une valeur donnée extérieure à sa plage de valeurs, il lui est possible d'étendre ses bornes jusqu'à l'intégrer à la plage de validité par un unique ajustement d'une de ses bornes. Un agent *contexte* se considère lui-même "validable" si tous ses AVRT (un AVRT par donnée perçue) sont, soit valides, soit validables. L'algorithme 3.6 représente le traitement réalisé par un agent *contexte* pour déterminer son état de validité.

Algorithme 3.6 : Statut d'un agent *contexte*

1: **Si** $\forall AVRT \in List\_AVRT$, $AVRT$ est valide **Alors**
2: L'agent *contexte* est valide
3: **Sinon**
4: **Si** $\forall AVRT \in List\_AVRT$, $AVRT$ est valide ou validable **Alors**
5: L'agent *contexte* est validable
6: **Sinon**
7: L'agent *contexte* est invalide
8: **Fin Si**
9: **Fin Si**

Lorsqu'un agent *contexte* devient validable, il envoie sa proposition d'action à l'agent *contrôleur* de la même façon que s'il avait été valide, mais en précisant son état de validité. Ainsi, si l'ensemble des propositions d'actions envoyées par des agents *contexte* valides est vide, l'agent *contrôleur* peut dans ce cas sélectionner la proposition d'action d'un agent validable. L'agent *contexte* qui a proposé cette action utilise alors le mécanisme d'intégration d'une valeur sur ses plages de validité validables pour modifier ses plages de valeurs et devenir valide.

Par exemple, si un agent *contexte C* est invalide à cause de sa plage de validité liée à la luminosité, celle-ci couvrant les valeurs de 62 à 79 lux alors que la luminosité actuelle est de 80 lux, l'agent *C* peut se rendre compte qu'il est capable d'intégrer cette valeur en étendant légèrement sa plage de validité. Il est donc validable, et propose alors son action à son agent *contrôleur*, en précisant bien qu'il est validable et non valide. Si l'agent *contrôleur* ne dispose d'aucune proposition d'action envoyée par un agent *contexte* valide, il étudie les propositions d'action des agents *contexte* validables. Si c'est la proposition d'action de l'agent *contexte C* qui est choisie, l'agent *C* est sélectionné. Dans ce cas là, il modifie sa plage de validité pour devenir valide dans cette situation.

Seconde solution Lorsque les listes des propositions d'actions envoyées par les agents *contexte* valides et validables sont vides, la solution précédente ne peut pas être appliquée. Dans ce cas, une autre solution consiste à imiter le comportement des utilisateurs . Puisque les utilisateurs ne font rien dans la situation courante, l'agent *contrôleur* crée un nouvel agent *contexte* à partir de la situation courante, et lui attribue comme action à proposer celle actuellement réalisée par les utilisateurs, à savoir le maintien de l'effecteur dans l'état courant. Cet agent sera alors susceptible de faire évoluer ses plages de validité par la suite, au travers de l'application de la première solution.

3.2.4.3 Deux agents *contexte* proposent deux actions différentes

Problématique La résolution des SNC précédentes mène l'agent *contrôleur* à créer des agents *contexte* à partir des actions des utilisateurs. Ces agents *contexte* proposent alors leurs actions chaque fois que la situation courante est similaire à celle ayant menée à leur création. Il est cependant possible que deux agents *contexte* C_1 et C_2 se retrouvent simultanément valides alors qu'ils proposent deux actions A_1 et A_2 différentes. Ces deux agents *contexte* sont alors face à une situation de **conflit** : tous les deux pensent contribuer à la même situation (satisfaire les utilisateurs) au travers d'actions différentes.

Détection Cette SNC est détectée par l'agent *contrôleur* associé aux agents *contexte* en conflit, lorsqu'il reçoit leurs propositions.

Résolution Cette SNC mène l'agent *contrôleur* à devoir choisir entre (au moins) deux propositions d'actions A_1 et A_2. Pour faire ce choix, il lui faut disposer d'une valeur accompagnant les propositions d'actions, qui lui permettrait de les comparer entre elles et de choisir la meilleure.

Supposons que nous disposons d'un oracle qui, considérant une situation donnée, est capable de dire si les utilisateurs sont satisfaits ou pas, et d'attribuer une note explicite représentative de cette satisfaction. La présence de cet oracle faciliterait alors la capacité des agents *contexte* à attribuer une valeur de gain estimé pour évaluer la pertinence de leurs actions. Ainsi, observer la différence entre la satisfaction des utilisateurs avant et après la réalisation de l'action ayant mené à la création d'un agent *contexte* permettrait à cet agent d'initialiser le gain estimé, qu'il ajusterait alors en observant les effets successifs de la réalisation de cette action au fur et à mesure de ses propositions. L'agent *contrôleur* n'aurait alors plus qu'à choisir, à tout moment, la proposition associée au gain estimé le plus élevé pour résoudre cette SNC.

Cependant, pour une mise en pratique dans un environnement réel, il faudrait demander régulièrement et explicitement aux utilisateurs leur niveau de satisfaction, ce qui est une solution inappropriée. Cependant, l'agent *contrôleur* n'a pas besoin de connaitre explicitement les valeurs des gains estimés, mais uniquement de savoir laquelle de ces valeurs est la plus élevée. Nous avons donc mis en place une évaluation du gain estimé relative entre agents *contexte*, en concevant un mécanisme appelé "mécanisme d'auto-ordonnancement entre agents", détaillé en section 4.2. Plus précisément, ce mécanisme repose sur la défini-

tion d'un type d'agent générique, appelé agent *self-ordered*. Chaque agent *self-ordered* possède une liste d'agents supérieurs et une liste d'agents inférieurs. Son objectif est de fournir une valeur numérique, appelée son "niveau", telle que la valeur de son niveau soit strictement inférieure à celles des niveaux de ses agents supérieurs, et strictement supérieure à celles des niveaux de ses agents inférieurs. De plus, nous avons rendu l'agent *self-ordered* capable de mettre à jour son niveau en cas de modification (ajout ou suppression d'un agent) dans une de ses deux listes, de façon à toujours respecter la relation d'inégalité entre son niveau et ceux des agents supérieurs et inférieurs, tout en supprimant l'éventuelle apparition d'incohérence dans les relations d'ordre entre les agents *self-ordered* (la description des incohérences susceptibles d'apparaître, ainsi que le mécanisme pour les résoudre, étant aussi décrits en section 4.2).

Nous avons alors défini l'agent *contexte* comme une spécialisation de l'agent *self-ordered*. Dès lors, chaque agent *contexte* se retrouve doté d'une liste d'agents *contexte* supérieurs, d'une liste d'agents *contexte* inférieurs, et de l'aptitude à fournir et à mettre à jour son niveau. Pour évaluer son gain estimé relatif aux autres agents *contexte*, un agent *contexte* utilise alors la valeur de son niveau comme valeur du gain estimé. Ainsi, il est capable d'associer à sa proposition d'actions une valeur de gain estimé telle que tout agent *contexte* qui lui est supérieur propose un gain estimé supérieur, tandis que tout agent *contexte* inférieur propose un gain estimé inférieur.

Lorsqu'un agent *contexte* C est créé, cela signifie qu'un utilisateur a effectué une action. Par conséquent, tous les agents *contexte* valides à ce moment là sont apparemment moins bons que le nouvel agent *contexte*, puisqu'ils n'ont pas réussi à satisfaire les utilisateurs (ils n'ont pas été choisis par l'agent *contrôleur*). Par conséquent, l'agent C initialise sa liste d'agents inférieurs LC_{inf} avec ces agents. Ceux-ci ajoutent quand à eux ce nouvel agent C dans leur liste d'agents *contextes* supérieurs LC_{sup}. L'initialisation d'un agent *contexte* avec sa liste d'agents inférieurs, et l'ajout éventuels d'agents supérieurs par la suite au fur et à mesure de l'apparition d'autres agents *contexte*, crée progressivement des hiérarchies entre les agents *contexte*. Ce mécanisme d'ordonnancement entre agents *contexte* permet donc à l'agent *contrôleur* d'évaluer, dans une situation donnée, l'action dont le système croit qu'elle satisfera au mieux les utilisateurs, et donc de résoudre cette SNC en choisissant l'agent *contexte* proposant le gain estimé le plus élevé.

Une précision importante est à apporter concernant le calcul du gain estimé. L'utilisation du mécanisme d'auto-ordonnancement entre agents assure que si un agent *contexte* est supérieur à un autre, alors son gain estimé sera supérieur à celui de cet autre agent. Il est cependant possible que ces deux agents ne possèdent aucune relation d'ordre, la supériorité de la valeur du gain estimé du premier agent n'étant qu'un hasard. Dans ce cas, il n'y a aucune raison de favoriser un agent plutôt qu'un autre. Par conséquent, si le gain estimé d'un agent *contexte* est supérieur à celui d'un autre agent *contexte*, c'est soit parce que le premier agent *contexte* est effectivement supérieur au second agent *contexte*, auquel cas il faut choisir le premier agent *contexte*, soit par pur hasard, auquel cas il revient au même de choisir le premier ou le second agent *contexte*. C'est pourquoi il est inutile de vérifier si la supériorité en terme de gain estimé est un hasard ou pas ; dans tous les cas, c'est l'agent possédant le gain estimé le plus élevé qui est choisi. Si celui-ci est contredit par la suite, une relation d'ordre sera alors créée avec l'autre agent qui lui deviendra supérieur.

3.2.4.4 Une action proposée par un agent *contexte* est contredite

Problématique Lorsqu'un agent *contexte* ayant proposé une action à l'agent *contrôleur* est sélectionné, l'agent *contrôleur* ordonne l'application de cette action sur l'effecteur associé. L'agent *contrôleur* pense que la réalisation de cette action dans la situation courante va satisfaire les utilisateurs. Cependant, l'agent *contexte* peut se tromper, soit parce qu'il a proposé une action dans la mauvaise situation, soit parce que cette action n'est plus appropriée à cette situation du fait des changements de préférences des utilisateurs. L'agent *contrôleur* va donc se retrouver incapable de décider de la bonne action à réaliser à cause des informations erronées envoyées par ses agents *contexte*, ce qui l'amènera à être contredit ; il est donc en situation d'incompétence.

Détection En l'absence de retour explicite des utilisateurs sur leur satisfaction, l'agent *contrôleur* doit être en mesure de détecter si l'action qu'il a réalisée est contredite ou pas par un utilisateur (les utilisateurs sont satisfaits ou pas). En effet, si une action du système est contredite par l'utilisateur, nous pouvons supposer que le système a eu tort de réaliser cette action, tandis que si elle ne l'est pas, alors le système a bien agi.

Soit une action a réalisée par un agent *contexte* ; l'agent *contrôleur* doit alors déterminer si l'action réalisée par un utilisateur suite à l'action a est une contradiction, ou une action indépendante. Pour cela, nous définissons une action a_c comme une contradiction de l'action a si elle vérifie les deux conditions suivantes :

1. L'action a_c est réalisée après a, sans qu'aucune autre action du système ou d'un utilisateur n'ait été réalisée entre-temps sur ce même effecteur ;
2. La situation dans laquelle a a été réalisée est "similaire" à celle qui suit la réalisation de a_c.

Une action est donc contredite si l'action "contradictoire" ramène l'environnement dans un état similaire à celui ayant précédé cette action. Dans un environnement statique, la seconde condition aurait été simplifiée : la situation précédant l'action a serait identique à la situation suivant l'action de contradiction a_c. N'étant pas dans ce cas de figure, la réaction des utilisateurs suite à l'action erronée réalisée par le système n'est pas instantanée. Même si ce laps de temps peut être relativement court, la dynamique de l'environnement ambiant fait que la situation aura pu évoluer entre-temps, et donc la situation suivant la réalisation de la contradiction a_c ne pourra pas être parfaitement identique à la situation dans laquelle l'action a a été réalisée.

Un agent *contrôleur* n'a de notion de situation qu'au travers des propositions envoyées par les agents *contexte*. Aussi, de son point de vue, une situation ne se caractérise pas par l'état courant des données perçues, mais par l'ensemble des agents *contexte* valides et non valides. Par conséquent, si deux situations S_1 et S_2 sont similaires, alors les mêmes agents *contexte* seront valides dans chacune de ces situations.

Lorsqu'une action a_1 est réalisée, l'agent *contrôleur* enregistre la liste des agents *contexte* valides dans cette situation (avant la réalisation de l'action a_1). Par la suite, au moment où l'action suivante a_2 est réalisée, l'agent *contrôleur* compare la liste des agents *contexte* alors valides (après la réalisation de l'action a_2) avec celle précédemment mémorisée. Si ces

deux listes sont identiques, l'agent *contrôleur* considère que la seconde action a_2 était une contradiction de la première action a_1.

Résolution Le traitement réalisé pour résoudre cette SNC dépend du type de la contradiction. Trois types de contradiction peuvent se présenter, selon que les actions a_1 et a_2 ont été réalisées par le système ou par un utilisateur :

- l'action a_2, réalisée par un utilisateur, contredit l'action a_1 réalisée par le système ;
- l'action a_2, réalisée par le système, contredit l'action a_1 réalisée par un utilisateur ;
- l'action a_2, réalisée par le système, contredit l'action a_1 réalisée aussi par le système.

Pour les deux premiers types de contradictions, peu importe si c'est l'utilisateur qui contredit le système, ou le système qui contredit l'utilisateur. Dans les deux cas, nous considérons que l'action de l'utilisateur prévaut toujours sur celle du système ; c'est donc l'agent *contexte* qui est contredit.

Nous présentons d'abord une première solution basée sur un mécanisme d'ajustement et permettant de résoudre cette SNC pour les deux premiers types de contradiction. Si cette solution s'avère inapplicable, une seconde solution basée sur un mécanisme de réorganisation est proposée. Enfin, nous présentons la méthode de résolution employée pour le troisième type de contradiction, où une action d'*Amadeus* contredit une de ses propres actions. En revanche, nous ne considérons pas le cas d'une contradiction d'un utilisateur par un utilisateur ; dans le pire des cas, cette situation entrainera la création de deux agents *contexte* qui mèneront le système à se contredire lui-même, et donc au cas de l'auto-contradiction du système.

Première solution La première solution consiste à faire en sorte que si la même situation S se présente, l'agent *contexte* contredit ne soit pas valide afin qu'il ne soumette pas à nouveau sa proposition d'action dans une situation similaire. Il s'agit donc d'exclure la situation courante des plages de validité de l'agent *contexte*. Une situation étant représentée par l'ensemble des états de chaque variable perçue à un instant donné, il suffit d'exclure la valeur courante d'au moins une plage de validité pour que la situation soit elle-même exclue.

L'AVRT, qui est utilisé pour modéliser chaque plage de validité, propose un certain nombre de mécanismes énoncés plus en détail dans le chapitre 4.1. En particulier, il propose un mécanisme d'exclusion de valeur qui, étant donnée une valeur v appartenant à la plage de valeur de l'AVRT, va tendre à exclure v. Cette exclusion n'est pas directe : selon la proximité de la valeur v avec une des bornes de la plage de valeur, l'AVRT peut réussir à l'exclure directement en modifiant une de ses bornes, ou bien uniquement tendre à l'exclure sans modifier suffisamment sa borne pour que v se retrouve en dehors de la plage de valeur. L'AVRT propose alors un second mécanisme qui, considérant cette même valeur v, sera capable de dire si l'AVRT est capable de l'exclure directement grâce à son mécanisme d'exclusion ou pas.

S'il existe au moins une plage de validité chez l'agent *contexte* pour laquelle le mécanisme d'exclusion de l'AVRT est applicable, cette solution est suffisante pour résoudre cette SNC. De plus, elle n'implique qu'un mécanisme de tuning, car elle n'entraîne de modifications que dans l'état interne de l'agent *contexte*. Dans la mesure où cette solution est applicable,

elle sera préférée à la seconde solution, cette-ci impliquant un mécanisme de réorganisation.

Seconde solution La seconde méthode de résolution consiste à modifier le gain estimé de l'agent *contexte* contredit. Lorsque l'agent *contrôleur* détecte une contradiction, il envoie un message à l'agent *contexte* contredit, dans lequel il lui précise la liste des agents *contexte* valides en même temps que lui.

Vu que son action s'est avérée inappropriée, l'agent *contexte* contredit considère qu'il aurait mieux valu que l'agent *contrôleur* sélectionne un autre agent *contexte* à sa place. Cependant, ni lui ni l'agent *contrôleur* n'est capable de déterminer *a priori*, parmi la liste des agents *contexte* qui étaient valides en même temps que lui, lequel aurait réellement était le plus appropriée à la situation. En revanche, l'agent *contexte* sait qu'il ne l'a pas été, aussi n'importe lequel de ces agents est susceptible de fournir une proposition d'action plus satisfaisante. C'est pourquoi il intègre l'ensemble de ces agents *contexte* à sa liste d'agents supérieurs. Plus de précisions sur le mécanisme d'auto-ordonnancement utilisé par les agents *contexte* sont présentés au chapitre 4.2.

Cet ajout entraîne la mise à jour automatique de sa valeur de gain estimé, qui est alors inférieure à celles de tous les agents *contexte* valides en même temps que lui lors de sa sélection. Ainsi, si cet agent *contexte* se retrouve valide dans la même situation que celle où il a proposé son action contredite, il est assuré (sauf changement ultérieur dans les préférences des utilisateurs) qu'au moins un autre agent *contexte* proposera une action accompagnée d'un meilleur gain estimé que le sien, et sera donc sélectionné à sa place.

Auto contradiction Dans le cas d'une contradiction *Amadeus*/utilisateur, nous considérons que l'action de l'utilisateur est prépondérante, et donc que la contradiction concerne forcément l'agent *contexte*. En revanche, dans le cas d'une auto-contradiction d'*Amadeus*, deux agents *contexte* sont concernés, sans information permettant de savoir lequel d'entre eux est à l'origine de la mauvaise proposition d'action. Plusieurs solutions sont alors envisageables pour déterminer l'agent *contexte* responsable de cette SNC : choisir au hasard un des agents, ou bien choisir l'agent le moins confiant, ou au contraire le plus confiant (la notion de confiance intégrée aux agents *contexte* sera introduite plus en détail en section 3.2.4.5), ou encore les considérer les deux comme responsables.

Toutes ces solutions présentent forcément un inconvénient. La dernière solution nous parait pourtant être la plus pertinente. En effet, nous considérons qu'il est plus important d'éviter que le système agisse mal (quitte à ce qu'il n'agisse pas du tout) plutôt que de faire en sorte qu'il agisse à tout prix (quitte à agir mal). En dégradant les deux agents *contexte*, nous empêchons l'agent *contexte* d'agir, mais nous empêchons aussi le bon agent *contexte* d'agir, cette dégradation de l'agent *contexte* correct étant toujours réparable par la suite. De plus, dans le cas extrême où les deux agents *contexte* ont tort, cette solution permet de résoudre en une fois les deux conflits.

La résolution de la SNC liée à l'auto-contradiction d'*Amadeus* est donc réalisée par les deux agents *contexte* et est identique à celle mise en oeuvre lorsque l'agent *contexte* est contredit par un utilisateur.

3.2.4.5 Deux agents *contexte* proposent la même action

Problématique Considérons n'importe quel agent *contexte* C_i. Lorsque cet agent *contexte* C_i est valide, il envoie une proposition d'action que nous notons P_i. Nous considérons pour l'instant que cette proposition se compose de deux éléments :

- Une description de l'action a_i proposée par C_i ;
- Un gain estimé g_i (voir section 3.2.4.3) représentant la pertinence de réaliser cette action lorsque C_i est valide.

Lorsqu'un agent *contrôleur* perçoit des propositions d'actions différentes (autrement dit, des affectations d'états différents), il compare les gains estimés intégrés à ces propositions d'actions afin de savoir quelle action semble la plus pertinente. Cependant, il est possible que deux agents *contexte* proposent dans une même situation la même action à réaliser. Cette situation pose problème dès lors que ces deux agents accompagnent leurs propositions d'actions avec une valeur de gain estimé différente.

Par exemple, nous supposons qu'un agent *contrôleur* reçoit des agents *contexte* C_1, C_2 et C_3 les propositions d'actions $P_1 = a_1, g_1$,$P_2 = a_2, g_2$ et $P_3 = a_3, g_3$. Nous supposons aussi l'hypothèse suivante :

$$a_1 = a_3 \text{ et } g_1 < g_2 < g_3$$

L'agent *contrôleur* doit donc choisir quelle action réaliser entre l'action a_1 (égale à l'action a_3) et l'action a_2. Or, l'action a_1 est associé à deux valeurs de gain estimé g_1 et g_3 différentes. Si l'agent *contrôleur* considère que l'action a_1 est associée au gain estimé g_1, comme le dit l'agent *contexte* C_1, il choisira de réaliser l'action a_2, dont le gain estimé g_2 est supérieur au gain estimé g_1. En revanche, si l'agent *contrôleur* considère que l'action a_1 est associé au gain estimé g_3, comme le dit l'agent *contexte* C_3, c'est l'action a_1 qui sera sélectionné, car son gain estimé g_3 est supérieur au gain estimé g_2.

Quel agent *contexte*, entre c_1 et c_3, l'agent *contrôleur* doit-il croire pour savoir si l'action qu'ils proposent est préférable à celle de c_2 ? Il s'agit ici d'une SNC de concurrence entre les agents *contexte* proposant la même action mais avec un gain estimé différent.

Détection La détection de cette SNC est réalisée par l'agent *contrôleur* associé aux agents *contexte* en concurrence, lorsqu'il reçoit leurs propositions.

Résolution Pour résoudre cette SNC, nous avons intégré aux agents *contexte* la notion de confiance. Plus un agent *contexte* possède une confiance élevée, plus il croit que l'action qu'il propose satisfera les utilisateurs si elle est sélectionnée par l'agent *contrôleur*.

La confiance d'un agent *contexte* est modélisée par une valeur numérique comprise entre 0 et 1 (0 signifiant une absence totale de confiance, et 1 une certitude totale). Initialement, la confiance d'un agent *contexte* est initialisée à 0,5. Celle-ci augmente au fur et à mesure que l'agent *contexte* est sélectionné et que l'action qu'il a proposé n'est pas contredite. En revanche, elle diminue si l'agent *contexte* est sélectionné et si l'action qu'il a proposée est contredite.

Plus précisément, le calcul de la confiance T_t d'un agent *contexte* au temps t se base sur sa confiance T_{t-1} au temps $t-1$ et sur le retour R qu'il a reçu. Ce calcul est paramétré par une valeur réelle $\lambda \in [0;1]$ qui représente l'impact du retour sur le calcul du nouveau niveau de confiance. La formule 3.1 fournit précisément le calcul de mise à jour de la confiance :

$$T_{t+1} = T_t * (1 - \lambda) + R * \lambda \tag{3.1}$$

Un retour proche de 1 augmente la confiance de l'agent *contexte*, tandis qu'un retour proche de 0 la diminue. Le paramètre λ représente donc la dynamique supposée de l'environnement dans lequel est situé l'agent : plus sa valeur est importante, plus le niveau de confiance évolue rapidement en fonction des feedbacks. Avec un paramètre λ fixé à 0.1, le dernier retour reçu n'influence donc que de 10% la valeur du niveau de confiance. Ainsi, un agent *contexte* accorde plus d'importance au niveau de confiance qu'il possède qu'au feedback qu'il a reçu, ce qui limite l'impact des éventuelles actions non pertinentes des utilisateurs. En revanche, si un feedback est envoyé plusieurs fois successivement, il fini par modifier significativement le niveau de confiance de l'agent.

Pour résoudre la SNC de concurrence entre des agents *contexte* proposant la même action, chaque agent *contexte* associe son niveau de confiance à sa proposition d'action. S'il est sélectionné, l'agent *contrôleur* envoie alors un retour égal à 0 à l'agent *contexte* si son action a été contredite, et 1 sinon. L'agent *contexte* utilise alors ce retour pour mettre à jour son niveau de confiance.

Ainsi, lorsque l'agent *contrôleur* doit estimer quelle proposition est la plus intéressante pour satisfaire les utilisateurs, il commence par s'assurer que deux agents *contexte* ne proposent pas la même action au même moment avec des prévisions différentes. Si ce cas se présente, il choisit l'agent *contexte* dont le niveau de confiance est le plus élevé.

3.2.4.6 Envoi de mises à jour inutiles à des agents *donnée*

Problématique Chaque agent *capteur* (ou *effecteur*) génère des données potentiellement utiles notamment aux agents *contexte*. Un agent *donnée* au sein d'un AMAS *dispositif* est associé à un agent *capteur* (ou *effecteur*), et son objectif est de transmettre les données issues de l'agent *capteur* (ou *effecteur*) qui lui est associé aux agents *contexte* locaux.

Un agent *donnée* représente donc la connaissance au niveau d'un AMAS *dispositif* de l'état d'une variable, que celle-ci provienne d'un agent *capteur* (ou *effecteur*) local ou distant. Cette connaissance est mise à jour par envoie de message, en provenance de l'agent *capteur* (ou agent *effecteur*) associé à cet agent *donnée*. Cependant, si l'agent *capteur* (ou *effecteur* envoie une mise à jour alors que la donnée à envoyer n'a pas évolué depuis la dernière mise à jour, cet agent se retrouve en SNC d'inutilité.

Détection La détection peut se faire au niveau de l'agent *donnée* qui reçoit une mise à jour identique à la dernière mise à jour reçue. Cependant, elle peut aussi se faire par l'agent *capteur* (ou *effecteur*) si celui-ci garde en mémoire la dernière mise à jour envoyée.

Résolution La résolution de cette SNC se fait grâce au mécanisme de communication spontanée mis en place dans le comportement des agents *capteur* et *effecteur*. Plutôt que d'envoyer sa mise à jour à un agent *donnée* lorsque ce dernier la lui demande (ce qui risque d'entraîner des msies à jour inutiles), il va envoyer sa mise à jour lorsqu'il jugera cela pertinent. En l'occurrence, pour prévenir cette SNC d'inutilité, il jugera que l'envoie d'une mise à jour est pertinente si la valeur courante de la donnée est différente de celle de la dernière mise à jour. La résolution de cette SNC est donc directement intégré au comportement nominal de l'agent *donnée*.

3.2.4.7 Mises à jour pertinentes non envoyées à des agents *donnée*

Problématique Cette SNC est l'opposée de la SNC précédente : il s'agit d'une SNC d'improductivité qui se présente si un agent *contexte* ne produit pas le comportement adéquat à cause de l'agent *capteur* (ou *effecteur*), celui-ci n'ayant pas envoyé la dernière mise à jour à l'agent *donnée* associé à cet agent *contexte* malgré le changement de valeur de cette donnée.

Détection La détection de cette SNC ne peut se faire que par l'agent *capteur* (ou *effecteur*) qui, en gardant en mémoire la dernière mise à jour envoyée, peut détecter que la dernière donnée envoyée diffère de la donnée courante.

Résolution Cette SNC est résolue de la même façon que la SNC précédente, grâce au mécanisme de communication spontanée intégré dans le comportement nominal de l'agent *donnée*. Dès qu'un changement de valeur est détecté, l'agent *capteur* (ou *effecteur*) envoie spontanément une mise à jour aux agents *donnée* qui lui sont associés.

3.2.4.8 Envoie de données inutiles à des agents *contexte*

Problématique Il est parfois possible que la connaissance de la valeur d'une variable pour un agent *contexte* soit inutile, voire dégrade son fonctionnement. En effet, si une situation ayant précédemment mené à la réalisation d'une action par un utilisateur (et donc à la création d'un agent *contexte*) se présente à nouveau, mais avec seulement l'état d'une variable inutile différent, l'agent *contexte* interprètera à tort qu'il s'agit d'une situation différente. Par exemple, si un agent *contexte* considère qu'il faut allumer la lumière quand un utilisateur entre dans la pièce, un changement du niveau d'humidité perçue par un capteur d'humidité (variable inutile) amène cet agent à se considérer dans une nouvelle situation, et donc à ne pas proposer son action.

Une SNC apparaît donc lorsqu'un agent *donnée* envoie sa valeur à un agent *contexte* qui n'en a pas besoin. Il s'agit d'une SNC de conflit entre l'agent *donnée*, dont l'objectif est de transmettre sa donnée aux agents *contexte* qui en ont besoin, et l'agent *contexte* qui peut se retrouver à se croire invalide à tort du fait de la présence de cette donnée inutile.

Détection Nous avons défini un processus local impliquant l'agent *donnée* et les différents agents *contexte* de son voisinage, afin d'évaluer non pas l'inutilité de la variable, mais plutôt

son utilité. L'objectif recherché est alors d'estimer le niveau d'utilité de la variable en fonction des différents effecteurs, pour ensuite considérer comme inutile une variable présentant un niveau d'utilité trop faible vis-à-vis d'un effecteur, et ainsi prévenir l'apparition de cette SNC.

Un agent *contexte* est créé chaque fois qu'un utilisateur réalise une action sur un effecteur. Cet agent associe l'action réalisée à une description de la situation dans laquelle l'utilisateur l'a réalisée (voir chapitre 3.2.3.4). Cette description est composée de l'ensemble des valeurs des données perçues au moment où l'action a été réalisée. Quand un agent *contexte* devient valide, il considère qu'il est dans la même situation que celle ayant mené à sa création ; il propose alors son action à l'agent *contrôleur*.

L'invalidité d'un agent *contexte* peut-être justifiée par deux causes : (i) il se peut qu'il soit invalide tout simplement parce que la situation courante n'est pas la même que celle de sa création ; (ii) il se peut aussi qu'une donnée inutile appartienne à la description de sa situation et, possédant une valeur différente de celle qu'elle avait au moment de la création de l'agent *contexte*, amène celui-ci à croire à tort qu'il est invalide. Pour savoir si une variable qui l'invalide est inutile, un agent *contexte* doit savoir dans laquelle de ces deux situations il se trouve.

Considérons deux agents *contexte* V et I. Dans une situation donnée, l'agent V est valide, alors que l'agent I ne l'est pas. L'agent V envoie alors sa proposition d'action à l'agent *contrôleur* qui le sélectionne et dit à l'agent *effecteur* associé de réaliser l'action proposée par V. De son côté, l'agent I observe son état interne afin d'évaluer s'il a eu raison de ne pas envoyer aussi sa proposition d'action, autrement dit s'il a bien fait de ne pas être valide en même temps que l'agent V.

Considérons l'ensemble des données D perçues par les agents *contexte* V et I, et le sous-ensemble $D_i \in D$ des données invalides pour l'agent *contexte* I. Lorsque l'agent *contexte* I détermine qu'il a eu raison d'être invalide dans une situation particulière, il sait que c'est grâce aux données de D_i, invalides dans cette situation. Par exemple, supposons les données d_1 et $d_2 \in D_i$ invalides dans la situation où l'agent I a eu raison d'être invalide. L'agent I sait qu'il doit son invalidité à d_1 et/ou à d_2. Cependant, s'il est certain qu'au moins une de ces données est utile pour décrire la situation courante, il est possible que l'autre soit une donnée inutile, mais invalide par simple coïncidence dans la situation courante. N'étant pas capable de différencier les données vraiment utiles de celles qui ne le sont pas, l'agent I envoie un "signal d'utilité" à chacun des agents *donnée* invalides appartenant à D_i afin de les informer de leur potentielle utilité. Ce traitement de génération des signaux d'utilité sera traité au paragraphe suivant.

Afin d'évaluer son utilité, chaque agent *donnée* perçoit les signaux d'utilité envoyés par les différents agents *contexte* auxquels il est relié. Le but d'un agent *donnée* est alors de traiter ces signaux afin d'établir si la réception de ces signaux n'est qu'une coïncidence (due au hasard des combinaisons de valeurs des données) ou s'il a rendu invalide l'agent *contexte* à l'origine du signal d'utilité au bon moment car il lui est effectivement utile. Si un agent *donnée* est incapable de prouver son utilité vis-à-vis du contrôle d'un effecteur malgré les différents signaux d'utilité reçus, il finit par se considérer comme inutile. La SNC d'improductivité est alors détectée par cet agent *donnée*.

Dans la section suivante, nous présentons le traitement réalisé par l'agent *contexte* pour déterminer quand il a raison d'être invalide et quand il a tort, autrement dit le traitement menant à la génération des signaux d'utilité. Puis dans la section suivante, nous présentons le processus réalisé par les agents *donnée* pour traiter ces signaux d'utilités afin d'établir l'utilité (ou l'inutilité) de leur variable.

Génération des signaux d'utilité Pour expliquer le processus de génération des signaux d'utilité, nous considérons que l'agent *contexte* V est l'agent valide qui a été sélectionné par le contrôleur dans la situation S, et que son action a été réalisée sans être contredite. Nous considérons aussi un autre agent *contexte* I qui, lui, est invalide dans la situation S à cause du sous-ensemble D_i de données invalides parmi l'ensemble des données D perçues par l'agent I. Pour établir la potentielle utilité des agents D_i, l'agent *contexte* I va faire l'hypothèse que la totalité des agents D_i sont inutiles. Autrement dit, il suppose qu'il aurait dû être valide en même temps que l'agent V. Si cette hypothèse entraîne l'apparition d'une contradiction, elle doit être considérée fausse, et donc au moins une donnée appartenant à D_i est utile.

Une contradiction apparaît lorsque la proposition d'action de l'agent I est incompatible avec celle qu'a envoyé l'agent V. Considérons la proposition P_I de l'agent I constituée de l'action a_I et du gain estimé g_I , et la proposition P_V de l'agent V constituée de l'action a_V et du gain estimé g_V. Une contradiction apparaît lorsque I et V proposent deux actions différentes avec un gain estimé de I supérieur au gain estimé de V, autrement dit lorsque :

$$a_I \neq a_V \text{ et } g_I > g_V \Rightarrow CONTRADICTION \tag{3.2}$$

En effet, si l'agent *contexte* I avait été valide en même temps que l'agent *contexte* V, le gain estimé de I étant supérieur à celui de V, c'est lui qui aurait été sélectionné. Or, l'agent V n'ayant pas été contredit, nous considérons que son action A_V était correcte. L'agent I a donc eu raison d'être invalide dans cette situation, et donc il existe au moins une donnée utile parmi l'ensemble des données D_i qui rendait I invalide.

Plus concrètement, supposons que les agents *contexte* V et I sont associés à une lampe. L'agent I propose de l'allumer avec un gain estimé fort, tandis que l'agent V propose de la laisser éteinte avec un gain estimé moyen. Nous supposons que l'agent V est valide lorsque la lampe est éteinte, lorsque le capteur de luminosité fournit une valeur forte, et lorsque le capteur d'humidité (donnée inutile) fournit lui aussi une valeur forte.

$$V_{valide} \Rightarrow \neg Lampe \wedge Luminosite^{+} \wedge Humidite^{+}$$

Supposons que l'agent I est valide si la lampe est éteinte et si la luminosité et l'humidité sont faibles.

$$I_{valide} \Rightarrow \neg Lampe \wedge Luminosite^{-} \wedge Humidite^{-}$$

Lorsque V est sélectionné, les données portant sur la luminosité et sur l'humidité empêchent I d'être valide. L'action de V n'étant pas contredite, I a eu raison de ne pas proposer

son action en même temps. Il existe donc au moins une donnée utile entre la donnée portant sur la luminosité et la donnée portant sur l'humidité.

L'agent I ne peut déterminer quelles données de D_i sont réellement utiles, et quelles données sont inutiles mais invalides par hasard en même temps que des données utiles. Dans notre exemple, notamment, seule la luminosité est utile ; l'humidité n'est invalide dans cette situation que par hasard. C'est pourquoi, chaque fois que l'agent *contexte* I détecte une contradiction, il se limite à prévenir chacun des agents *donnée* invalides en leur envoyant un "signal d'utilité". Ce signal signifie que l'agent *contexte* a détecté que ces agents *donnée* sont "peut-être" utiles.

Les agents *contexte* sont donc capables de détecter et d'associer des situations ou informations sur l'utilité des variables. Cependant, ces informations concernent l'utilité de ces variable vis-à-vis des agents *contexte*, alors que nous cherchons à détecter l'inutilité de ces données. C'est pourquoi, nous décrivons maintenant comment les agents *donnée* traitent les différents *signaux d'utilité* reçus au cours du temps pour déterminer s'ils sont inutiles.

Traitement des signaux d'utilité Les agents *contexte* sont regroupés en différents ensembles, chaque ensemble d'agents *contexte* étant associé à un unique agent *contrôleur*, lui-même en charge d'un unique effecteur. La présence de plusieurs effecteurs sur un même dispositif entraîne alors la présence de plusieurs ensembles d'agents *contexte* séparés en fonction de l'effecteur associé. Lorsqu'un agent *donnée* reçoit un message d'utilité, ce signal informe donc l'agent d'une possible utilité de sa variable vis-à-vis de l'effecteur associé à l'agent *contexte* ayant envoyé ce signal. Le traitement présenté ici est donc réalisé séparément pour chaque ensemble d'agents *contexte*.

La solution la plus naturelle pour exploiter ces signaux pourrait être de considérer qu'un agent *donnée* considéré fréquemment comme utile (c'est-à-dire recevant fréquemment des signaux d'utilité), est probablement utile, alors qu'un agent *donnée* ne recevant que rarement des signaux d'utilité ne l'est pas. Cependant, ce raisonnement s'avère ne pas toujours être correct. Prenons un exemple simple, avec trois agents *données*, l'un d'eux étant inutile. Il est alors possible que, sur une base de 100 signaux d'utilité envoyé par les agents *contexte*, 50 sont envoyés à un agent *donnée* utile et l'agent *donnée* inutile, et 50 sont envoyé à l'autre agent *donnée* utile et toujours à l'agent *donnée* inutile. L'agent *donnée* inutile a donc reçu un total de 100 signaux d'utilité, tandis que chaque agent *donnée* utile n'en a reçu que 50. Cet exemple basique met en évidence la facilité avec laquelle cette solution peut faire défaut.

Pour répondre au problème de l'inutilité d'une variable, nous avons conçu un mécanisme de filtrage de variables inutiles. Ce mécanisme est présenté au chapitre 4.3. Il permet à un agent fournisseur de données (ici, les agents *donnée*) de traiter les signaux d'utilité envoyés par les agents qui exploitent ces données (ici, les agents *contexte*). Cet outil implique d'instancier aux niveaux des agents exploitant les données le mécanisme de génération des signaux d'utilité en fonction de l'action à réaliser, et exploite ces signaux de façon générique pour fournir une évaluation de l'utilité de la variable vis-à-vis de chaque action. En effet, certaines variable peuvent être utiles pour la réalisation d'une action sur un effecteur, mais inutile pour la réalisation d'une autre action sur ce même effecteur (par exemple, un capteur informant si le réservoir d'un distributeur d'eau est vide est utile pour savoir s'il faut

s'activer, mais inutile pour savoir s'il faut se désactiver). Pour rappel, une action consiste à une affectation d'un état à un effecteur, indépendamment de son état antérieur à l'action ; les actions d'allumer une lampe et de la laisser allumer sont donc considérées comme identique.

Lorsqu'un agent *donnée* considère que sa valeur est inutile vis-à-vis d'une action particulière, il en informe l'ensemble des agents *contexte* proposant cette action, et cesse alors de leur envoyer des mises à jour concernant cette valeur. Étant donné que cette donnée est devenue inutile pour eux, ces agents *contexte* suppriment la plage de validité (autrement dit, l'AVRT) associée à cette donnée.

Le traitement de détection et de résolution de cette SNC permet de réaliser un filtrage des données inutiles en cours de fonctionnement et sans connaissance *a priori*. Un prétraitement peut éventuellement être réalisé en amont du système grâce à des méthodes exploitant par exemple la sémantique des données pour accélérer ou simplifier le processus de filtrage, mais notre objectif est de permettre à *Amadeus* de pouvoir fonctionner même en l'absence de sémantique associée aux données. Notre filtrage des données inutiles assure le respect du critère de généricité de notre système, tout prétraitement ne permettant alors que d'optimiser les performances d'*Amadeus* sans être pour autant nécessaire à son fonctionnement.

3.2.4.9 Une nouvelle donnée est perçue par un agent *contexte* existant

Problématique La dynamique de l'environnement dans lequel se trouve un AMAS *dispositif* fait que de nouvelles données peuvent apparaître en cours de fonctionnement (ajout de nouveaux dispositifs). Ces nouvelles données ne posent pas de problème aux agents *contexte* qui vont apparaître par la suite. En revanche, les agents *contexte* existants ne sont pas capables de déterminer s'ils doivent être valides ou pas au vu de la valeur courante de cette donnée. Ils sont donc en situation d'ambiguïté.

Détection Cette SNC est détectée localement par chaque agent *contexte*, lorsqu'il reçoit une donnée pour laquelle il ne possédait jusqu'à présent aucune plage de validité.

Résolution Un agent *contexte* qui reçoit une nouvelle donnée est incapable de savoir quelle plage de validité attribuer à cette donnée. Si la plage de donnée choisie n'est pas la bonne, il se retrouvera à ne plus être valide dans les situations où il l'aurait été s'il n'avait pas connu cette donnée. C'est pourquoi, considérant qu'il était capable de déterminer sa validité sans connaître cette donnée, il va continuer à faire de même en attribuant une plage de validité pour cette donnée couvrant l'intégralité des valeurs que peut prendre cette donnée (de sa valeur minimale à sa valeur maximale). Ainsi, dans le meilleur des cas il continuera à fonctionner comme avant, et dans le pire des cas adaptera cette plage de validité au cours du temps en intégrant (voir section 3.2.4.2) ou en excluant (section 3.2.4.4) des situations de ses plages de validité. Il se peut aussi que d'autres agents *contexte* apparaissent par la suite et deviennent plus précis que lui, et donc sélectionnés à sa place.

3.2.4.10 Une donnée n'est plus perçue par un agent *contexte* existant

Problématique Tout comme pour la SNC précédente, la dynamique de l'environnement dans lequel se trouve un AMAS *dispositif* peut entraîner la disparition de données existantes en cours de fonctionnement (disparition de dispositifs existants). Pour les agents *contexte* existants, ils se retrouvent à nouveau en situation d'ambiguïté : n'ayant plus accès à la valeur de cette donnée, ils sont incapables de déterminer si la plage de validité de cette donnée est valide ou pas, et sont donc incapables de déterminer leur propre état de validité.

Détection Cette SNC est détectée par l'agent *donnée* associé à la donnée ayant disparu (nous supposons que le gestionnaire de contexte est capable de gérer la disparition physique d'un dispositif, l'agent *donnée* ne faisant qu'interpréter cette disparition comme la disparition de la donnée).

Résolution Lorsqu'un agent *donnée* détecte cette SNC, il prévient les agents *contexte* de son voisinage. Dès lors, un agent *contexte* ne percevant plus une donnée existante est incapable de gérer sa validité vis-à-vis de cette donnée. S'il décidait de se considérer par défaut comme invalide, il deviendrait alors inutile. C'est pourquoi il décide juste de supprimer la plage de validité associée à cette donnée, de la même façon que si cette donnée avait été jugée inutile (voir section 3.2.4.8).

3.2.4.11 Un agent *donnée* n'envoie sa valeur à aucun agent *contexte*

Problématique Initialement, un agent *donnée* ignore pour quels agents *contexte* sa donnée est utile ; aussi il envoie cette valeur à tous les agents *contexte* auxquels il est associé. Cependant, si un agent *donnée* découvre que sa donnée est inutile pour l'ensemble des agents *contexte* de son voisinage (les agents *contexte* appartenant au même AMAS *dispositif* que lui), cela le mène dans une SNC d'inutilité.

Détection Cette SNC est détectée par l'agent *donnée* lui-même lorsque, suite au processus de filtrage des variables inutiles présenté section 3.2.4.8, il supprime de ses destinataires les derniers agents *contexte* qui semblaient avaient besoin de sa valeur au sein de l'AMAS *dispositif* auquel appartient l'agent *donnée*.

Résolution La résolution de cette SNC est réalisé par l'agent *donnée* qui, n'ayant plus de raison d'exister, se supprime lui-même.

3.2.5 Implémentation

La dernière phase de la méthode ADELFE consiste à implémenter le système multi-agent. Pour implémenter correctement un SMA, [Arcangeli *et al.*, 2013] propose de distinguer deux niveaux de conception :

1. La conception du SMA durant laquelle une description des différents types d'agents est produite, incluant la définition de leurs comportements et des mécanismes d'interactions qu'ils utilisent.

2. La conception micro-architecturale durant laquelle les abstractions définies durant la conception du SMA sont mises en place, en tenant compte des exigences propres à sa mise en application (cadencement des agents, synchronisation, architecture interne des agents, etc.). La conception micro-architecturale s'intéresse donc à la réalisation des agents, mais aussi de leur plateforme d'exécution.

La conception micro-architecturale de notre système passe donc par la conception de la plateforme d'exécution de nos agents, afin de leur fournir les mécanismes nécessaire à leurs comportements. Pour répondre à ce besoin, [Noel, 2012] a proposé le modèle de composants SpeAD basé sur trois types de composants :

- L'*écosystème* représente la plateforme d'exécution des agents ;
- L'*espèce* représente un type d'agent instanciable dans un écosystème donné ;
- Le *transverse* représente la connexion entre une espèce et son écosystème. Chaque agent exploite donc les transverses de son espèce pour interagir avec son environnement.

Nous avons donc défini un écosystème pour l'AMAS *dispositif*. Cet écosystème possède la capacité d'instancier cinq espèces différentes d'agents, une pour chacun des agents d'*Amadeus*, et assure l'exécution de l'ensemble de ces agents cycle par cycle. Il propose aussi à ses différentes espèces un certain nombre de transverses :

- Le *localSender* qui permet d'envoyer des messages locaux d'agent à agent (par exemple, un agent *contexte* qui envoi sa proposition d'action à son agent *contrôleur*) ;
- Le *localBroadcaster* qui permet d'envoyer des messages à un ensemble d'agents (par exemple, l'envoi d'une mise à jour à l'ensemble des agents *contexte*) ;
- Le *globalBroadcaster* qui permet d'envoyer un message à un ensemble d'agents appartenant à d'autres AMAS (par exemple, lorsqu'un agent *capteur* transmet sa mise à jour aux agents *donnée* des autres AMAS *dispositif*) ;
- Le *connecter* qui permet de connecter de façon transparente un agent avec un élément extérieur à l'écosystème (par exemple, un agent *effecteur* connecté à son effecteur physique pour en percevoir ou modifier l'état) ;
- Le *receiver* qui permet de recevoir un message de façon transparente (par exemple, une mise à jour pour un agent *donnée*, indépendamment de la localité de l'agent *capteur* qui lui a envoyé cette mise à jour) ;
- Le *creator* qui permet d'instancier un nouvel agent en cours de fonctionnement (par exemple, un agent *contrôleur* qui crée un nouvel agent *contexte* suite à une action d'un utilisateur).

Il est important d'assurer la séparation claire entre la conception du SMA et la conception micro-architecturale, afin de bien distinguer les problématiques appartenant à l'un ou l'autre de ces deux niveaux. Par exemple, la problématique liée à l'initialisation d'un AMAS *dispositif* sur un dispositif (qui implique l'instanciation d'agents *capteur* et *effecteur* en fonction des capteurs et effecteurs présents sur ce dispositif) appartient au niveau micro-architecture. C'est la raison pour laquelle ces agents apparaissent sans qu'il s'agisse d'un mécanisme d'évolution du système découlant de la résolution d'une SNC.

Il en est de même pour l'arrivée d'une donnée au niveau d'un AMAS *dispositif* provenant d'un autre AMAS *dispositif*. Celle-ci est reçue par l'architecture du système, mais aucun agent ne perçoit cette donnée. L'absence d'agent *donnée* associé est détectée au niveau de l'infrastructure du système multi-agent. Cette problématique relève du niveau micro-architectural, c'est pourquoi c'est directement l'infrastructure qui crée l'agent *donnée* capable de gérer cette nouvelle *donnée*. Par la suite, la gestion de cette nouvelle donnée par les agents *contexte* redevient une problématique au niveau macro-architectural (voir chapitre 3.2.4.9).

La conception micro-architecturale de notre système nous a ensuite mené à l'implémentation elle-même de l'AMAS *dispositif*. Il a été implémenté grâce à l'outil *Make Agent Yourself* (MAY) qui permet de décrire l'architecture à composants produite durant la phase implémentation d'ADELFE grâce au langage SpeADL, puis d'implémenter au coeur des espèces de cette infrastructure le comportement des agents défini durant la phase de conception du SMA.

Enfin, concernant *Amadeus* en tant qu'AMAS formé d'AMAS *dispositif*, son écosystème n'a pas à être défini de la même façon. En effet, appliquer *Amadeus* dans un système ambiant passe par l'instanciation d'un AMAS *dispositif* par un utilisateur. Quant aux interactions entre ces AMAS, elles sont gérées par le gestionnaire de contexte. C'est donc le gestionnaire de contexte qui fait office d'écosystème pour *Amadeus*.

3.3 Synthèse

Ce chapitre présente la contribution essentielle de notre travail, à savoir le système *Amadeus*. Nous avons conçu *Amadeus* comme un AMAS d'AMAS : chaque agent *dispositif* qui compose *Amadeus* est lui-même un AMAS. L'objectif de chaque AMAS *dispositif* est d'attribuer un comportement correct au dispositif qui lui est associé. Pour atteindre leur objectif, les différents AMAS *dispositif* coopèrent entre eux pour s'échanger les données nécessaires à leur fonctionnement respectif.

Nous avons ensuite décrit les différents types d'agents qui composent un AMAS *dispositif* (les agents *capteur*, *effecteur*, *donnée*, *contexte* et *contrôleur*). Nous avons présenté le but local des différents types d'agents définis, ainsi que leur comportement nominal. Enfin, nous avons énoncé l'ensemble des Situations Non Coopératives auxquelles ces agents peuvent être confrontés durant leur fonctionnement, puis expliqué comment ces différents agents sont capables de détecter et résoudre ces SNC. Les interactions entre les AMAS *dispositif* (par lesquels ces AMAS s'échangent leurs données) ont alors été réifiées au niveau des agents *capteur* et *effecteur* d'une part et des agents *donnée* d'autre part.

Un AMAS *dispositif* (i) observe toutes les actions réalisées par les utilisateurs sur les effecteurs présents sur le dispositif auquel il est associé, (ii) mémorise les situations où ces actions ont été réalisées, puis (iii) utilise ces connaissances pour décider dynamiquement, face à des situations similaires à celles rencontrées précédemment, quelle action semble la plus appropriée. Il perçoit également des données depuis ses capteurs qu'il transmet aux autres AMAS *dispositif*, à moins que l'un d'eux ne l'informe que l'une de ces données lui est inutile (auquel cas il cesse de la lui envoyer).

Si nous observons plus en détail le comportement des agents qui composent un AMAS *dispositif*, nous distinguons l'agent *contrôleur* qui décide de l'action à réaliser la plus appropriée à partir des propositions d'actions envoyées par les agents *contexte*. L'ensemble de ces agents *contexte* représente les actions précédentes enregistrées. Les agents *capteur* et *effecteur* assurent la transmission de l'état de leur capteur ou effecteur associé aux agents *donnée* locaux, ainsi qu'aux agents *donnée* appartenant à d'autres AMAS *dispositif*. Enfin, ces agents *donnée* établissent la pertinence de leur donnée vis-à-vis des agents *contexte* de leur voisinage.

L'apprentissage réalisé par un AMAS *dispositif* ne résulte pas d'un algorithme d'apprentissage classique. Il s'agit du résultat du comportement coopératif de l'ensemble des agents qui le composent. En premier lieu, cet apprentissage résulte de la création des agents *contexte*. Ceux-ci représentent en effet des connaissances acquises par un AMAS *dispositif*, et la création de nouveaux agents *contexte* augmente donc la base de connaissances de cet AMAS *dispositif*. Cependant, contrairement à des algorithmes d'apprentissage plus classiques, notamment le CBR (voir section 2.2.1.7) où les cas précédents sont généralement enregistrés comme des connaissances statiques, un agent *contexte* représente une connaissance dynamique capable de s'auto-adapter si elle devient erronée (par prise en compte des contradictions). En particulier, ils sont capables d'intégrer (création d'une nouvelle plage de validité en cas de perception d'une nouvelle donnée) ou de supprimer (filtrage des variables inutiles) des variables en entrée en cours de fonctionnement. Bien que l'agent *contrôleur* soit en charge de décider de l'action à effectuer dans les différentes situations qui se présentent, il n'a pas à gérer directement les connaissances représentées par les agents *contexte* ; ceux-ci possèdent un comportement autonome et lui envoient leurs propositions d'actions quand ils le jugent opportun.

L'apprentissage réalisé par un AMAS *dispositif* est aussi le résultat des adaptations que réalisent les agents *contexte* au cours du temps. En effet, contrairement à certains algorithmes d'apprentissage plus classiques où il est nécessaire, pour tout changement dans l'environnement, de recommencer l'apprentissage depuis le début (et donc, d'enregistrer la totalité des cas qui se présentent au cours du temps), chaque agent *contexte* observe de façon autonome dans quelles situations sa proposition d'action devient inappropriée. Pour cela, il attend de savoir si cette action est acceptée. Si c'est le cas, la confiance de cet agent *contexte* augmente, ce qui renforce la connaissance que cet agent représente. Dans le cas où cette action est contredite par les utilisateurs, non seulement la confiance de l'agent *contexte* va diminuer, mais il va aussi tenter de s'améliorer (par exemple, en ajustant ses plages de validité pour ne plus être valide dans la situation où il était lorsqu'il a envoyé sa proposition d'action).

Enfin, l'apprentissage d'un AMAS *dispositif* est amélioré par un filtrage des données inutiles grâce aux interactions entre les agents *donnée* et les agents *contexte*. Si nous reprenons les catégories de filtrage de données présentées en section 4.3, nous constatons qu'il s'agit ici d'une méthode de type *embedded*, le filtrage étant directement intégré à l'AMAS *dispositif* au travers des agents *donnée*.

Le comportement d'*Amadeus* résulte des interactions entre les agents qui le composent (les différents AMAS *dispositif* présents dans le système ambiant). Le théorème de l'adéqua-

tion fonctionnelle, énoncé par [Camps, 1998], assure que le comportement global obtenu au niveau de notre système est fonctionnellement adéquat si ses différentes parties ont un comportement coopératif. Dans notre cas, les interactions coopératives entre les AMAS *dispositif*, autrement dit les interactions entre les agents *capteur* et *effecteur* d'une part et les agents *donnée* d'autre part, permettent à *Amadeus* de produire un comportement fonctionnellement adéquat. Ainsi, *Amadeus* apprend au cours du temps quel comportement attribuer aux différents dispositifs du système ambiant, cet apprentissage étant réalisé localement à chaque dispositif. C'est ce que nous vérifierons avec les expérimentations présentées dans le chapitre 5.

4 Contribution aux outils AMAS

La conception d'un AMAS mène le concepteur à étudier les SNC face auxquelles les différents agents du système sont susceptibles de se retrouver. La résolution de ces SNC peut être facilitée par l'utilisation de différents mécanismes proposées suite à la conception d'autres AMAS, tels que le mécanisme de communication spontanée de [Camps, 1998], ou l'AVT de [Lemouzy, 2011].

La résolution de certaines SNC lors de la conception d'*Amadeus* nous a amené à proposer un certain nombre de nouveaux mécanismes :

- l'*Adaptive Value Range Tracker* pour modéliser les plages de validité des agents *contexte* (section 3.2.3.4) ;
- le mécanisme d'auto-ordonnancement entre agents pour représenter le gain estimé des agents *contexte* (section 3.2.4.3) ;
- le mécanisme de filtrage des variables inutiles intégré aux agents *donnée* (section 3.2.4.8).

Nous avons fait en sorte de concevoir ces outils le plus indépendamment possible de l'application pour qu'ils puissent être utilisés dans d'autres contextes et d'autres systèmes. L'objectif est que ces outils soient suffisamment génériques pour aider à la mise en oeuvre d'autres AMAS.

4.1 Adaptive Value Range Tracker

L'AVRT (*Adaptive Value Range Tracker*, ou Traqueur de Plage de Valeurs Adaptatif) est une version étendue de l'AVT (*Adaptive Value Tracker*, ou Traqueur de Valeurs Adaptatif) proposé par [Lemouzy, 2011]. Nous commençons donc par présenter l'AVT, puis nous présentons l'AVRT tel que nous l'avons conçu dans le cadre de cette thèse.

4.1.1 Principe de l'Adaptive Value Tracker (AVT)

Le travail réalisée par [Lemouzy, 2011] dans le cadre de sa thèse a mené à la conception d'un outil appelé l'AVT (*Adaptive Value Tracker*). Il s'agit d'un composant logiciel générique dont l'objectif est de rechercher une valeur réelle dynamique comprise entre deux bornes $v \in [v_{min}; v_{max}]$ en se basant uniquement sur des retours (*feedbacks*) de types "plus petit", "plus grand" ou "correct" fournis par l'environnement. La particularité de l'AVT est qu'il

fonctionne même lorsque la valeur recherchée est dynamique, c'est-à-dire lorsqu'elle évolue au cours de la recherche. Un tel outil peut donc s'avérer très utile pour permettre à un agent d'ajuster une valeur au cours du temps, notamment un de ses paramètres par exemple.

Un AVT est constitué de deux variables v_t et Δ_t qu'il fait évoluer en fonction des feedbacks reçus depuis son environnement : la valeur v_t représente la valeur proposée par l'AVT à l'instant t, et la valeur Δ_t représente le pas de l'AVT à l'instant t. Un exemple de l'utilisation d'un AVT est illustré figure 4.1 :

1. Au temps t, l'AVT propose sa valeur v_t à son environnement ;
2. L'AVT reçoit un feedback en retour, ici le feedback "plus grand" ;
3. En accord avec le feedback reçu, l'AVT incrémente sa valeur v_t avec sa valeur Δ_t.

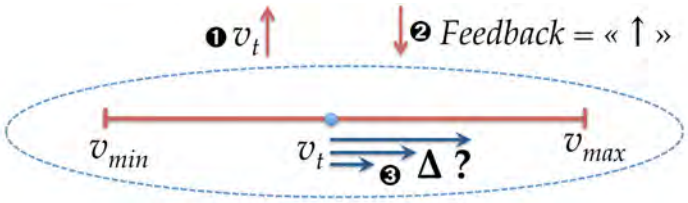

Figure 4.1 – Recherche d'une valeur dynamique par un AVT, extrait de [Lemouzy, 2011]

De façon générale, un AVT ajoute Δ_t à v_t lorsqu'il reçoit un feedback positif, et retire Δ_t à v_t lorsqu'il reçoit un feedback négatif. Concernant la variable Δ_t, elle évolue aussi en fonction des feedbacks reçus, selon les trois cas suivants :

1. L'AVT reçoit deux feedbacks successifs de même direction : le Δ augmente pour accélérer l'évolution de v (voir figure 4.2) ;

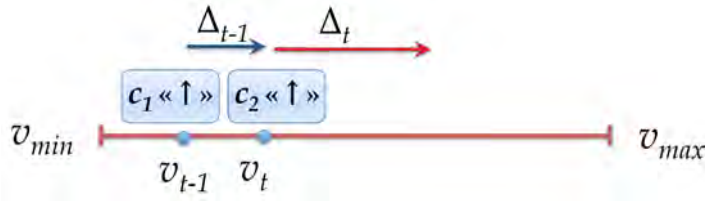

Figure 4.2 – Deux feedbacks successifs identiques augmentent Δ_t

2. L'AVT reçoit deux feedbacks successifs de directions différentes : le Δ diminue pour décélérer l'évolution de v (voir figure 4.3) ;

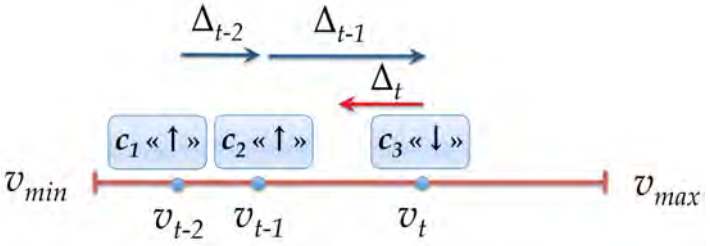

Figure 4.3 – Deux feedbacks successifs différents diminuent Δ_t

3. L'AVT reçoit un feedback de type "correct" : le Δ diminue pour décélérer l'évolution de v (voir figure 4.4).

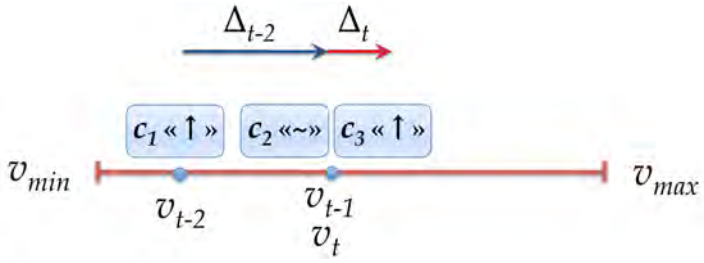

Figure 4.4 – Un feedback "correct" diminue Δ_t

4.1.2 Adaptive Value Range Tracker : une extension de l'AVT

Alors que l'AVT ne modélise qu'une seule valeur, l'AVRT permet de modéliser une plage de valeurs. L'objectif d'un AVRT est alors de rechercher les bornes d'une plage de valeurs. Il se compose pour cela de deux AVTs, l'AVT$_{min}$ et l'AVT$_{max}$ représentant respectivement les valeurs minimales et maximales de la plage de valeur. L'AVT$_{min}$ possède donc une valeur comprise entre v_{min} et la valeur courante de l'AVT$_{max}$, tandis que ce dernier possède une valeur entre la valeur courante de l'AVT$_{min}$ et v_{max}. Chaque fois que la valeur d'une des bornes évoluent, l'autre AVT met à jour sa valeur minimale ou maximale. Par exemple, si la valeur de l'AVT$_{min}$ change, l'AVT$_{max}$ modifie sa valeur maximale en lui attribuant la nouvelle valeur de l'AVT$_{min}$.

Un AVRT reçoit de son environnement un ensemble d'exemples $E = E_{in} \sqcup E_{out}$ formé par l'union disjointe entre, d'une part, l'ensemble des exemples E_{in} représentant les valeurs devant appartenir à la plage des valeurs recherchée, et d'autre part, l'ensemble des exemples E_{out} représentant les valeurs devant être extérieures à la plage des valeurs recherchée.

Considérons un exemple $e \in E$, et sa valeur v_e. A la réception de cet exemple, l'AVRT ajuste ses bornes en fonction de l'appartenance de e à l'ensemble des valeurs à intégrer E_{in} ou des valeurs à exclure E_{out}, et en fonction de l'appartenance (ou pas) de v_e à la plage de valeurs courante de l'AVRT. Trois cas peuvent alors se présenter :

- La valeur v_e n'appartient pas à la plage de valeur [AVT$_{min}$; AVT$_{max}$] alors que l'exemple e appartient aux exemples à intégrer E_{in}. Pour intégrer la valeur v_e de sa plage de valeur, l'AVRT regarde quelle borne est la plus proche de v et va envoyer un feedback à cette borne pour l'amener en direction de cette valeur. Par exemple, si $v_e > AVT_{max}$, l'AVRT envoie un feedback "plus grand" à AVT$_{max}$ pour agrandir d'un pas la plage de valeur. Inversement, si $v_e < AVT_{min}$, l'AVRT envoie un feedback "plus petit" à AVT$_{min}$ (voir figure 4.5) ;
- La valeur v_e appartient à la plage de valeur [AVT$_{min}$;AVT$_{max}$] alors que l'exemple e appartient aux exemples à exclure E_{out}. Pour exclure la valeur v_e de sa plage de valeur, l'AVRT regarde quelle borne est la plus proche de v et va envoyer un feedback à cette borne pour l'amener en direction de cette valeur. Par exemple, si la borne supérieure est la plus proche de v_e, l'AVRT envoie un feedback "plus petit" à AVT$_{max}$ pour réduire

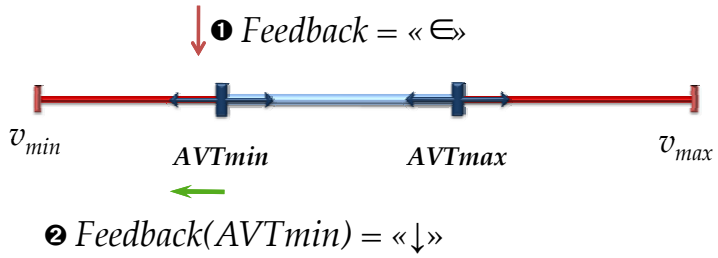

Figure 4.5 – Feedback d'intégration d'une valeur

d'un pas la plage de valeur. Inversement, si la borne inférieure est la plus proche de v_e, l'AVRT envoie un feedback "plus grand" à AVT_{min} pour réduire d'un pas la plage de valeur (voir figure 4.6) ;

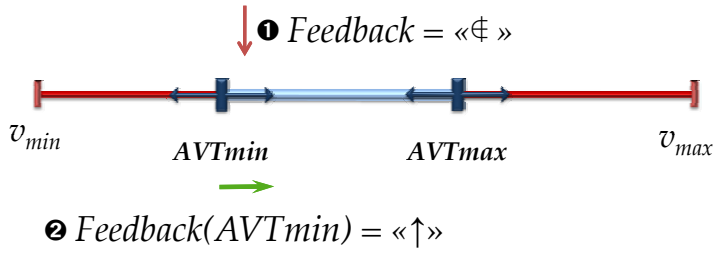

Figure 4.6 – Feedback d'exclusion d'une valeur

– La valeur v_e appartient à la plage de valeur $[\text{AVT}_{min}\,;\text{AVT}_{max}]$ et e appartient aux exemples à intégrer E_{in}, ou bien le valeur v_e n'appartient pas à la plage de valeur $[\text{AVT}_{min}\,;\text{AVT}_{max}]$ et e appartient aux exemples à exclure E_{out}. L'exemple e tend à confirmer la justesse des valeurs courantes des bornes. L'AVRT envoie donc un feedback "correct" à chacune de ses bornes (voir figure 4.7).

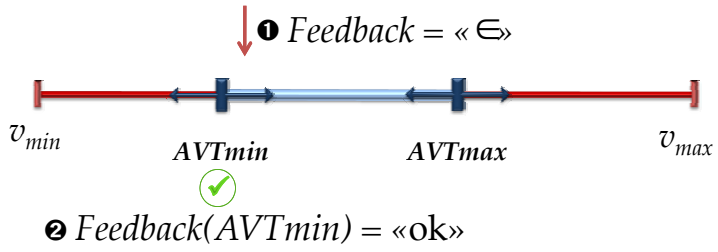

Figure 4.7 – Feedback de confirmation de l'appartenance d'une valeur

L'intégration ou l'exclusion d'une valeur de la plage de valeurs d'un AVRT n'est pas nécessairement instantanée : si la différence entre la valeur v_e et la valeur de la borne la plus proche est trop importante, un seul feedback envoyé à cette borne ne suffira pas à intégrer ou à exclure v_e à la plage de valeurs.

L'utilisation d'AVTs pour représenter les bornes de la plage de valeurs d'un AVRT présente deux intérêts :

– L'AVT a été conçu pour rechercher des valeurs susceptibles d'évoluer. C'est pourquoi, en implémentant les bornes de l'AVRT sous la forme d'un AVT, nous avons rendu notre AVRT capable de rechercher une plage de valeurs, même si les bornes de cette

plage de valeurs évoluent au cours du temps ;
- Le fait qu'une borne n'intègre pas directement une valeur extérieure à la plage de valeur, ou n'exclut pas directement une valeur intérieure à la plage de valeur, mais se limite à tendre dans la direction de cette valeur (pour l'intégrer ou l'exclure) permet une résistance de l'AVRT aux signaux incorrects. Par exemple, si un AVRT reçoit un exemple lui disant à tort d'intégrer une valeur fortement supérieure à la borne maximale de l'AVRT, celle-ci ne l'intégrera pas directement. Il se limitera à un simple ajustement, qu'il pourra le cas échéant corriger facilement.

En plus de son mécanisme de mise à jour, l'AVRT propose des mécanismes permettant à son environnement d'accéder à certaines informations sur sa plage de valeurs :
- test d'appartenance : il est possible d'envoyer une valeur v à un AVRT qui répond si cette valeur est incluse dans sa plage de valeurs ou pas ;
- test d'intégration : il est possible d'envoyer à un AVRT une valeur v n'appartenant pas à sa plage de valeur, et il répond s'il est possible de l'intégrer directement à sa plage de valeurs. Autrement dit, l'AVRT considère l'AVT le plus proche de v, et observe s'il est possible, par un unique feedback à cet AVT, de modifier sa valeur de façon à intégrer v à la plage de valeurs ;
- test d'exclusion : il est possible d'envoyer à un AVRT une valeur v appartenant à sa plage de valeur, et il répond s'il est possible de l'exclure directement de sa plage de valeurs. Autrement dit, l'AVRT considère l'AVT le plus proche de v, et observe s'il est possible, par un unique feedback à cet AVT, de modifier sa valeur de façon à exclure v à la plage de valeurs.

4.2 Mécanisme d'auto-ordonnancement entre agents

L'objectif de ce mécanisme est de permettre à deux agents d'être connectés par une relation d'ordre, l'un des agents étant considéré comme supérieur à l'autre.

4.2.1 Problématique de l'auto-ordonnancement d'agents

La notion d'ordonnancement est transitive : même si deux agents n'ont pas de lien d'ordonnancement direct, l'un des deux agents peut être supérieur à l'autre par transitivité avec ses liens d'ordonnancement le reliant à d'autres agents intermédiaires. Soit des agents A ordonnés, chaque agent $a \in A$ possédant une liste Sup_a d'agents qui lui sont supérieurs. La relation d'ordre entre deux agents a_i et a_j appartenant à A est alors définie par la formule 4.1.

$$\forall a_i, a_j \in A, a_i < a_j \Rightarrow \left| \begin{array}{l} a_j \in Sup_{a_i} \\ \exists a_n \in A \text{ tel que } a_n \in Sup_{a_i} \text{ et } a_n < a_j \end{array} \right. \tag{4.1}$$

Déterminer parmi les deux agents a_i et a_j lequel est supérieur à l'autre peut impliquer un traitement très important si les nombres d'agents et de relations d'ordre entre les agents sont importants. De plus, ce traitement peut être fortement perturbé si des incohérences sont

présentes (par exemple, un agent a_1 supérieur à un agent a_2, lui-même supérieur à un agent a_3, qui s'avère finalement être supérieur à l'agent a_1).

Nous avons donc travaillé à la conception d'un mécanisme permettant à deux agents d'assurer de façon locale la cohérence des relations d'ordre qui les relient. Pour éviter que la comparaison entre deux agents implique systématiquement un parcours des relations d'ordre de ces agents, nous cherchons à doter nos agents de la capacité de fournir une valeur numérique, appelé "niveau de l'agent", tel que si un agent a_i est inférieur à un autre agent a_j, alors le niveau $\eta(a_i)$ de l'agent a_i est inférieur au niveau $\eta(a_j)$ de l'agent a_j. Cette définition est représentée par la formule 4.2.

$$\forall a_i, a_j \in A, a_i < a_j \Rightarrow \eta(a_i) < \eta(a_j) \tag{4.2}$$

4.2.2 Description de l'agent *self-ordered*

Pour mettre en place ce mécanisme d'auto-ordonnancement entre agents, nous avons défini un agent générique capable de s'auto-ordonnancer : l'agent *self-ordered*. Notre objectif est que tout concepteur ayant besoin de concevoir des agents au sein d'un AMAS possédant des relations d'ordre entre-eux n'ait plus qu'à spécialiser l'agent *self-ordered* pour que son propre type d'agent soit capable de s'auto-ordonnancer.

Un agent *self-ordered* $a \in A$ possède une liste d'agents Sup_a qui lui sont supérieurs, et une liste d'agents Inf_a qui lui sont inférieurs, les notions de supériorité et d'infériorité entre agents étant basés sur l'équation 4.1 de la section 4.2.1. Son objectif est de fournir la valeur de son niveau $\eta(a)$, cette valeur devant respecter les contraintes établies par les formules 4.3 et 4.4.

$$\forall i \in Inf_a, \eta(i) < \eta(a) \tag{4.3}$$

$$\forall s \in Sup_a, \eta(a) < \eta(s) \tag{4.4}$$

Nous définissons la notion de degré de liberté $\delta(a_1, a_2)$ entre deux agents *self-ordered* a_1 et a_2 possédant une relation d'ordre entre eux, comme la différence entre les valeurs de leurs niveaux respectifs $\eta(a_1)$ et $\eta(a_2)$. Il s'agit donc de la valeur maximale avec laquelle l'agent inférieur peut incrémenter son niveau (respectivement, décrémenter pour l'agent supérieur) tout en respectant la relation d'ordre qui les lie entre eux. Le calcul du degré de liberté entre deux agents *self-ordered* a_1 et a_2 est représenté par la formule 4.5.

$$\delta(a_1, a_2) = |\eta(a_1) - \eta(a_2)| \tag{4.5}$$

En plus de son niveau $\eta(a)$, un agent *self-ordered* a se caractérise par son "degré de liberté supérieur" $\delta_{SUP}(a)$, représentant le degré de liberté entre lui et l'agent supérieur possédant le niveau le plus faible, et par son "degré de liberté inférieur" $\delta_{INF}(a)$ représentant le degré de liberté entre lui et l'agent inférieur possédant le plus haut niveau. Ces valeurs peuvent

donc être vues comme la "marge de manoeuvre" de l'agent a pour modifier la valeur de son niveau $\eta(a)$.

Un agent *self-ordered* connaît les niveaux $\eta(i)$ ainsi que les degrés de liberté supérieurs $\delta_{SUP}(i)$ de chaque agent i appartenant à sa liste d'agents supérieurs. Symétriquement, un agent *self-ordered* connaît les niveaux $\eta(s)$ ainsi que les degrés de liberté inférieurs $\delta_{INF}(s)$ de chaque agent s appartenant à sa liste d'agents inférieurs.

Dans l'exemple de la figure 4.8, l'agent a possède trois agents supérieurs et quatre agents inférieurs. Pour chaque agent supérieur s, l'agent a connaît non seulement le niveau $\eta(s)$ de s à partir duquel il déduit le degré de liberté $\delta(s,a)$ qui sépare s de lui, mais aussi le degré de liberté supérieur $\delta_{SUP}(s)$ de s. Symétriquement, pour chaque agent inférieur i, l'agent a connaît non seulement le niveau $\eta(i)$ de i à partir duquel il déduit le degré de liberté $\delta(i,a)$ qui sépare i de lui, mais aussi le degré de liberté inférieur $\delta_{INF}(i)$ de i. Il n'a cependant pas de vue ni sur les agents supérieurs à ses propres agents supérieurs, ni sur les agents inférieurs à ses propres agents inférieurs. Dans cet exemple, le degré inférieur $\delta_{INF}(a)$ est celui qui le sépare de l'agent inférieur le plus proche, soit l'agent i_2, tandis que le degré supérieur $\delta_{SUP}(a)$ est celui qui le sépare de l'agent supérieur le plus proche, soit l'agent s_1.

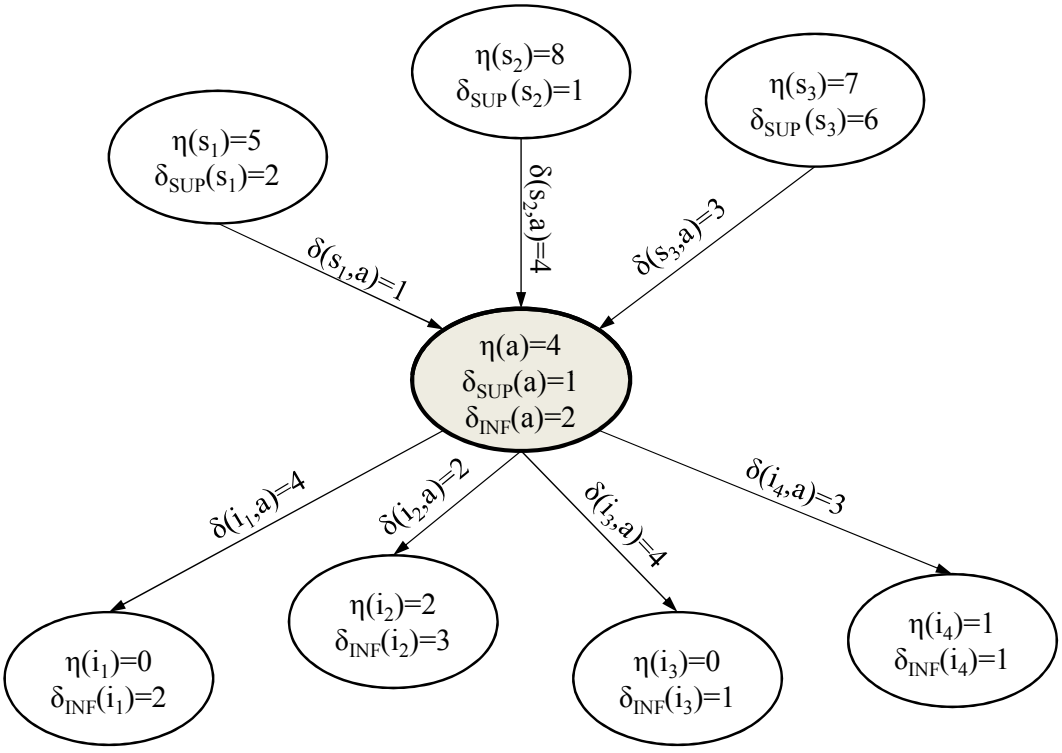

Figure 4.8 – Exemple de la connaissance d'un agent *self-ordered* sur ses agents supérieurs et inférieurs

L'objectif d'un agent *self-ordered* est de s'assurer de façon locale que la valeur de son niveau est supérieure au niveau de tous ses agents inférieurs, et inférieure aux niveaux de tous ses agents supérieurs. Il est capable d'ajouter des agents dans sa liste d'agents supérieurs ou inférieurs, et met alors à jour la valeur de son niveau. Il est aussi capable de prévenir ses agents supérieurs et inférieurs en cas de changement dans la valeur de son niveau.

La valeur initiale du niveau d'un agent *self-ordered* en l'absence d'agent supérieur ou inférieur est fixée par défaut à 0. Cependant, le choix de la valeur par défaut n'a pas d'importance dans le fonctionnement des agents *self-ordered* du moment que tous les nouveaux agents utilisent la même valeur par défaut en l'absence d'agent supérieur ou inférieur.

Enfin, nous avons détecté une SNC (Situation Non Coopérative) susceptible de se présenter en cas d'ajout d'un agent dans la liste des agents supérieurs ou inférieurs d'un agent *self-ordered*. Le détail de celle-ci fait l'objet du paragraphe suivant.

4.2.3 Résolution des incohérences entre les niveaux de deux agents *self-ordered*

Problématique Considérons un nouvel agent *self-ordered* a_{new} ajouté soit dans la liste des agents supérieurs Sup_a, soit dans la liste des agents inférieurs Inf_a de l'agent a. Afin de simplifier notre discours, nous fixons le cas d'un ajout de a_{new} dans la liste des agents supérieurs de a, le même raisonnement étant applicable de façon symétrique si a_{new} est ajouté dans la liste des agents inférieurs de a.

Les agents a et a_{new} possèdent chacun leur propre valeur de niveau $\eta(a)$ et $\eta(a_{new})$. Considérant la nouvelle relation d'ordre reliant les agents a et a_{new} qui stipule que $a < a_{new}$, il est nécessaire que la même relation d'ordre soit appliquée à leur niveau ; nous sommes donc supposés avoir $\eta(a) < \eta(a_{new})$. Une SNC de conflit apparaît donc si cette inégalité n'est pas respectée, autrement dit si $\eta(a) \geq \eta(a_{new})$.

Détection Chaque fois qu'un agent intègre un nouvel agent dans une de ses listes (agents supérieurs ou agents inférieurs), il compare le niveau de cet agent avec son propre niveau. Si cette comparaison ne correspond pas à la nouvelle relation d'ordre qui le lie à cet agent, il détecte une SNC.

Résolution Idéalement, un agent a recevant un agent a_{new} à intégrer dans la liste de ses agents supérieurs est capable de résoudre cette SNC par un simple mécanisme de tuning, en modifiant la valeur de son propre niveau. Cela est possible si a peut diminuer son niveau en dessous de celui de a_{new} sans entrer en contradiction avec un agent de sa liste d'agents inférieurs. Autrement dit, il faut que la différence entre $\eta(a)$ et $\eta(a_{new})$ soit inférieure au degré de liberté inférieur $\delta_{INF}(a)$ de l'agent a. Dans ce cas là, l'agent a affecte la valeur $\eta(a_{new}) - 1$ à son niveau $\eta(a)$.

Par exemple, si nous reprenons l'exemple de la figure 4.8 et que nous supposons que $\eta(a_{new}) = 4$. La condition d'infériorité de $\eta(a)$ par rapport à $\eta(a_{new})$ n'est pas respectée. En revanche, la différence entre $\eta(a)$ et $\eta(a_{new})$ est égale à 0, et est donc inférieure au degré de liberté $\delta_{INF}(a)$ de l'agent a, qui est égal à 2. Dans ce cas, l'agent a résout cette SNC en affectant son niveau $\eta(a)$ à la valeur $\eta(a_{new}) - 1$, soit la valeur 3.

Lorsque cette condition n'est pas respectée, l'agent a est incapable de résoudre lui-même la SNC qu'il vient de détecter. Dès lors, il en informe l'agent avec qui il est en conflit, afin d'entamer une résolution collective de cette SNC. Deux solutions sont alors possibles : soit l'agent a cherche à diminuer son niveau, soit l'agent a_{new} cherche à augmenter le sien. A

chaque cycle, chaque agent doit donc déterminer s'il est le plus apte à modifier sa valeur, ou s'il vaut mieux laisser l'autre agent modifier la sienne.

Criticité d'un agent *self-ordered* Si nous reprenons la définition de la coopération donnée au chapitre 3.1.2.2, qui la définit comme le juste milieu entre l'égoïsme et l'altruisme, nous pouvons considérer que dans le cas présent, si un même agent modifie systématiquement sa valeur, il fait preuve d'altruisme, tandis que si un agent laisse systématiquement les autres agents modifier la sienne, il fait preuve d'égoïsme. Pour établir quel est le comportement coopératif à adopter, nous reprenons la notion de criticité, définie en section 3.1.2.3 comme "*la notion de distance entre la situation courante et le but local de l'agent*".

Nous définissons la criticité d'un agent *self-ordered* comme son incapacité à modifier la valeur de son niveau. En tant qu'agents coopératifs, les agents *self-ordered* cherchent à diminuer la criticité de l'agent le plus critique. C'est donc toujours l'agent *self-ordered* le moins critique qui agit, afin d'aider les autres agents plus critiques.

Pour résoudre la SNC de conflit entre a et a_{new}, l'agent a veut diminuer son niveau en dessous de $\eta(a_{new})$; plus sa capacité à diminuer son niveau sans entrer en conflit avec d'autres agents est faible (autrement dit, son degré de liberté inférieur $\delta_{INF}(a)$ est faible), plus l'agent a est critique. De même, l'agent a_{new} veut augmenter son niveau au dessus de $\eta(a)$, c'est pourquoi plus son degré de liberté supérieur $\delta_{SUP}(a_{new})$ est faible, plus l'agent a_{new} est critique.

Chaque agent a connaissance du niveau des agents de son voisinage, ainsi que du degré de liberté inférieur de chacun de ses agents inférieurs et du degré de liberté supérieur de chacun de ses agents supérieurs (voir figure 4.8). Les agents a et a_{new} sont donc capables de détecter lequel d'entre eux est le moins critique. Dès lors qu'un agent a établi qu'il était le moins critique, il modifie alors sa valeur. S'il s'agit de a, alors il décrémente la valeur de son niveau, alors que s'il s'agit de a_{new}, il l'incrémente.

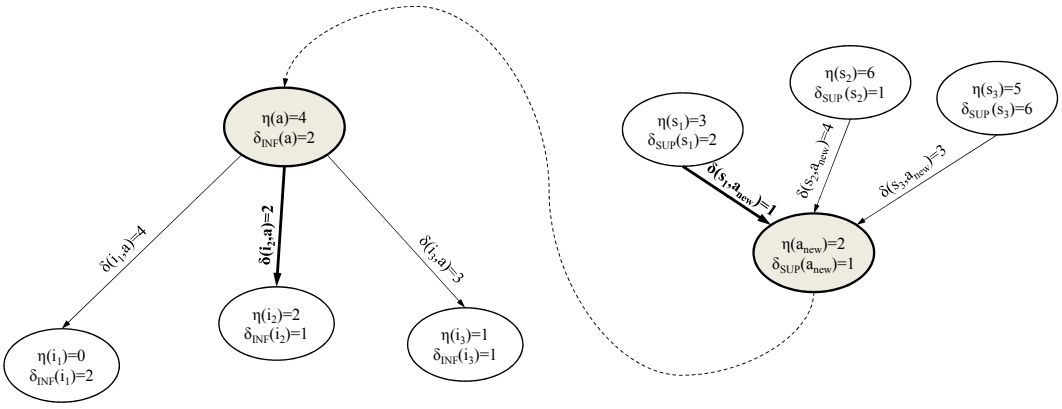

Figure 4.9 – Exemple de l'évaluation de l'agent le plus critique en comparant $\delta_{INF}(a)$ et $\delta_{SUP}(a_{new})$

Nous posons $MaxInf_a$ la liste des agents inférieurs de a possédant le niveau le plus élevé de Inf_a (autrement dit, ceux dont le niveau est égal à $\eta(a) - \delta_{INF}(a)$)), et $MinSup_{a_{new}}$ la liste des agents supérieurs de a_{new} possédant le niveau le plus faible de $Sup_{a_{new}}$ (autrement dit,

ceux dont le niveau est égal à $\eta(a_{new}) + \delta_{SUP}(a_{new})$)). Pour déterminer quel est l'agent le plus critique entre a et a_{new}, ces deux agents comparent leurs degrés de liberté respectifs $\delta_{INF}(a)$ et $\delta_{SUP}(a_{new})$, celui possédant le degré de liberté le plus faible étant le plus critique. Par exemple, sur la figure 4.9, comme le degré de liberté inférieur $\delta_{INF}(a)$ de a est de 2 à cause de l'agent i_2 et que le degré de liberté supérieur $\delta_{SUP}(a_{new})$ de a_{new} est de 1 à cause de l'agent s_1, c'est a_{new} qui est l'agent le plus critique.

En cas d'égalité, les deux agents se départagent en fonction du nombre d'agents qui limitent leur degrés de liberté respectifs, autrement dit en fonction de la taille de $MaxInf_a$ et de celle de $MinSup_{a_{new}}$; celui dont la liste possède le plus d'agents est l'agent le plus critique. Par exemple, sur la figure 4.10, l'agent a possède un degré de liberté inférieur $\delta_{INF}(a)$ égal à 2 à cause de deux agents inférieur i_2 et i_3, tandis que a_{new} possède un degré de liberté supérieur $\delta_{SUP}(a_{new})$ de 2 à cause d'un seul agent supérieur s_1. C'est donc l'agent a qui sera considéré comme le plus critique.

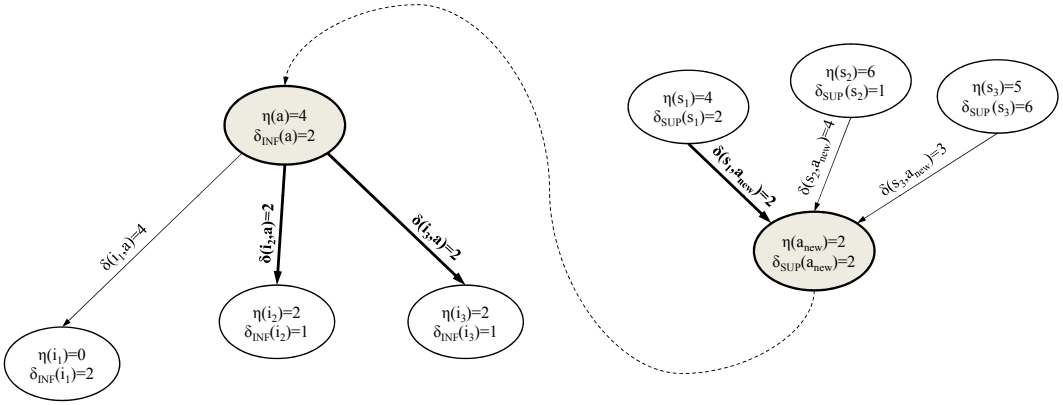

Figure 4.10 – Exemple de l'évaluation de l'agent le plus critique en comparant $Taille(MaxInf_a)$ et $Taille(MinSup_{a_{new}})$

Si $MaxInf_a$ et $MinSup_{a_{new}}$ sont de même taille, les agents a et a_{new} déterminent qui est le plus critique en fonction du degré de liberté des agents de $MaxInf_a$ et $MinSup_{a_{new}}$. Le degré de liberté inférieur $\min_{i \in MaxInf_a} \delta_{INF}(i)$ de l'agent de $MaxInf_a$ possédant le degré inférieur $\delta_{INF}(i)$ le plus faible est comparé au degré de liberté supérieur $\min_{s \in MinSup_{anew}} \delta_{SUP}(s)$ de l'agent de $MinSup_{a_{new}}$ possédant le degré supérieur $\delta_{SUP}(s)$ le plus faible, l'agent ayant obtenu la valeur la plus faible étant la plus critique. Par exemple, sur la figure 4.11, le degré inférieur $\min_{i \in MaxInf_a} \delta_{INF}(i)$ le plus faible parmi les agents de $MaxInf_a$ est à égalité celui de i_2 et i_3, et est égal à 1, tandis que le degré supérieur $\min_{s \in MinSup_{anew}} \delta_{SUP}(s)$ le plus faible parmi les agents de $MinSup_{a_{new}}$ est celui de s_1, et est égal à 2. C'est donc l'agent a qui est le plus critique.

Enfin, en cas de criticité égale, nous fixons arbitrairement que c'est l'agent qui a détecté la SNC (l'agent a) qui modifie son niveau (en l'incrémentant ou le décrémentant). Le traitement réalisé pour déterminer l'agent le plus critique entre deux agents *self-ordered* a et a_{new} est représenté par l'algorithme 4.1 (cet algorithme tient aussi compte de l'hypothèse formulée précédemment, stipulant que a_{new} est un agent supérieur à l'agent a ; un traitement symétrique est appliqué dans le cas où a_{new} est un agent inférieur).

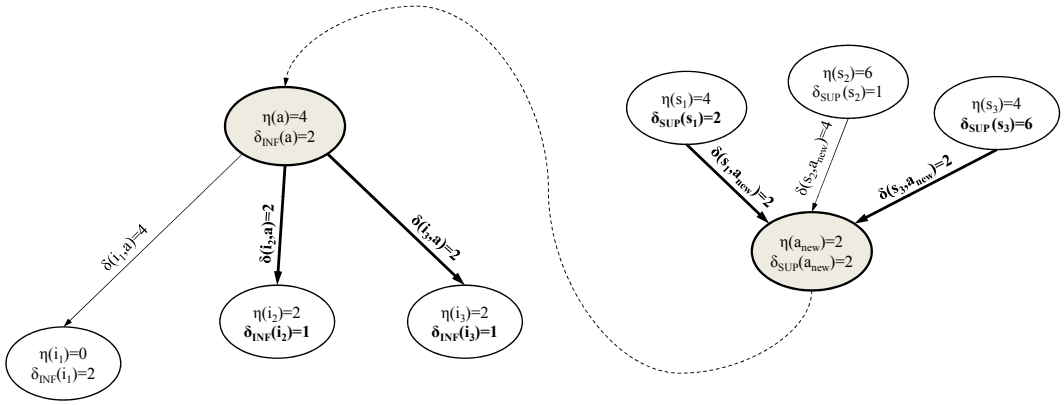

Figure 4.11 – Exemple de l'évaluation de l'agent le plus critique en comparant $\max_{i \in Inf_a} \delta_{INF}(i)$ et $\min_{s \in Sup_{anew}} \delta_{SUP}(s)$

Propagation des SNCs de conflit Lorsque l'agent *self-ordered* le moins critique ne peut faire évoluer davantage son niveau (son degré de liberté est nul), il n'a pas d'autre choix, pour augmenter son degré de liberté, que de renoncer à respecter son ordre de grandeur avec l'agent *self-ordered* limitant son degré de liberté. Il peut alors continuer son traitement de résolution de la SNC courante, mais la modification de son niveau entraîne l'apparition d'une autre SNC de conflit avec l'agent *self-ordered* vis-à-vis duquel il ne peut rien faire. Cependant, les agents a et a_{new} poursuivent leur processus jusqu'à ce que le niveau de a soit au dessus de celui de a_{new}. La SNC de conflit entre eux est alors résolue, en minimisant l'apparition d'autres SNCs de conflit.

Pour résoudre les SNCs découlant de la résolution d'une première SNC, le processus réalisé est le même, à une différence près. Reprenons l'exemple du conflit entre a et a_{new}, et supposons que la résolution de cette SNC amène l'agent a à diminuer son niveau jusqu'à entrer en conflit avec un agent a_{inf}. Cet agent sait que si l'agent a l'a mené dans cette SNC, c'est qu'il était trop contraint pour éviter cette SNC. L'agent a_{inf} va donc considérer que c'est nécessairement a qui est l'agent le plus critique. A partir de là, il réalise le même traitement que celui expliqué précédemment, en diminuant sa valeur jusqu'à résolution de cette SNC.

La résolution d'une SNC de conflit entre deux agents a et a_{new} peut donc provoquer d'autres conflits, dont leurs propres résolutions peuvent entraîner d'autres conflits, etc. Les perturbations se propagent donc de proche en proche jusqu'à ce que tous les conflits aient disparus.

Cependant, un dernier cas peut encore apparaître. Supposons que la résolution du conflit entre a et a_{new} mène par propagation à ce qu'un agent a_{sup} supérieur à l'agent a (directement ou par transitivité) entre en conflit avec un des agents inférieurs a_{inf} à l'agent a. L'agent a_{inf} est alors dans l'incapacité d'augmenter son niveau, tout comme a_{sup} est dans l'incapacité de diminuer le sien. Nous sommes alors dans un cas de résolution de problème surcontraint : l'agent a_{inf} ne peut pas être simultanément inférieur à a et supérieur à a_{sup}. Dans ce cas, nous considérons que la relation d'ordre la plus ancienne doit disparaître, au profit de la plus récente. Un agent (ici l'agent a_{inf}) détectant qu'il est dans l'incapacité de satisfaire simultanément deux relations d'ordre décide donc de supprimer la relation d'ordre la plus

Algorithme 4.1 : Évaluation de l'agent *self_ordered* le plus critique entre a et a_{new}

Si $\delta_{INF}(a) < \delta_{SUP}(a_{new})$ **Alors**
  Retour(a)
Sinon Si $\delta_{INF}(a) > \delta_{SUP}(a_{new})$ **Alors**
  Retour(a_{new})
Sinon
  $MaxInf_a = \{\}$
  $\eta_{max} = (\max_{i \in Inf_a} \delta_{INF}(i))$, le degré inférieur maximal des agents de Inf_a
  Pour tout agent $i \in Inf_a$, la liste des agents inférieurs de a **Faire**
    Si $\eta(i) = \eta_{max}$ **Alors**
      $MaxInf_a$.Ajouter(i)
    Fin Si
  Fin Pour
  $MinSup_{a_{new}} = \{\}$
  $\eta_{min} = (\min_{s \in Sup_{anew}} \delta_{SUP}(s))$, le degré supérieur minimal des agents de $Sup_{a_{new}}$
  Pour tout agent $s \in Sup_{a_{new}}$, la liste des agents supérieurs de a_{new} **Faire**
    Si $\eta(s) = \eta_{min}$ **Alors**
      $MinSup_{a_{new}}$.Ajouter(s)
    Fin Si
  Fin Pour
  Si $Taille(MaxInf_a) < Taille(MinSup_{a_{new}})$ **Alors**
    Retour(a)
  Sinon Si $Taille(MaxInf_a) > Taille(MinSup_{a_{new}})$ **Alors**
    Retour(a_{new})
  Sinon Si $\eta_{max} < \eta_{min}$ **Alors**
    Retour(a)
  Sinon Si $\eta_{max} > \eta_{min}$ **Alors**
    Retour(a_{new})
  Sinon
    Retour(a) par défaut
  Fin Si
Fin Si

ancienne.

4.3 Mécanisme de filtrage de variables inutiles

Dans un système multi-agent, il est fréquent d'être en présence d'un "agent *variable*", qui représente l'état d'une variable, et qui a pour objectif de fournir les données de cette variable à d'autres "agents *récepteur*". Ces derniers exploitent alors ces données pour déterminer quelle action réaliser. Dans le cadre d'*Amadeus*, par exemple, les agents *contexte* sont des agents *récepteur* qui exploitent les données transmises par les agents *variable* que sont les

agents *donnée*, dans le but de proposer des actions à l'agent *contrôleur*.

En fonction du domaine d'application, certaines variables reçues par les agents *récepteur* peuvent leur être inutiles. Nous définissons une variable inutile comme une variable dont la valeur n'a pas d'impact sur la capacité des agents *récepteur* à atteindre leur objectif. Dans certains cas, ces variables inutiles vont jusqu'à dégrader la capacité des agents *récepteur* à atteindre leur objectif. Dans *Amadeus*, par exemple, des agents *contexte* peuvent se trouver invalides à cause de variables inutiles.

Dans certains cas spécifiques, la détection de variables inutiles est possible, mais dans d'autres cas, comme pour *Amadeus*, déterminer si une variable est utile ou inutile est impossible. Le mieux que les agents puissent faire est de détecter certaines situations où une variable a semblé utile. Nous avons donc mis au point un mécanisme utilisable par un agent *variable* en charge de la transmission de la valeur d'une variable (l'agent *donnée* dans *Amadeus*, par exemple) afin de déterminer si cette variable est utile aux autres agents ou pas.

4.3.1 Présentation générale

Nous considérons un système multi-agent comme composé d'agents *variable*, d'agents *récepteur*, et éventuellement d'autres types d'agents. Un agent *variable* possède la connaissance de l'état de sa variable, et possède la capacité de transmettre cette donnée à un ensemble d'agents *récepteur*. L'agent *récepteur* possède la capacité de réaliser certaines actions en fonction des données envoyées par les agents *variable*. Il s'agit là de deux types d'agents génériques, à spécialiser en fonction du problème à résoudre.

Le mécanisme de filtrage de variables inutiles que nous proposons ici se compose de deux parties :

- La première partie se situe au niveau des agents *récepteur*. Le concepteur de l'agent *récepteur* doit établir, en fonction du domaine d'application, dans quelle situation l'agent *récepteur* est capable de dire qu'une variable semble avoir été utile, et vis-à-vis de quelle action à réaliser. Lorsqu'un agent *récepteur* se retrouve dans une telle situation, il envoie un signal d'utilité à l'agent *variable* qui lui a été utile, en précisant l'action pour laquelle la variable a été utile. Par exemple, dans *Amadeus*, ce signal est généré par les agents *contexte* lorsqu'une donnée empêche un agent *contexte* d'être valide dans une situation où il semble que sa validité l'aurait mené à réaliser une mauvaise action.
- La seconde partie se situe au niveau de l'agent *variable*, et concerne la capacité d'un tel agent à traiter les signaux d'utilité reçus, pour évaluer l'utilité de sa donnée vis-à-vis de la réalisation d'une action par les agents *récepteur*.

L'instanciation d'un agent *récepteur* est donc très spécifique au domaine d'application, afin de savoir quand générer les signaux d'utilité. En revanche, le mécanisme intégré à l'agent *variable* est générique.

4.3.2 Propriétés des variables utiles et inutiles

Lorsqu'un agent *variable* reçoit un signal d'utilité envoyé par un agent *récepteur*, cela signifie que l'agent *récepteur* a jugé que la valeur de l'agent *variable* lui a été utile pour décider de réaliser une action a. L'objectif de l'agent *variable* est alors de déterminer si sa valeur a

réellement été utilisée dans la prise de décision de l'agent *récepteur* vis-à-vis de l'action a (l'agent *variable* est donc utile), ou si l'agent *récepteur* aurait pris la même décision concernant l'action a même sans connaître cette valeur (l'agent *variable* est donc inutile).

Considérons un agent *variable* V, ainsi que l'ensemble D des données générées par cet agent *variable*. Nous considérons aussi l'échantillon $\mathcal{E}ch_a = \{d \in D \ / \ a\}$ des données de l'agent *variable* V lors de la réception de signaux d'utilité concernant l'action a, et $\mathcal{E}ch_{\bar{a}} = \{d \in D \ / \ a\}$ l'échantillon des données de l'agent *variable* V lors de la réception de signaux d'utilité concernant n'importe quelle action autre que a. Pour toute action a, l'agent *variable* V possède donc deux échantillons $\mathcal{E}ch_a$ et $\mathcal{E}ch_{\bar{a}}$ qui représentent l'état de sa variable à la réception des signaux d'utilité selon que ces signaux portaient sur l'action a, ou pas.

La valeur de la variable de l'agent V à la réception d'un signal d'utilité concernant l'action a suit une fonction de densité théorique $d_a(V)$, tandis que la valeur de cette même variable à la réception d'un signal d'utilité concernant une autre action que a suit une fonction de densité théorique $d_{\bar{a}}(V)$. Ces fonctions de densité représentent donc la distribution théorique des valeurs appartenant aux échantillons $\mathcal{E}ch_a$ et $\mathcal{E}ch_{\bar{a}}$.

A présent, nous considérons deux agents *variable* U et I, U étant associé à une variable utile et I étant associé à une variable inutile vis-à-vis de la réalisation d'une action a. Le principal souci dans la détection de l'inutilité de la variable I réside dans le fait que, parfois, sa valeur peut sembler utile à un agent *récepteur*. Les agents U et I sont donc tous les deux susceptibles de recevoir des signaux d'utilité de la part d'agents *récepteur*.

Cependant, lorsque la variable de l'agent I est jugée utile, ce n'est qu'un *hasard* ; la réception d'un signal d'utilité par l'agent I est donc indépendante de la valeur de la variable de I. Quelle que soit l'action a associée au signal d'utilité reçu, les probabilités concernant la valeur de la variable de I sont les mêmes. C'est pourquoi la valeur de la variable de l'agent I à la réception d'un signal d'utilité suit la même fonction de densité théorique quelle que soit l'action associée au signal d'utilité. Autrement dit, si nous supposons l'action a, nous pouvons considérer l'égalité représentée par la formule 4.6.

$$d_a(I) = d_{\bar{a}}(I) \tag{4.6}$$

En revanche, la valeur de la variable représentée par l'agent U étant utile, il y a une corrélation entre sa valeur et la génération des signaux d'utilité. Ainsi, si la variable de l'agent U est utile à la réalisation de l'action a, il en résulte une différence dans les fonctions de densité que suit la valeur de cette variable à la réception d'un signal d'utilité selon que ce signal porte sur l'action a ou pas. Cette différence est représentée par la formule 4.7.

$$d_a(U) \neq d_{\bar{a}}(U) \tag{4.7}$$

Les formules 4.6 et 4.7 nous permettent d'établir une différence entre les agents *variable* utiles et inutiles vis-à-vis de toute action a. Nous présentons au paragraphe suivant comment chaque agent *variable* exploitent cette différence pour réaliser une évaluation de leur utilité.

4.3.3 Évaluation de l'utilité des variables

Les formules du paragraphe précédent portent sur les fonctions de densité théorique que suivent les valeurs des différents agents *variable* à la réception d'un signal d'utilité en fonction de l'action associée à ce signal. Ne disposant pas de ces fonctions de densité théorique, chaque agent *variable* V va donc construire une fonction de densité empirique $\hat{d}_a(V)$ à partir de l'échantillon $\mathcal{E}ch_a$ afin d'estimer la fonction de densité théorique $d_a(V)$. De même, chaque agent *variable* V construit une fonction de densité empirique $\hat{d}_{\bar{a}}(V)$ à partir de l'échantillon $\mathcal{E}ch_{\bar{a}}$ pour estimer la fonction de densité théorique $d_{\bar{a}}(U)$.

La construction des fonctions de densité empiriques $\hat{d}_a(V)$ et $\hat{d}_{\bar{a}}(V)$ se fait en continu, sans requérir d'enregistrer tous les échantillons $\mathcal{E}ch_a$ et $\mathcal{E}ch_{\bar{a}}$. Chaque fois qu'un agent *variable* V reçoit un signal d'utilité associé à une action a, il met à jour la fonction de densité empirique $\hat{d}_a(V)$ à partir de la valeur courante de la variable V. De même, lorsqu'il reçoit un signal d'utilité associé à une action différente de a, il met à jour la fonction de densité empirique $\hat{d}_{\bar{a}}(V)$ à partir de la valeur courante de la variable V.

Au fur et à mesure que des signaux d'utilité sont reçus par l'agent *variable*, les différentes fonctions de densité empiriques se rapprochent des fonctions de densité théoriques. Ainsi, les égalités représentées par les formules 4.6 et 4.7 peuvent être observées empiriquement. La distinction entre une donnée inutile I et une donnée utile U peut alors être observée en comparant d'une part les fonctions de densité $\hat{d}_a(I)$ et $\hat{d}_{\bar{a}}(I)$,et d'autre part les fonctions de densité et $\hat{d}_a(U)$ et $\hat{d}_{\bar{a}}(U)$. Pour I, les fonctions $\hat{d}_a(I)$ et $\hat{d}_{\bar{a}}(I)$ seront similaires, tandis que pour U, une différence entre les fonctions $\hat{d}_a(U)$ et $\hat{d}_{\bar{a}}(U)$ pourra être observée.

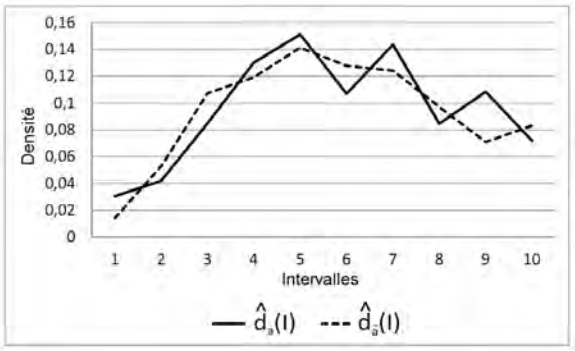

Figure 4.12 – Illustration de la similarité entre les fonctions de densité empiriques $\hat{d}_a(I)$ et $\hat{d}_{\bar{a}}(I)$ d'une variable inutile

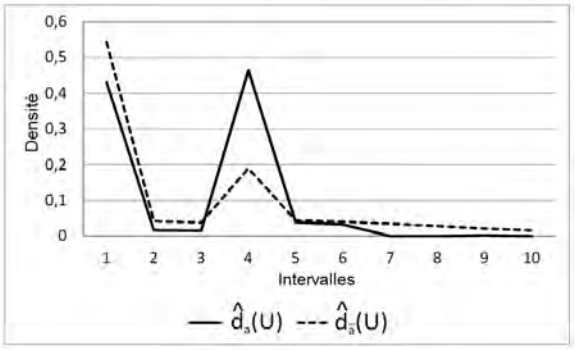

Figure 4.13 – Illustration de la différence entre les fonctions de densité empiriques $\hat{d}_a(U)$ et $\hat{d}_{\bar{a}}(U)$ d'une variable utile

Pour tout agent *variable* V, la comparaison entre deux fonctions de densité $\hat{d}_a(V)$ et $\hat{d}_{\bar{a}}(V)$ renvoie une valeur représentant la distance entre ces deux fonctions, et est réalisée à travers le calcul de la distance du *chi-square* [Greenwood et Nikulin, 1996] :

$$\delta\left(\hat{d}_a(V), \hat{d}_{\bar{a}}(V)\right) = \sum_{Valeur\ i} \frac{(\hat{d}_a(V)(i) - \hat{d}_{\bar{a}}(V)(i))^2}{\hat{d}_a(V)(i)}$$

où $\hat{d}_a(V)(i)$ représente la fréquence générale de la valeur i pour la variable V lorsque

le signal d'utilité reçu est associé à l'action a, et $\hat{d}_{\overline{a}}(V)(i)$ la fréquence de la valeur i pour la variable V lorsque le signal d'utilité reçu n'est pas associé à l'action a.

L'utilisation de la distance du *chi-square* pour comparer deux fonctions de densité permet d'obtenir une valeur statistiquement significative. En effet, si nous supposons qu'une variable est inutile, alors nous pouvons dire que $\delta\left(\hat{d}_{\overline{a}}(V), \hat{d}_a(V)\right)$ suit une loi de *chi-square* [Baillargeon, 1982], ce qui signifie que la loi suivie par la fonction de densité $\hat{d}_{\overline{a}}(V))$ est la même que celle suivie par la fonction de densité $\hat{d}_a(V))$.

Quand un agent *variable* reçoit un *signal d'utilité* associé à une action a, il met à jour ses fonctions de densité vis-à-vis cette action. Ainsi, l'évaluation de l'utilité de cette variable pour chaque action réalisable est de plus en plus précise. Finalement, quand le niveau d'utilité d'un agent *variable* vis-à-vis d'une action a passe en dessous d'un seuil τ préalablement fixé, l'agent *variable* considère sa donnée inutile pour les agents voulant réaliser l'action a.

Concernant la valeur de τ, une valeur trop élevée rend plus difficile le filtrage des variables inutiles, tandis qu'une valeur trop faible augmente le risque de filtrer à tort des variables utiles. Des tests empiriques lors de l'application de ce mécanisme de filtrage dans *Amadeus* nous a permis de fixer la valeur de τ à 0.5 pour notre système. Cependant, la principale perspective portant sur la conception de ce mécanisme de filtrage des variables inutiles sera de recherche comment s'abstraire d'un seuil fixé, en rendant le paramètre τ adaptatif.

Troisième partie

Évaluation

5 Évaluation

NOUS nous intéressons dans ce chapitre à l'évaluation du système *Amadeus*. Plus précisément, nous nous intéressons à la capacité d'*Amadeus* à observer l'activité des utilisateurs de sorte à réaliser à leur place leurs actions régulières sans connaissance *a priori*.

Pour évaluer un système destiné à un environnement ambiant, trois approches sont possibles. La première consiste à appliquer ce système à un environnement ambiant existant. Cependant, comme l'explique [Shafti *et al.*, 2012], "*while real-spaces in laboratories are good enough to analyze the performance of many applications and perform some basic end-user studies, obtaining large amount of real-world data needs a real space with people living for a long time, requiring large amounts of investment and time*" [1]. Une telle solution sera donc appropriée lorsque nous obtiendrons une version "finale" de notre système. Mais à court terme, il est nécessaire de disposer d'une solution alternative pour réaliser des évaluations rapides des différentes versions produites par la conception incrémentale de notre système.

Une autre approche consiste à utiliser des données préalablement enregistrées (*data sets*). Plusieurs études ont déjà été menées dans des environnements ambiants existants (par exemple, le projet MavHome de [Cook *et al.*, 2003]), et ont mené à l'enregistrement d'un grand nombre de jeux de données. Pour évaluer certains algorithmes, réutiliser ces jeux de données existants est une solution tout à fait satisfaisante. Cependant, si l'application d'un algorithme *hors ligne* sur un jeu de données pré-enregistré est tout à fait possible, il n'en est pas de même avec un algorithme *en ligne* tel que celui utilisé par *Amadeus*. En effet, *Amadeus* a besoin d'interagir avec le système ambiant pour observer les effets de ses actions sur celui-ci, ce qui n'est pas applicable avec des données enregistrées.

Finalement, nous avons choisi une troisième méthode, qui consiste à appliquer *Amadeus* à un simulateur de système ambiant. Un tel outil doit alors permettre de simuler un environnement virtuel et d'établir un comportement pour des utilisateurs virtuels, puis de simuler les actions de ces utilisateurs dans leur environnement virtuel. Les trois avantages de cette approche sont i) de ne pas nécessiter d'environnement et d'utilisateurs réels, ii) de pouvoir simuler des évolutions de l'environnement sous l'effet d'actions des utilisateurs ou du système, et iii) de s'affranchir de la contrainte du temps en "accélérant" les scénarios de tests.

1. Notre traduction : "si les espaces réels en laboratoire sont suffisamment satisfaisants pour analyser les performances de nombreuses applications et réaliser des études basiques concernant les utilisateurs finaux, obtenir une grande quantité de données réelles nécessite un environnement réel avec des personnes y vivant sur une longue durée, et exige des investissements importants et du temps"

Dans ce chapitre, nous commençons par présenter le simulateur que nous avons conçu. Nous présentons ensuite les différentes évaluations réalisées sur *Amadeus* à l'aide de ce simulateur. Chacune de ces évaluations a pour objectif de montrer une propriété particulière de notre système.

Dans un premier temps, nous évaluons la capacité d'*Amadeus* à apprendre un comportement proactif à appliquer à un unique dispositif dans des conditions optimales. Nous étendons ensuite cette simulation à un cas avec un second dispositif. Nous étudions alors l'indépendance des apprentissages réalisés localement au niveau de chacun de ces deux dispositifs. Autrement dit, nous montrons comment chacun des dispositifs est capable d'apprendre quelle action réaliser dans les différentes situations rencontrées sans nécessiter de concertation entre eux.

Les deux études suivantes visent à montrer la capacité d'ouverture d'*Amadeus*. Pour cela, nous observons l'impact qu'a l'ajout d'un AMAS *dispositif* associé à un dispositif sur un autre AMAS *dispositif* déjà en fonctionnement, ainsi que l'effet sur le nouvel AMAS *dispositif* de la présence d'un autre AMAS *dispositif* instancié avant lui. Nous étudions ensuite l'impact de la suppression d'un AMAS *dispositif* sur un autre AMAS *dispositif* en cours de fonctionnement.

Ensuite, nous évaluons l'aptitude d'*Amadeus* à agir dans un environnement dynamique, avec un utilisateur dont les préférences (et par conséquent, le comportement) changent en cours de simulation. Nous étudions d'abord cette aptitude en n'effectuant qu'une légère modification des préférences de l'utilisateur. Puis nous réitérons cette expérience en provoquant cette fois un changement radical dans ses préférences, modifiant alors complètement son comportement.

Enfin, nous étudions comment *Amadeus* apprend malgré la présence de variables inutiles à son processus. Pour cela, nous évaluons l'impact de ces variables sur le processus d'apprentissage d'un AMAS *dispositif*, puis nous évaluons le mécanisme de filtrage des variables inutiles sur un cas d'étude simple, puis sur un second cas d'étude plus complet.

Pour chacune de ces évaluations, nous suivrons un plan similaire. Dans un premier temps, nous précisons l'objectif de l'évaluation. Nous présentons ensuite le cadre dans lequel est réalisée l'évaluation, autrement dit la configuration de la simulation utilisée. Nous présentons enfin les résultats obtenus avant de conclure l'étude par une discussion.

5.1 Présentation du simulateur de système ambiant

Notre simulateur de système ambiant permet de concevoir un environnement virtuel dans lequel le concepteur peut rajouter des dispositifs. Une illustration de ce simulateur est visible en section 5.6.3 (voir figure 5.40).

5.1.1 Description du système ambiant virtuel

Actuellement, le simulateur permet d'établir la topologie de l'environnement (nombre de pièces, position des portes et des fenêtres, etc.), ainsi que d'ajouter :

- des capteurs :
  - capteurs de luminosité ;
  - capteurs de température ;
  - capteurs de présence ;
  - capteurs d'état (ouvert ou fermé) des volets, des portes ou des fenêtres.
- des effecteurs :
  - des lampes (allumées/éteintes) ;
  - des chauffages (à puissance forte ou moyenne, ou éteints).

Le simulateur gère les échanges thermiques et lumineux à l'intérieur du monde virtuel en fonction du temps et de l'état des différents dispositifs. Le temps lui-même évolue par cycle de 5 secondes. Au cours d'une journée simulée, la température et la luminosité extérieures (qui influencent les luminosités et les températures des pièces de la simulation) évoluent progressivement entre une valeur minimale et une valeur maximale, ces valeurs étant générées aléatoirement à chaque simulation.

5.1.2 Description des utilisateurs virtuels

Enfin, nous avons ensuite intégré des utilisateurs virtuels à notre simulation. L'objectif de ce travail a été d'établir le plus simplement possible un comportement pour des utilisateurs virtuels. Pour nos études, ce comportement doit être régulier, mais non basé sur un scénario pré-établi. Les utilisateurs virtuels doivent donc être dotés d'un comportement général décrivant leurs activités, associé à des préférences sur leur environnement. Ces préférences doivent être accompagnées de la liste des actions à réaliser pour les satisfaire. Le comportement d'un utilisateur est donc séparé en deux niveaux : les règles de préférence et les règles de comportement, les règles de préférence étant appliquées en priorité par rapport aux règles de comportement. Chaque règle, de comportement ou de préférence, est elle-même séparée en une liste de conditions et d'actions.

Les règles de préférence sont séparées en trois catégories : les règles de préférence générales, les règles de préférence d'entrée (dans la pièce) et les règles de préférence de sortie. Lorsque la simulation débute, le simulateur regarde d'abord les conditions des règles de préférence générales de chaque utilisateur, et conserve alors les règles pour lesquelles les conditions sont respectées. Il va alors réaliser en priorité les actions associées à ces préférences. Par exemple, si un utilisateur préfère avoir la luminosité au dessus d'un certain seuil, alors cette préférence aura pour condition que la luminosité de la pièce courante est en dessous de ce seuil, et aura pour action, par exemple, le fait d'allumer la lumière afin d'augmenter la luminosité courante au dessus de ce seuil.

Lors de l'évaluation des conditions d'une règle de préférence, une condition supplémentaire est automatiquement générée afin de représenter la possibilité de réaliser une action. Par exemple, si l'action à réaliser consiste à allumer une lampe dans la pièce courante, alors la condition "il y a au moins une lampe allumable dans la pièce courante" est automatiquement ajoutée aux conditions à respecter. Les différentes conditions peuvent porter sur des données spécifiques (la luminosité de la cuisine, l'état de la lampe du salon, etc.) ou sur des données relatives (la luminosité de la pièce courante, l'état de la totalité ou d'au moins une lampe de la pièce courante, etc.).

Les règles de préférence en entrée et en sortie sont similaires aux règles de préférence générale, mais ne sont appliquées qu'au moment où l'utilisateur vient d'entrer dans une pièce (règles de préférence en entrée), ou lorsqu'il s'apprête à sortir d'une pièce (règles de préférence en sortie). Par exemple, la préférence d'un utilisateur de toujours fermer la porte qu'il vient de passer est une règle de préférence en entrée, tandis que sa préférence d'éteindre systématiquement les lampes en sortant d'une pièce dans laquelle il était le dernier est une préférence en sortie.

L'ensemble des règles de comportement et de préférence est décrit par un fichier XML. Il est aussi possible de paramétrer à nouveau les règles des utilisateurs avec un autre fichier XML en cours de simulation, pour modifier les comportements et préférences des utilisateurs. Pour les différentes études qui suivront, nous considérons que ces préférences sont cohérentes, c'est-à-dire que l'utilisateur ne se contredit pas lui-même.

5.1.3 Fonctionnement du simulateur

Chaque fois qu'un utilisateur virtuel termine une action, le simulateur vérifie ses règles de préférence générales. Dès lors qu'elles sont toutes respectées et que l'utilisateur n'a plus d'actions à réaliser, le simulateur s'intéresse aux règles de comportement de cet utilisateur. Ces règles sont aussi composées de conditions à respecter pour être valides, et d'actions à réaliser le cas échéant. De plus, elles possèdent un niveau de priorité et un coefficient de pondération. Si plusieurs règles de comportement sont valides au même moment, c'est la règle possédant la plus haute priorité qui est sélectionnée. S'il existe plusieurs règles possédant le même plus haut niveau de priorité, l'utilisateur choisit alors une de ces règles au hasard en pondérant les probabilités de chacune de ces règles par leur coefficient de pondération respectif. Par exemple, si un utilisateur possède deux règles de comportement de même priorité, l'une l'incitant à se déplacer dans une position au hasard de la pièce courante avec un poids de 3, et l'autre l'incitant à se déplacer dans une position d'une autre pièce avec un poids de 1, alors l'utilisateur a trois fois plus de chance de choisir la première règle que la seconde.

Lorsque l'utilisateur entre dans une pièce ou s'apprête à en sortir, même s'il est en train de réaliser une action liée à une règle de comportement, il interrompt son action courante pour vérifier ses règles de préférence, réalise les actions associées le cas échéant, puis reprend son action courante.

Le simulateur est connecté à *Amadeus*. A la fin de chaque cycle du simulateur, les mises à jour de chaque donnée sont envoyées à l'agent *capteur* (ou à l'agent *effecteur*) de l'AMAS *dispositif* associé de la même façon que si cet AMAS *dispositif* avait été connecté au vrai dispositif au travers d'un gestionnaire de contexte. Les différents AMAS *dispositif* réalisent leurs cycles en parallèle, puis les ordres d'actions des agents *contrôleur* sont transmis par l'agent *effecteur* jusqu'au simulateur. Celui-ci réalise alors un nouveau cycle, et ainsi de suite. Il est donc possible de faire fonctionner le simulateur et *Amadeus* en vitesse accélérée. Par exemple, pour des simulations de 50 jours, chaque simulation réalisant un cycle toutes les 5 secondes simulées (soit 432 000 cycles pour chaque simulation), le traitement total prenait entre 30 minutes pour les simulations les plus basiques, et au maximum 10h pour une simulation bien plus importante.

Enfin, la génération des déplacements d'un utilisateur est réalisée à partir d'un générateur de nombres pseudo-aléatoires (fourni par la classe Random, en Java) qui, étant donnée une valeur initiale appelée *seed*, génère une suite de valeurs aléatoires. Une des propriétés de ce générateur est que, bien que ces valeurs soient aléatoires, l'utilisation d'une *seed* identique permet de regénérer exactement la même suite de valeurs aléatoires. Ainsi, il est possible de relancer plusieurs fois la même simulation en utilisant la même valeur pour la *seed*.

5.2 Évaluation des capacités d'apprentissage d'*Amadeus*

Dans cette section, nous présentons différentes évaluations réalisées sur *Amadeus* à l'aide de notre simulateur. Ces évaluations ont pour objectif de montrer sa capacité à apprendre un comportement proactif à appliquer aux dispositifs d'un système ambiant à partir de l'observation des actions de l'utilisateur.

5.2.1 Apprentissage du comportement pour un dispositif

Pour cette première étude, nous évaluons les capacités d'apprentissage d'*Amadeus* dans des conditions optimales : la totalité des informations perçues sont utiles pour son apprentissage, l'environnement ne change pas au cours du temps (pas d'ajout ni de suppression d'un dispositif en cours de fonctionnement) et il n'y a qu'un utilisateur présent dans cet environnement et qu'un seul effecteur à contrôler.

L'objectif de cette évaluation est de montrer la capacité d'*Amadeus* à anticiper les actions que l'utilisateur s'apprête à réaliser sur un dispositif. Les phases d'apprentissage et d'action ne sont pas dissociées : notre système agit en continu au fur et à mesure de son apprentissage. En attribuant un comportement régulier à notre utilisateur virtuel, c'est-à-dire un comportement basé sur des préférences qui resteront les mêmes au cours de toute la simulation, nous nous attendons à ce qu'*Amadeus* tende progressivement à agir à la place de l'utilisateur.

Cadre d'étude La simulation utilisée pour cette étude modélise un appartement composé de plusieurs pièces. Dans l'une d'elles, nous ajoutons une lampe, à laquelle nous associons un AMAS *dispositif*. Nous ajoutons dans cette même pièce un capteur de luminosité et un capteur de présence, ainsi qu'un capteur d'état pour chacune des deux portes permettant d'accéder à la pièce afin de savoir si elles sont ouvertes ou fermées (permettant donc de savoir lorsqu'un utilisateur passe au niveau de chaque porte). Un AMAS *dispositif* est également associé à chaque capteur, bien que chacun d'eux ne soit constitué que d'un agent *capteur*.

Au final, en plus de l'état courant de la lampe, l'AMAS *dispositif* associé à la lampe perçoit un certain nombre de données depuis les autres AMAS *dispositif* :

- La valeur envoyée par le capteur de luminosité situé dans la même pièce que la lampe ;
- La valeur envoyée par le capteur de présence situé dans la même pièce que la lampe ;
- Les valeurs envoyées par les capteurs placés sur les deux portes de la même pièce que la lampe.

Nous ajoutons à cette simulation un unique utilisateur doté d'un comportement très simple : il se promène aléatoirement dans l'appartement en réalisant des actions sur les différents dispositifs en fonction de ses préférences. Si le niveau de luminosité courant ne le satisfait pas, l'utilisateur allume la lumière. Par contre, il l'éteint si la luminosité atteint un niveau assez élevé (il peut se passer de la lampe). Enfin, avant de sortir d'une pièce, il prend soin d'éteindre d'abord la lampe derrière lui.

Résultats Dans un premier temps, nous avons lancé plusieurs simulations sans intégrer *Amadeus*, afin d'observer le nombre d'actions réalisées par l'utilisateur chaque jour. Au total, nous avons effectué 20 simulations, chacune d'elles simulant 50 jours. Bien que l'utilisateur possède des préférences fixes en terme de luminosité, ses déplacements étant aléatoires, nous observons des différences en nombre d'actions par jour. Ce nombre varie entre 66 et 114 actions par jour, avec une moyenne de 88.1 actions par jour. La figure 5.1 illustre le nombre d'actions réalisées par l'utilisateur au cours d'une simulation type (extraite de l'ensemble des simulations). De façon générale, le nombre d'actions réalisées par jour en fonction de la simulation est représenté par la figure 5.2.

Pour représenter les résultats obtenus, nous utilisons des boites de Tukey, plus représentative qu'une simple moyenne des actions. Pour chaque jour, l'extrémité supérieure du trait continu représente la valeur maximale des valeurs obtenues, tandis que l'extrémité inférieure représente la valeur minimale. Le rectangle recouvre l'ensemble des valeurs situées entre le premier et le troisième quartile, c'est-à-dire les valeurs telles que 25% des données sont situées en dessous du premier quartile, et 25% des données sont situés au dessus du troisième quartile. L'écart interquartile correspond donc à 50% des valeurs situées dans la partie centrale de la distribution, et est donc utilisé comme indicateur de dispersion.

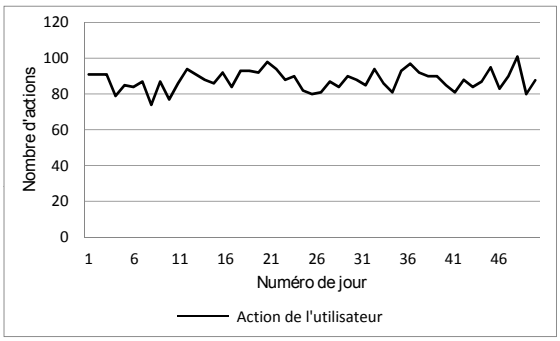

Figure 5.1 – Nombre d'actions de l'utilisateur dans une simulation type

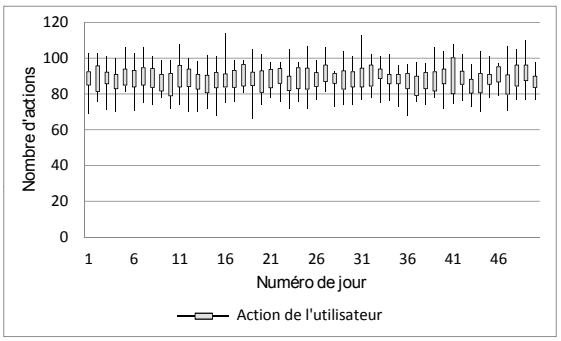

Figure 5.2 – Nombre d'actions de l'utilisateur dans l'ensemble des simulations

Comme nous avons intégré à notre simulateur la possibilité de rejouer une même simulation, ces 20 simulations ont été rejouées une seconde fois, mais en y ajoutant *Amadeus*. La figure 5.3 illustre le nombre d'actions réalisées par l'utilisateur, ainsi que le nombre d'actions réalisées par *Amadeus*, sur la même simulation type que celle présentée sur la figure 5.1. De façon générale, le nombre d'actions réalisées chaque jour par l'utilisateur et par *Amadeus* au cours des différentes simulations est représenté par la figure 5.4.

Nous pouvons observer sur ces figures la très forte diminution du nombre d'actions

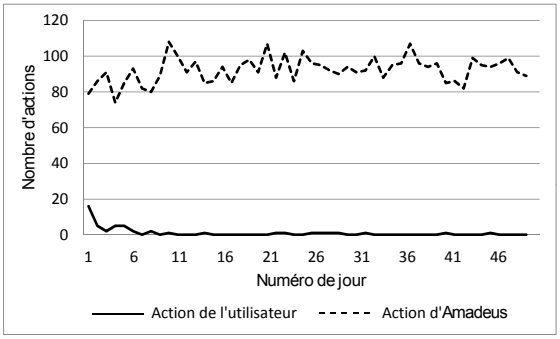

Figure 5.3 – Nombre d'actions de l'utilisateur et d'*Amadeus* dans une simulation

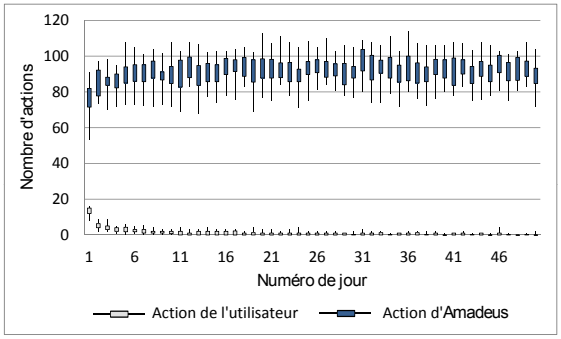

Figure 5.4 – Nombre d'actions de l'utilisateur et d'*Amadeus* dans l'ensemble des simulations

de l'utilisateur au cours de la simulation. Dès le premier jour, *Amadeus* réussit à réaliser un nombre important d'actions à la place de l'utilisateur. Le premier jour de chaque simulation, le nombre moyen d'actions de l'utilisateur passe de 87.4 sans *Amadeus* à 13.2 actions avec *Amadeus*. Si nous observons les actions réalisées durant les 10 premiers jours des différentes simulations, nous pouvons voir que le nombre moyen d'actions de l'utilisateur passe de 87.9 sans *Amadeus* à 3.7 avec *Amadeus*. A la fin de cette période, *Amadeus* a appris la grande majorité des actions à réaliser. Sur la période des 40 jours restants, le nombre d'actions de l'utilisateur passe donc de 88.1 sans *Amadeus* à 0.5 avec *Amadeus*. Cela représente donc une diminution du nombre d'actions de l'utilisateur de 95.8% durant la période où *Amadeus* est encore en train d'apprendre, et une diminution de 99.4% des actions de l'utilisateur une fois l'apprentissage d'*Amadeus* stabilisé.

Au final, le nombre total d'actions de l'utilisateur au cours d'une simulation de 50 jours passe de 4403.7 en moyenne sans *Amadeus* (figure 5.2) à 58 en moyenne avec *Amadeus* (figure 5.4), 83 dans le pire des cas. Nous pouvons également observer en moyenne un total de 4557.2 actions d'*Amadeus* par simulation (4476 dans le pire des cas). Parmi ces actions, nous observons en moyenne 1.8 actions contredites par simulation (5 dans le pire des cas), soit un taux de mauvaises actions de 0.04%, le taux de contradiction le plus important étant de 0.11%.

L'ajout d'*Amadeus* entraîne une légère augmentation de 4.2 actions réalisées en moyenne par jour par les utilisateurs et *Amadeus* (211.5 au total par simulation), soit une augmentation de 4.8 actions réalisées par jour. Parmi ces actions supplémentaires, 1.8 sont dues à une erreur d'*Amadeus*, et 1.8 sont donc dues à l'action de l'utilisateur pour corriger *Amadeus*, ce qui laisse encore une moyenne de 207.9 actions supplémentaires. Ce phénomène s'explique par le fait que, dans certaines situations, *Amadeus* considère à tort que l'utilisateur s'apprête à réaliser une action, mais la réalisation de cette action ne gêne pas pour autant l'utilisateur. Considérons par exemple que l'utilisateur allume la lampe si le niveau de luminosité tombe en dessous du seuil τ_{min}, et l'éteint si la luminosité est au dessus de τ_{max}. Les préférences de l'utilisateur incluent une certaine souplesse ; allumer la lampe quand la luminosité est à τ_{min} n'implique donc pas que la luminosité soit exactement à τ_{max}, mais plutôt à une valeur légèrement inférieure. Par conséquent, si *Amadeus* allume la lampe alors que la luminosité est à un niveau légèrement supérieur à τ_{min}, il est possible que le niveau de luminosité soit

alors légèrement inférieur à τ_{max}. Ce phénomène entraîne donc un certain nombre d'actions supplémentaires de la part d'*Amadeus* qui sont acceptées par l'utilisateur.

Enfin, la figure 5.5 représente le nombre d'agents *contexte* créé au cours des différentes simulations. Nous pouvons constater qu'après une période d'apprentissage où le nombre d'agents *contexte* créés augmentent, il commence à se stabiliser autour du 20ème jour, pour se stabiliser définitivement autour du 35ème jour.

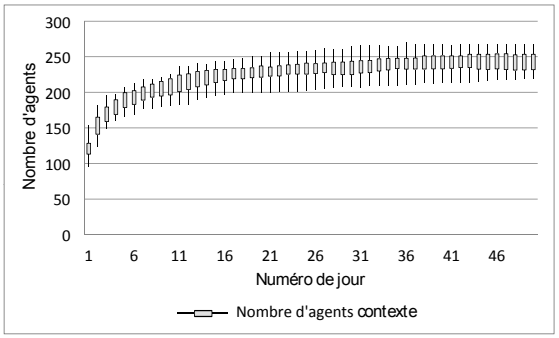

Figure 5.5 – Nombre d'agents *contexte* d'*Amadeus*

5.2.2 Indépendance de l'apprentissage entre deux dispositifs

Dans cette seconde évaluation, nous nous intéressons à la capacité d'*Amadeus* à apprendre quel comportement attribuer à différents dispositifs hétérogènes. Notre objectif est notamment d'observer l'indépendance des traitements d'apprentissage réalisés par différents AMAS *dispositif* : chaque AMAS *dispositif* est capable de réaliser son processus d'apprentissage sans avoir besoin de se coordonner avec les autres AMAS *dispositif*.

Cadre d'étude Pour cette étude, nous reprenons la simulation utilisée dans le cadre de l'étude précédente en ajoutant un store électrique dans la même pièce que la lampe. Nous associons alors un AMAS *dispositif* à la lampe, un autre au store électrique, et enfin un autre pour chaque capteur (capteur de présence, capteur de luminosité et capteurs d'état des portes). Tout comme la lampe, l'état du store électrique (ouvert ou fermé) est modifiable par l'utilisateur.

Nous attribuons à l'utilisateur un comportement similaire à l'étude précédente : il se déplace aléatoirement dans l'appartement en s'assurant que la luminosité de chaque pièce est convenable lorsqu'il y est, tout en restant "écologique". Concrètement, cela signifie que, dans ses préférences, lorsqu'il est hors de la pièce, sa seule préoccupation est que la lampe ne soit pas allumée inutilement. Par contre, quand il y est, s'il y fait trop sombre il essaiera d'abord d'éclairer la pièce en ouvrant le store, puis allumera la lampe si ce n'est pas suffisant. Dans le cas où la luminosité est trop forte, il éteindra d'abord la lampe si elle est allumée, et fermera le store sinon.

La fermeture du store diminue l'impact qu'a la luminosité extérieure sur la luminosité intérieure. L'utilisateur ne s'intéressant pas à l'état du store en son absence de la pièce, il en résulte qu'il n'agit dessus généralement que deux fois par jour, pour le fermer quand la

luminosité est trop forte (ce qui, au mieux, n'arrive qu'une fois par jour, voire pas du tout si l'utilisateur n'entre pas dans la pièce durant la période la plus lumineuse de la journée), et pour le réouvrir quand elle est de nouveau trop faible. Le reste des actions de l'utilisateur liées au contrôle du niveau de luminosité se fait sur la lampe. Cette différence permet d'observer aussi les capacités d'apprentissage d'un AMAS *dispositif* avec un dispositif utilisé plus rarement que la lampe.

Résultats Comme pour l'étude précédente, nous commençons par lancer notre série de 20 simulations de 50 jours sans *Amadeus* afin d'observer le nombre d'actions réalisées l'utilisateur. La figure 5.6 représente, sur les 20 simulations, le nombre d'actions réalisées par l'utilisateur, sans différenciation de l'effecteur.

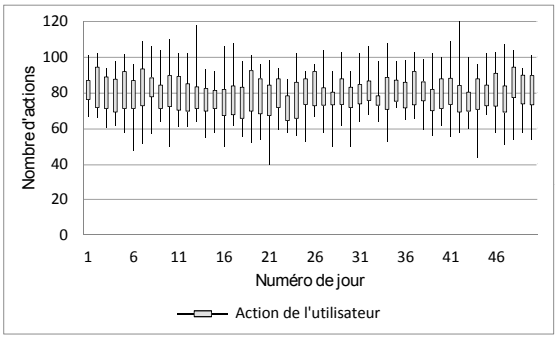

Figure 5.6 – Nombre d'actions de l'utilisateur sur l'ensemble des effecteurs

Les figures 5.7 et 5.8 illustrent le nombre d'actions de l'utilisateur en fonction de l'effecteur. Le faible nombre d'actions par jour sur le store rend inapproprié une représentation sous forme de boite de Tukey ; c'est pourquoi nous illustrons plutôt le nombre d'actions sur le store en affichant le nombre moyen d'actions par jour sur l'ensemble des simulations.

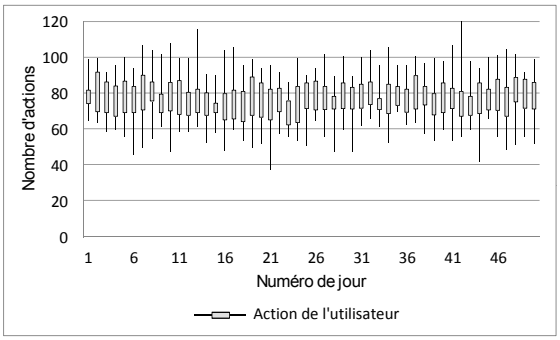

Figure 5.7 – Nombre d'actions de l'utilisateur sur la lampe

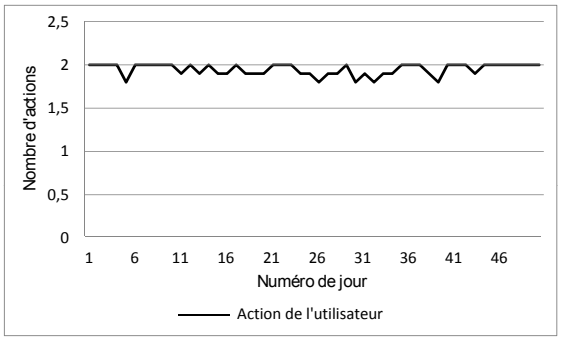

Figure 5.8 – Nombre d'actions de l'utilisateur sur le store électrique

Le nombre d'actions par jour de l'utilisateur sur la lampe (5.7) varie entre 38 et 128, et atteint une moyenne de 84 actions par jour. Ce nombre est inférieur à celui obtenu dans la simulation précédente, car l'utilisateur possède un autre effecteur sur lequel il peut agir pour augmenter la luminosité. Concernant le store (5.8), le nombre d'actions est généralement de deux par jour, à l'exception de certaines journées sans action (moyenne de 1.9 actions par jour).

Nous étudions maintenant les mêmes simulations en y ajoutant *Amadeus*. Nous présentons dans la figure 5.9 le nombre d'actions réalisées respectivement par l'utilisateur et par *Amadeus* sur l'ensemble des effecteurs et pour l'ensemble des simulations.

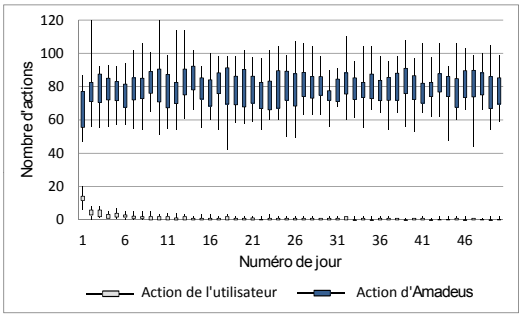

Figure 5.9 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur l'ensemble des effecteurs

Le résultat obtenu est une diminution progressive du nombre d'actions de l'utilisateur, passant d'une moyenne de 80.8 actions le premier jour sans *Amadeus* à une moyenne de 12.55 avec *Amadeus*. Sur les 10 premiers jours, le nombre d'actions de l'utilisateur passe de 80.2 à 3.5. Pour les 40 jours suivants, le nombre d'actions de l'utilisateur passe de 78.7 à 0.5. Concernant les actions réalisées par *Amadeus*, nous obtenons une moyenne de 65.4 actions pour le premier jour, et une moyenne de 76.8 sur les 10 premiers jours. Pour les 40 jours suivants, la moyenne est de 79.3 actions par jour. Enfin, parmi les actions d'*Amadeus*, nous observons une moyenne de 3.2 actions contredites par simulation.

A présent, nous nous intéressons aux résultats obtenus en fonction des deux effecteurs. Tout d'abord, les résultats obtenus au niveau de la lampe sont illustrés par la figure 5.10, tandis que ceux obtenus au niveau du store électrique sont illustrés par la figure 5.11. Pour chacun des effecteurs, le nombre d'actions de l'utilisateur décroit au cours du temps, tandis que le nombre d'actions de l'AMAS *dispositif* associé à chaque effecteur augmente puis se stabilise (très vite dans le cas de la lampe).

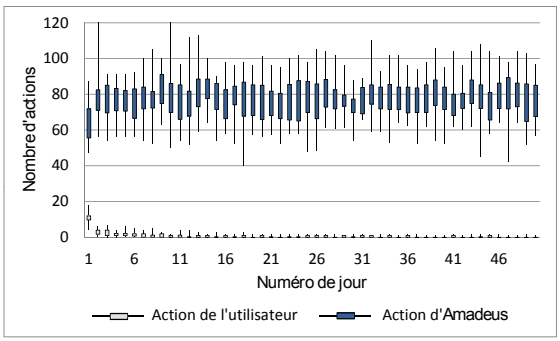

Figure 5.10 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur la lampe

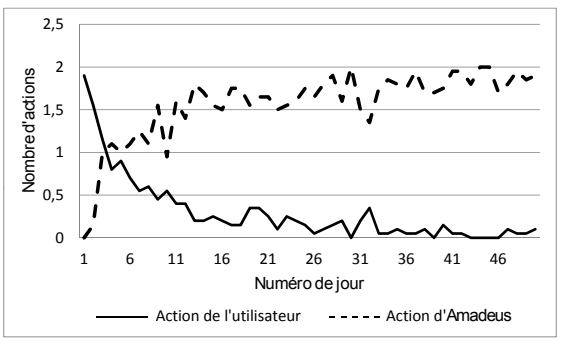

Figure 5.11 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur le store électrique

5.2.3 Discussion

Cette première étude a permis de réaliser une première évaluation des capacités d'apprentissage d'*Amadeus*. Nous avons ainsi pu constater sa capacité à apprendre quel comportement attribuer à un unique dispositif, en agissant de façon proactive sur l'état de son effecteur. Nous pouvons constater en particulier le très faible taux de mauvaises actions du système car, sans changement de préférences de l'utilisateur, la reproduction de ses actions passées permet d'obtenir un comportement satisfaisant. Le résultat le plus intéressant se situe au niveau du temps d'apprentissage : celui-ci se réalisant en ligne, *Amadeus* n'a pas besoin d'attendre d'avoir un comportement appris complet pour commencer à agir ; il réalise des actions à la place de l'utilisateur dès le premier jour. Son comportement s'enrichit ensuite progressivement jusqu'à réaliser la majorité des actions de l'utilisateur.

La seconde étude nous a ensuite confirmé que chaque AMAS *dispositif* est capable d'apprendre quel comportement attribuer à son dispositif indépendamment du traitement des autres AMAS *dispositif*. Cette indépendance entre l'apprentissage réalisé sur les deux dispositifs peut s'avérer surprenant. Par exemple, imaginons la situation où la luminosité est faible, avec la lampe éteinte et le store fermé. D'un point de vue global, il est clair que deux solutions sont possibles, mais qu'ouvrir le store augmentera la luminosité sans impliquer de consommation d'énergie. C'est d'ailleurs à travers ce raisonnement que l'utilisateur décide de son action (en accord avec les règles de préférence que nous lui avons attribuées). Cependant, en appliquant un apprentissage local à chaque dispositif, aucun processus n'apprend explicitement que pour augmenter la luminosité, il faut d'abord ouvrir le store et ensuite seulement allumer la lampe, et pourtant le bon comportement est néanmoins adopté.

Nous constatons donc que, même si *Amadeus* ne réalise pas d'apprentissage global, chaque AMAS *dispositif* est capable de réaliser un apprentissage local correct. En effet, il apprend les actions que l'utilisateur réalise sur son dispositif en fonction des données perçues, incluant l'état des autres effecteurs. Si la réalisation d'une action sur un dispositif dépend de l'état d'un second dispositif, cette dépendance existera aussi au niveau du comportement de l'utilisateur.

Par exemple, dans notre simulation, l'AMAS *dispositif* associé à la lampe apprend que lorsque l'utilisateur allume cette lampe, le store est toujours ouvert. Il n'y a donc pas de décision d'*Amadeus* de préférer ouvrir le store plutôt que d'allumer la lampe, mais uniquement la décision au niveau de l'AMAS *dispositif* de la lampe de la maintenir éteinte, et la décision au niveau de l'AMAS *dispositif* du store de l'ouvrir. Cela montre qu'un apprentissage complet et centralisé du comportement de l'utilisateur n'est pas nécessaire.

L'intérêt de cette propriété réside dans le fait qu'il n'est pas nécessaire de centraliser les traitements d'apprentissage au niveau d'un unique processus. Un comportement correct pour les différents effecteurs d'un système ambiant peut donc être obtenu localement à chaque dispositif en tenant compte de l'état de son environnement (incluant l'état des autres effecteurs).

5.3 Évaluation de la propriété d'ouverture d'*Amadeus*

Nous nous intéressons à présent à la propriété d'ouverture d'*Amadeus*, autrement dit la capacité d'un AMAS *dispositif* associé à un dispositif à continuer à fonctionner correctement si un autre AMAS *dispositif* associé à un dispositif apparaît ou disparaît en cours de fonctionnement. Deux études sont présentées, pour montrer l'effet provoqué respectivement par l'apparition et par la disparition d'un AMAS *dispositif*, l'ensemble des résultats étant discuté après la seconde étude.

Notons que dans les évaluations de ce chapitre, ce n'est pas le second dispositif qui apparaît ou disparaît de la simulation, mais uniquement l'AMAS *dispositif* qui lui est associé. Le comportement de l'utilisateur n'évolue donc pas au cours de la simulation, mais l'AMAS *dispositif* du premier dispositif ne perçoit l'état du second dispositif que lorsqu'un AMAS *dispositif* lui est associé. Le fonctionnement d'*Amadeus* en cas de changement de préférences de l'utilisateur sera présenté dans la section 5.4.

De plus, toutes les données perçues par chaque AMAS *dispositif* sont utiles à son apprentissage. L'impact provoqué par la présence de variables inutiles sera traité en section 5.6.

5.3.1 Ajout d'un AMAS *dispositif*

Nous étudions dans un premier temps le comportement d'*Amadeus* lors de l'ajout d'un AMAS *dispositif* associé à un des dispositifs du système ambiant en cours de fonctionnement.

Cadre d'étude Le cadre d'étude de cette évaluation est le même que celui de l'étude précédente, soit un appartement de plusieurs pièces avec une lampe et un store électrique dans une des pièces. Cependant, un AMAS *dispositif* est associé à cette lampe dès le début de la simulation, tandis que l'AMAS *dispositif* associé au store électrique n'est ajouté qu'après 25 jours de simulation. Dans les premiers 25 jours, l'utilisateur peut utiliser le store, mais l'AMAS *dispositif* associé à la lampe n'en perçoit pas l'état.

Résultats Pour commencer, nous présentons les actions réalisées par l'utilisateur et par *Amadeus* sur l'ensemble des effecteurs avec un AMAS *dispositif* associé à la lampe dès le début des simulations, puis avec un second AMAS *dispositif* associé au store électrique à partir du 25ème jour. Ces résultats sont présentés figure 5.12.

Les actions réalisées par l'utilisateur et par *Amadeus* respectivement sur la lampe et sur le store électrique sont représentées par les figures 5.13 et 5.14.

Au niveau du store, durant la première moitié de la simulation, l'intégralité des actions sont réalisées par l'utilisateur. Une fois qu'un AMAS *dispositif* lui est associé, celui-ci commence son apprentissage et prend progressivement la main sur le contrôle du store. Le fait qu'il soit ajouté après l'AMAS *dispositif* de la lampe n'a pas d'impact sur son traitement. Les traitements d'apprentissage réalisés par les différents AMAS *dispositif* étant indépendant, il

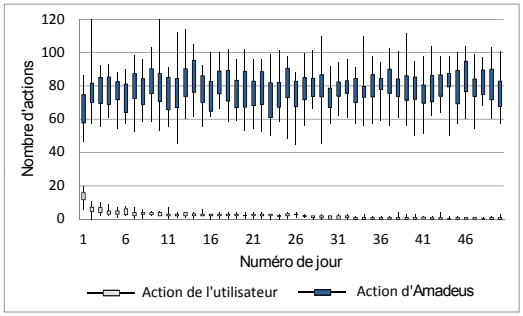

Figure 5.12 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur l'ensemble des effecteurs

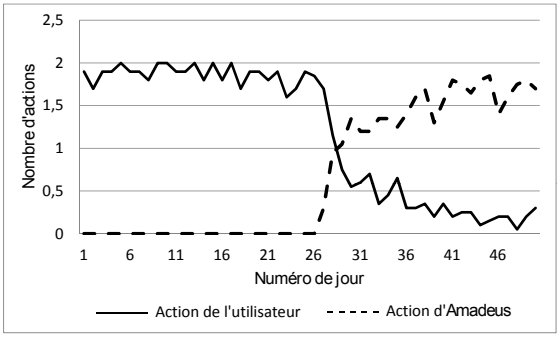

Figure 5.13 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur le store électrique

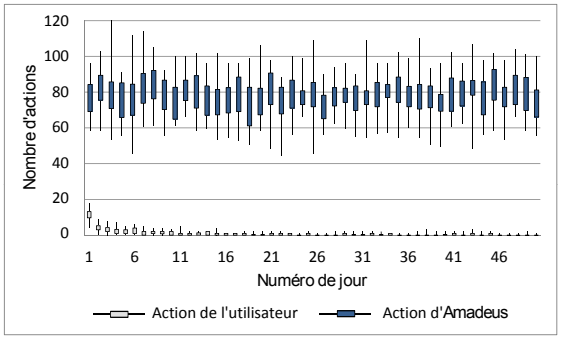

Figure 5.14 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur la lampe

n'est pas nécessaire qu'ils soient réalisés simultanément ; un AMAS *dispositif* peut réaliser son apprentissage après qu'un autre AMAS *dispositif* ait déjà commencé le sien.

Nous pouvons observer (5.14) que, durant la première moitié de la simulation, l'AMAS *dispositif* associé à la lampe tente d'apprendre quel comportement attribuer à cet effecteur sans connaître l'état du store électrique. L'absence de cette donnée a un impact négatif sur son processus d'apprentissage, le nombre moyen d'actions de l'utilisateur restantes par jour sur la première moitié de la simulation sans connaître l'état du store étant de 1.55 contre 1.28 actions par jour en moyenne en connaissant l'état du store (en associant un agent *capteur* au store plutôt qu'un agent *effecteur*, ce qui permet d'en connaître l'état sans pour autant permettre à *Amadeus* de le modifier). Cet impact reste tout de même faible, les situations où la connaissance de l'état du store impacte le choix de l'action à réaliser sur la lampe se présentant rarement.

En revanche, une fois qu'un AMAS *dispositif* est associé au store électrique, un agent *donnée* associé à l'état du store est créé au niveau de l'AMAS *dispositif* de la lampe, et les agents *contexte* de cet AMAS perçoivent alors cette nouvelle donnée. Pour la majorité des agents existants, cette valeur n'a pas d'impact particulier sur leur fonctionnement. Cependant, les nouveaux agents *contexte* créés par la suite (soit à cause d'une action normale de l'utilisateur, soit à cause d'une contradiction) sont capables de proposer des actions dont la validité dépend de l'état du store. Par exemple, en cas de luminosité faible, un ancien agent *contexte* qui propose d'allumer la lampe sans tenir compte de l'état du store finira par être contredit, et sera considéré comme moins satisfaisant que l'agent *contexte* créé suite à la contradiction

et qui sait qu'il faut maintenir la lampe éteinte si le store est fermé. Notons dans ce cas que le premier agent n'aura pas besoin d'adapter sa proposition, elle restera valide dans les autres situations (quand le store est ouvert) et ne sera pas sélectionné dans cette situation là (store fermé) car un autre agent *contexte* sera plus satisfaisant que lui.

5.3.2 Disparition d'un AMAS *dispositif*

A présent, nous étudions le comportement d'*Amadeus* en cas de disparition en cours de fonctionnement d'un AMAS *dispositif* associé à un des dispositifs du système ambiant.

Cadre d'étude Le cadre d'étude est quasi-identique à celui de l'étude précédente, à la différence qu'un AMAS *dispositif* est associé au store électrique dès le début de la simulation. Puis, après 25 jours de simulation, l'AMAS *dispositif* associé au store électrique est retiré. *Amadeus* ne réalise alors plus d'actions sur le store électrique, et l'AMAS *dispositif* associé à la lampe ne perçoit plus l'état du store.

Résultats Le nombre d'actions réalisées par l'utilisateur et *Amadeus* est représenté par la figure 5.15. Nous pouvons alors constater que sur la première moitié de la simulation, le résultat obtenu est le même que celui de l'étude précédente, avec deux dispositifs fonctionnant en parallèle.

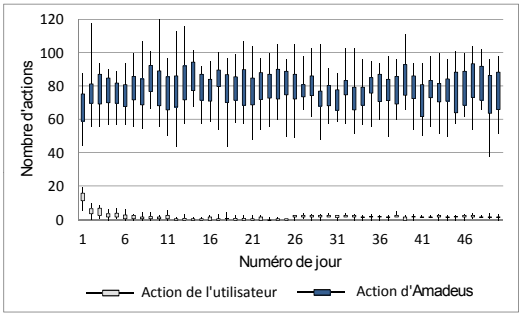

Figure 5.15 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur l'ensemble des effecteurs

A la moitié de la simulation, l'AMAS *dispositif* associé au store électrique disparaît. Cependant, le nombre d'actions réalisées sur ce dispositif étant relativement faible, l'impact de sa disparition sur le nombre total d'actions réalisées durant les 25 derniers jours de la simulation est assez faible. Pour observer ce phénomène, nous nous focalisons sur les actions réalisées d'une part sur le store, présentées figure 5.16, et d'autre part sur la lampe, présentées figure 5.17.

Au niveau du store, nous pouvons observer que l'AMAS *dispositif* associé prend progressivement la main sur le store électrique jusqu'à la moitié de la simulation. Puis une fois qu'il est supprimé, l'ensemble des actions est alors réalisé par l'utilisateur.

L'AMAS *dispositif* associé à la lampe fonctionne normalement durant la première moitié de la simulation, mais perd ensuite sa perception de la donnée caractérisant l'état du store. L'ensemble des agents *contexte* de cet AMAS *dispositif* suppriment alors la plage de valeur associée à cette donnée.

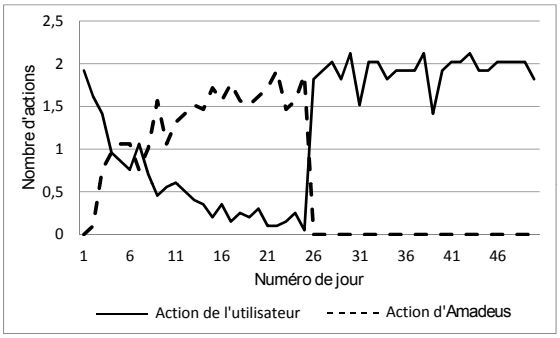

Figure 5.16 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur le store électrique

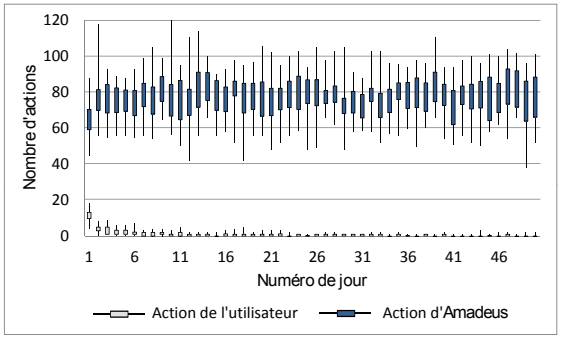

Figure 5.17 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur la lampe

Si nous comparons le nombre moyen d'actions réalisées par l'AMAS *dispositif* associé à la lampe dans la seconde moitié de cette simulation selon qu'il connaisse l'état du store (avec un agent *capteur* associé au store) ou pas, nous passons d'un nombre moyen de 0.26 actions de l'utilisateur dans le premier cas à un nombre moyen de 0.32 actions de l'utilisateur dans le second cas. Tout comme pour la première configuration, la disparition de la donnée décrivant l'état du store n'a qu'un impact très léger sur le fonctionnement de l'AMAS *dispositif* associé à la lampe.

5.3.3 Discussion

Cette évaluation a permis d'illustrer l'ouverture d'*Amadeus* au travers de l'ajout et de la suppression en cours de fonctionnement d'un AMAS *dispositif* associé à un dispositif. Dans notre cas, nous avons ajouté et supprimé l'AMAS *dispositif* associé au store électrique. En cas d'apparition d'une donnée, les nouveaux agents *contexte* sont plus précis dans leurs propositions, mais les anciens agents *contexte* continuent néanmoins d'exister et de proposer des actions. A l'inverse, en cas de disparition d'une donnée, les agents *contexte* perdent alors une partie de leurs connaissances mais restent néanmoins valides. Cette étude montre d'une part la capacité d'un AMAS *dispositif* à s'intégrer dans un environnement où d'autres AMAS *dispositif* évoluent déjà, et d'autre part la capacité que possède un AMAS *dispositif* à intégrer ou supprimer en cours de fonctionnement une donnée provenant d'un autre AMAS *dispositif*.

L'apparition ou la disparition d'une donnée n'a que peu d'impact négatif dans cette étude, car il portait sur l'état du store qui n'influence guère les décisions d'actions sur la lampe. Bien sûr, il ne s'agit pas pour autant de dire que la perte d'une donnée n'influence jamais l'apprentissage réalisé par un AMAS *dispositif* ; cela n'est vrai que si cette donnée est de base inutile à son fonctionnement. Par exemple, si nous réalisons cette même étude mais en ajoutant (ou supprimant) l'AMAS *dispositif* associé à la lampe plutôt que celui associé au store, ce dernier aurait réagi de la même façon en intégrant (ou supprimant) la donnée caractérisant l'état de la lampe, et en adaptant alors son fonctionnement. Mais dans ce cas là, l'impact de la perte de la donnée liée à l'état de la lampe sur le fonctionnement du store aurait été plus important, car l'état de lampe a un impact fort sur les décisions d'agir sur le store. Cet impact n'est cependant pas pertinent dans notre étude. L'objectif de cette étude

est de s'assurer qu'un AMAS *dispositif* est résistant à l'évolution du nombre de données perçues. Cela ne change en rien la véracité de l'hypothèse énoncée en section 3.2.1 et établissant que l'apprentissage d'*Amadeus* ne peut fonctionner que s'il dispose des capteurs suffisants pour lui fournir l'ensemble minimal des données requises à son apprentissage. La résistance d'*Amadeus* face à l'apparition ou la disparition de données, dans la mesure où les données restantes suffisent à assurer son fonctionnement, permet donc de confirmer qu'il respecte la propriété d'ouverture.

5.4 Changement du comportement de l'utilisateur

Dans l'étude précédente, nous avons illustré le fonctionnement d'*Amadeus* face à un des changements possibles de son environnement, à savoir l'ajout ou la suppression d'un dispositif en cours de fonctionnement. A présent, nous nous intéressons à un autre changement possible de son environnement, à savoir une évolution dans le comportement de l'utilisateur.

5.4.1 Variation du comportement de l'utilisateur

Dans une première étude, nous analysons l'impact d'une légère variation du comportement de l'utilisateur sur l'apprentissage d'*Amadeus*. Dans ce cas, la majorité du comportement préalablement appris par *Amadeus* reste correct, seules certaines situations le mènent à réaliser de mauvaises actions.

Cadre d'étude Pour cette étude, nous repartons de la simulation constituée d'une pièce avec une lampe et un store électrique, avec un utilisateur évoluant aléatoirement dans l'appartement, et s'assurant que la luminosité de la pièce soit correcte quand il s'y trouve, et que la lampe soit éteinte quand il ne s'y trouve plus. De plus, un AMAS *dispositif* est ajouté à chaque dispositif.

Les simulations durent 50 jours, mais nous réalisons un changement dans l'environnement d'*Amadeus* au bout de 25 jours. Ce changement consiste en un changement de comportement de l'utilisateur, à travers une légère variation de ses préférences en terme de luminosité. Avec ses nouvelles préférences, il cherche à obtenir une luminosité globalement plus élevée. Concrètement, cela signifie qu'il allumera la lampe ou ouvrira le store électrique plus rapidement, tandis qu'il attendra davantage avant d'éteindre la lampe ou de fermer le store électrique. Son comportement reste cependant relativement similaire une fois le changement de préférences réalisé.

L'objectif de cette étude est donc d'observer comment *Amadeus* est capable de s'adapter à des variations du comportement de l'utilisateur sans nécessiter de reprendre l'apprentissage depuis le début ; pour toutes les situations où ces variations n'impliquent pas de changement dans le comportement des utilisateurs, le comportement déjà appris est toujours correct.

Résultats Tout d'abord, nous réalisons une série de 20 simulations de 50 jours sans intégrer *Amadeus*, avec un changement dans les préférences de l'utilisateur au 25ème jour. Les actions réalisées en fonction du dispositif sont représentées par la figure 5.18 pour la lampe, et par la figure 5.19 pour le store. Sur ces figures, nous pouvons observer une augmentation du nombre d'actions réalisées sur les différents effecteurs. En effet, le nombre moyen d'actions par jour réalisées par l'utilisateur passe de 78.9 dans la première moitié de la simulation à 80.9 dans la seconde moitié. Plus précisément, le nombre d'actions moyen par jour passe de 77.1 à 79.7 au niveau de la lampe, tandis que le nombre d'actions moyen par jour passe de 1.8 à 1.2 au niveau du store électrique.

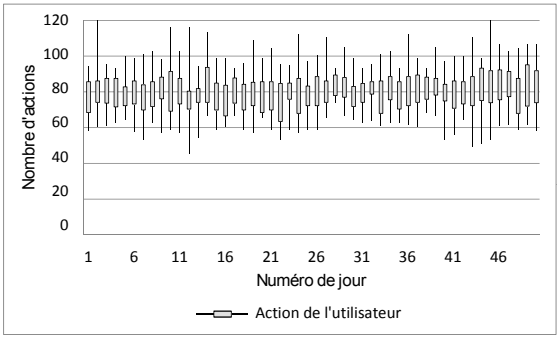

Figure 5.18 – Nombre d'actions de l'utilisateur sur la lampe

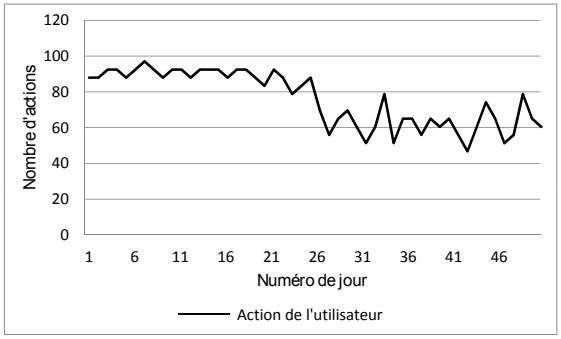

Figure 5.19 – Nombre d'actions de l'utilisateur sur le store électrique

A présent, nous réitérons les simulations en ajoutant *Amadeus*. Nous cherchons à observer ainsi l'impact que la modification dans les préférences de l'utilisateur a sur le comportement d'*Amadeus*. Par soucis de clarté, nous représentons les actions réalisées par l'utilisateur et par *Amadeus* réalisées sur la lampe (figure 5.20) séparément de celles réalisées sur le store électrique (figure 5.21). Néanmoins, nous nous intéressons de façon plus générale aux actions réalisées par *Amadeus*, indépendamment du dispositif.

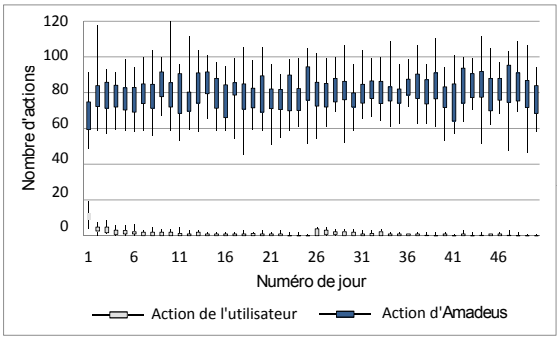

Figure 5.20 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur la lampe

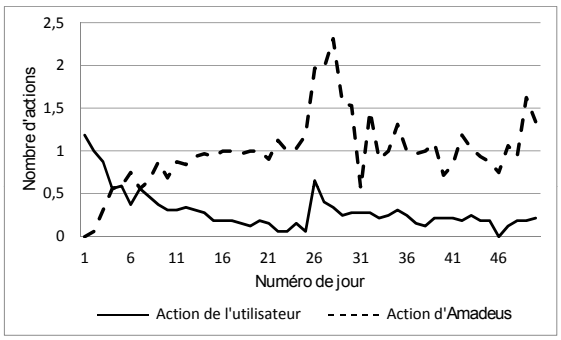

Figure 5.21 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur le store électrique

Pour la première moitié de la simulation, *Amadeus* réalise en moyenne 65.4 actions le premier jour contre 12.8 actions de l'utilisateur, ce qui fait un total de 83.6% des actions réalisées par *Amadeus* le premier jour. Si nous étendons la comparaison aux dix premiers jours, nous obtenons une moyenne de 76.4 actions par jour réalisées par *Amadeus* contre 3.9 réalisées par l'utilisateur, ce qui fait un total de 95% des actions réalisées par *Amadeus*. Enfin,

sur les 15 jours suivants qui concluent la première moitié de la simulation, *Amadeus* réalise en moyenne 78.6 actions par jours tandis que l'utilisateur en réalise en moyenne 0.9 par jour. Avec un total de 98.9% des actions réalisées par *Amadeus* durant ces 15 jours, nous pouvons considérer que l'apprentissage d'*Amadeus* est terminé au moment de la modification des préférences de l'utilisateur.

Une fois que la variation du comportement de l'utilisateur intervient, nous pouvons observer une légère reprise des actions de l'utilisateur. Ces actions servent à corriger les actions d'*Amadeus* devenues inadaptées aux nouvelles préférences de l'utilisateur. La figure 5.22 illustre plus précisément les contradictions détectées par le système au cours des différentes simulations. Durant la première moitié, les contradictions sont ponctuelles. Puis au moment des changements de préférences de l'utilisateur, le nombre de contradiction augmente fortement, puis diminue progressivement tandis qu'*Amadeus* réajuste son apprentissage au nouveau comportement de l'utilisateur.

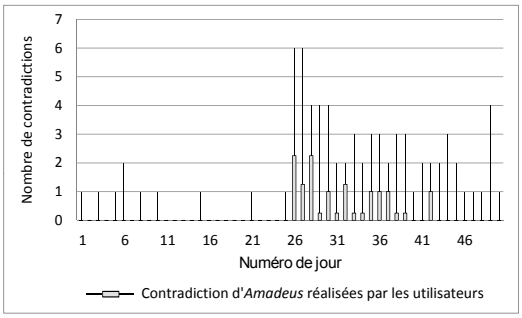

Figure 5.22 – Nombre moyen de contradictions d'*Amadeus* réalisées par l'utilisateur

Au final, sur les dix premiers jours de cette seconde moitié de la simulation, le nombre moyen d'actions par jour réalisées par *Amadeus* est de 80.5, tandis que l'utilisateur réalise une moyenne de 1.8 actions par jour. Si nous comparons ces résultats avec le nombre moyen d'actions réalisées par l'utilisateur sans *Amadeus*, qui est de 80.3, nous observons donc une augmentation de 2 actions par jour en moyenne. Cette augmentation est principalement due aux contradictions, avec une moyenne de 0.7 actions d'*Amadeus* contredites par jour (6 dans le pire des cas), auxquelles se rajoutent donc 0.7 actions par jour en moyenne de contradictions par l'utilisateur.

Sur le reste de la simulation (jours 35-50), par contre, le nombre moyen d'actions par jour d'*Amadeus* est de 81.3 tandis que le nombre moyen d'actions par jour de l'utilisateur est de 0.6. *Amadeus* réalise donc 99.2% des actions. Une fois passée la phase de réadaptation, *Amadeus* a donc adopté un nouveau comportement adéquat vis-à-vis des nouvelles préférences de l'utilisateur.

5.4.2 Changement du comportement de l'utilisateur

A présent, nous étudions l'impact d'un changement important dans le comportement de l'utilisateur. Il s'agit donc pour *Amadeus* de s'adapter à un changement radical qui invalide complètement le comportement appris préalablement.

Cadre d'étude Nous reprenons la simulation précédente, mais en apportant une modification bien plus importante des préférences de l'utilisateur vis-à-vis de la luminosité de son environnement qu'il va souhaiter très faible. Dès lors, il ne tolère plus que la lampe soit allumée, et ne laisse le store électrique ouvert que si la luminosité est très faible (il désire donc rester dans l'obscurité, avec le store ouvert la nuit). L'objectif est alors de voir si *Amadeus* est aussi capable de modifier complètement son apprentissage, en désapprenant le comportement préalablement appris afin d'adopter un nouveau comportement plus pertinent.

Nous suivons la même procédure expérimentale que pour la première étude, en réalisant une série de 20 simulations sans *Amadeus*, puis en réalisant ces 20 mêmes simulations avec *Amadeus*. Pour chacune de ces simulations, nous modifions complètement les préférences de l'utilisateur à la fin du 25ème jour.

Résultats Les actions de l'utilisateur sur la lampe sans *Amadeus* sont représentées par la figure 5.23, tandis que la figure 5.24 illustre les actions de l'utilisateur sur le store. Pour la première moitié de la simulation, les résultats obtenus sont identiques à ceux obtenus dans la première partie de cette étude. En revanche, pour la deuxième partie de l'étude, l'utilisateur cesse définitivement d'utiliser la lampe. Concernant le store, le nombre d'actions augmente légèrement et se stabilise à deux actions par jour quoiqu'il arrive. Notons qu'il ne s'agit pas des mêmes actions : dans la première moitié de la simulation, l'utilisateur ferme les stores en pleine journée pour diminuer la forte luminosité, alors que dans la seconde moitié de la simulation, l'utilisateur les garde fermés sauf si la luminosité est très faible.

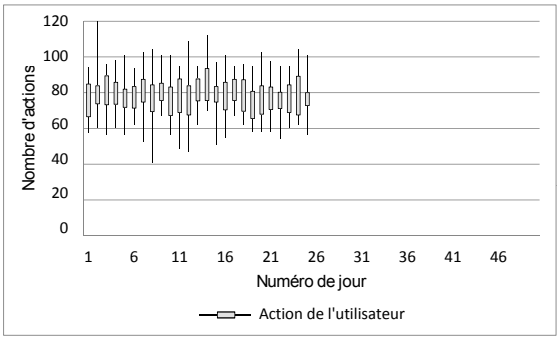

Figure 5.23 – Nombre d'actions de l'utilisateur sur la lampe

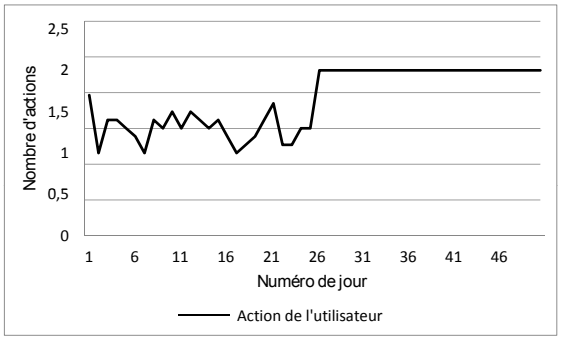

Figure 5.24 – Nombre d'actions de l'utilisateur sur le store électrique

Nous rajoutons maintenant *Amadeus* à nos simulations. Nous observons alors sa capacité à apprendre le comportement à attribuer aux différents dispositifs malgré le changement important dans les préférences de l'utilisateur. Les actions réalisées par l'utilisateur et par *Amadeus* sont représentées par la figure 5.25 pour la lampe, et par la figure 5.26 pour le store électrique.

Les performances d'*Amadeus* durant la première moitié de la simulation sont les mêmes que pour la première partie de l'étude. Dès lors, nous pouvons considérer que son apprentissage est terminé au moment où nous appliquons notre perturbation. Une fois les préférences de l'utilisateur modifiées, nous pouvons observer au niveau des deux dispositifs une augmentation des actions de l'utilisateur. Au niveau de la lampe (5.25), il ne s'agit que de

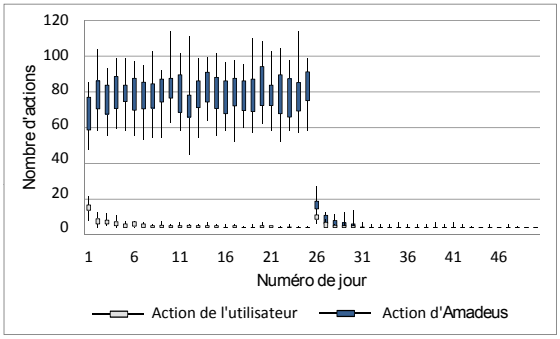

Figure 5.25 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur la lampe

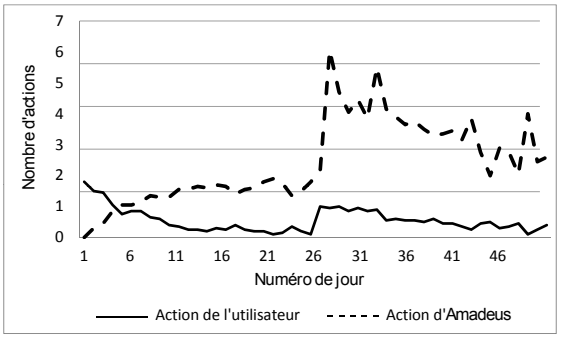

Figure 5.26 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur le store électrique

contradictions, car l'utilisateur ne veut plus l'allumer. C'est pourquoi, par la suite, l'AMAS *dispositif* associé à la lampe ne réalise plus d'actions sur son dispositif. Par contre, au niveau du store électrique (5.26), les contradictions sont mêlées avec les actions représentatives de son nouveau comportement. Les contradictions réalisées par l'utilisateur sont illustrées par les figures 5.27 (pour la lampe) et 5.28 pour le store électrique.

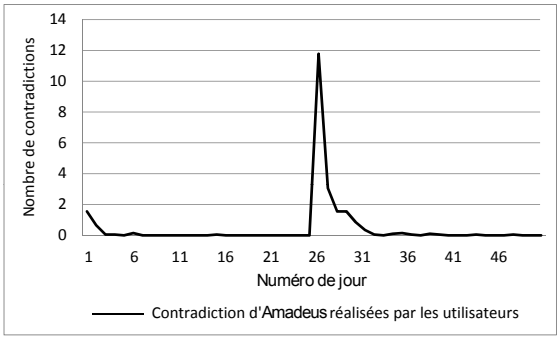

Figure 5.27 – Nombre moyen de contradictions réalisées par l'utilisateur sur l'AMAS *dispositif* associé à la lampe

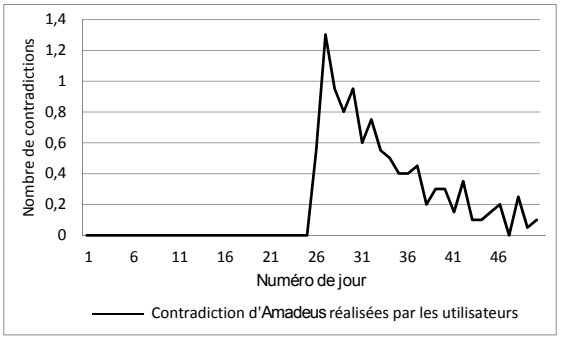

Figure 5.28 – Nombre moyen de contradictions réalisées par l'utilisateur sur l'AMAS *dispositif* associé au store électrique

La majorité des contradictions se font le premier jour, l'utilisateur contredisant systématiquement toute tentative d'*Amadeus* d'allumer la lampe. Puis progressivement, au fur et à mesure que le système s'adapte, le nombre de contradictions diminue. Au niveau de l'AMAS *dispositif* associé à la lampe, l'adaptation consiste donc à ce que les agents *contexte* proposant de maintenir la lampe éteinte soient sélectionnés à la place de ceux proposant de l'allumer, ces derniers ayant été contredits.

Au niveau du store électrique, le même phénomène se produit, bien que ces contradictions soient moins concentrées le premier jour, mais plutôt présentes sur une durée plus diffuse. Cette différence s'explique simplement par le nombre d'actions réalisées sur la lampe qui est plus important que sur le store, ce qui accélère la réalisation d'actions incorrectes sur la lampe, et donc leurs contradictions.

5.4.3 Discussion

Dans cette étude, nous avons appliqué *Amadeus* dans une certaine configuration, puis nous avons attendu que son apprentissage soit terminé pour modifier les préférences de l'utilisateur. Ce changement a entraîné une perturbation dans l'environnement d'*Amadeus*, et a rendu son comportement inadapté.

Dans la première configuration de cette étude, la perturbation a été relativement faible, car nous avons limité le changement à une préférence accrue de l'utilisateur en termes de luminosité. Le comportement de l'utilisateur restait globalement le même, à l'exception des quelques situations où le niveau de luminosité se situait entre l'ancien seuil et le nouveau.

Par conséquent, le changement nécessaire au niveau du comportement appris par *Amadeus* était donc faible lui aussi. La majorité des agents *contexte* créés avant la modification des préférences de l'utilisateur restaient donc valides ; seuls certains d'entre eux deviennent inappropriés par rapport aux nouvelles préférences de l'utilisateur, et sont alors contredites. Les agents *contexte* toujours corrects continuent d'observer que leurs propositions d'actions ne sont pas contredites. En revanche, les capacités d'introspection des agents *contexte* incorrects leur permettent de détecter que leurs propositions n'ont pas satisfait l'utilisateur (lorsqu'ils sont prévenus par leur agent *contrôleur* qu'ils ont été contredits par l'utilisateur), leur permettant alors de s'adapter. Au fur et à mesure que ces agents *contexte* sont contredits (diminuant alors leurs confiances et leurs gains estimés), tandis que d'autres agents *contexte* sont créés (possédant un gain estimé plus élevé), le comportement d'*Amadeus* tend à s'adapter aux nouvelles préférences de l'utilisateur.

Cette étude nous a donc permis de mettre en évidence la capacité d'*Amadeus* à réagir et s'adapter aux légers changements de la fonctionnalité à apprendre. L'originalité de cette capacité réside dans le fait qu'il n'a pas été nécessaire d'interrompre complètement les actions d'*Amadeus* et de relancer son apprentissage depuis le début, la plupart des actions d'*Amadeus* restant valides. Les agents *contexte* représentent des sous-parties de la fonctionnalité apprise. Ainsi, la capacité de ces agents à s'adapter lorsqu'ils s'avèrent incorrects permet à *Amadeus* de ne modifier que la partie de sa fonctionnalité devenue incorrecte sans modifier le reste de son comportement.

Nous nous sommes ensuite intéressés à un autre cas d'étude incluant un changement des préférences de l'utilisateur bien plus important. Celui-ci ne souhaite plus que la lampe soit allumée, et ne veut ouvrir le store électrique que si la luminosité est très basse, le refermant dès que la luminosité recommence à monter. Pour *Amadeus*, c'est donc un changement radical de la fonctionnalité à apprendre. Tous les agents *contexte* associés à la lampe proposant de l'allumer sont donc incorrects. De même, les conditions d'ouverture et de fermeture du store électrique ont complètement changées (l'utilisateur ne l'ouvre plus quand la pièce est très éclairée, mais au contraire quand il fait très sombre), rendant incorrects tous les agents *contexte* proposant de l'ouvrir ou de le fermer.

Ce très fort changement dans la fonctionnalité à apprendre entraîne plusieurs contradictions des actions d'*Amadeus* par l'utilisateur. Ces contradictions ont pour effet de diminuer l'estimation des gains et la confiance des agents contredits, mais aussi d'augmenter l'estimation de gains des agents *contexte* ne proposant pas ces actions. Ainsi, progressivement,

tous ces agents *contexte* incorrects se retrouvent suppléés par les agents *contexte* proposant de maintenir les différents effecteurs dans leurs états courants. C'est ainsi que se réalise le "désapprentissage" d'*Amadeus*, tandis que se réalise simultanément l'apprentissage des actions correctes à réaliser.

Cette étude a donc permis d'observer les capacités d'*Amadeus* à modifier son comportement en cours de fonctionnement en cas de changement dans les préférences de l'utilisateur, que ces changements soient légers ou radicaux. En particulier, cette adaptation ne nécessite pas de recommencer l'apprentissage depuis le début, ni de stopper les actions du système en attendant que la nouvelle fonctionnalité soit totalement apprise.

5.5 Apprentissage en présence de plusieurs utilisateurs

Les études précédentes ont montré la capacité d'*Amadeus* à apprendre quel comportement attribuer à différents dispositifs à partir de l'observation des actions d'un utilisateur. A présent, nous nous intéressons au cas où plusieurs utilisateurs évoluent dans l'environnement.

Cadre d'étude Pour cette étude, nous utilisons une simulation avec une lampe et un store électrique dans une pièce dotée d'un capteur de luminosité, ainsi que de capteurs d'états sur les deux portes permettant de savoir si elles sont ouvertes ou fermées. Nous rajoutons ensuite deux capteurs de présence, chacun de ces capteurs permettant de prévenir de la présence d'un utilisateur particulier.

Dans le cas où les deux utilisateurs ont les mêmes préférences, un simple capteur de présence aurait suffit. *Amadeus* aurait alors pu apprendre le comportement à attribuer aux différents dispositifs, sans distinction entre les deux utilisateurs. Cependant, si les deux utilisateurs ont des préférences différentes, il est nécessaire de pouvoir identifier quels utilisateurs sont présents dans la pièce pour savoir quelle action réaliser.

Nous réalisons donc une série de 20 simulations de 50 jours avec deux utilisateurs évoluant dans l'environnement. Pour toute la durée des simulations, nous attribuons les mêmes préférences que dans les études précédentes au premier utilisateur. Celui-ci évolue donc aléatoirement dans l'appartement, en s'assurant que la luminosité de la pièce soit correcte quand il s'y trouve (en privilégiant l'ouverture des stores), et que la lampe soit éteinte quand il ne s'y trouve plus. Le second utilisateur a un comportement initial similaire au premier utilisateur mais avec des préférences en terme de luminosité légèrement plus faibles (autrement dit, n ous lui attribuons les préférences de l'utilisateur de la section 5.4.1 après la légère variation). Puis, au 25ème jour, nous modifions radicalement les préférences du second utilisateur qui s'assure que la lampe soit toujours éteinte et que le store soit ouvert uniquement si la luminosité est très faible (autrement dit, nous lui attribuons les préférences de l'utilisateur de la section 5.4.2 après changement de comportement).

Dans le cas où les deux utilisateurs sont dans la pièce, il n'y a aucun souci s'ils sont en accord sur l'action à réaliser. En revanche, dans le cas où leurs préférences respectives sont en conflit, nous donnons dans la simulation la priorité au premier utilisateur. C'est donc

lui qui a le dernier mot en cas d'incompatibilité entre les préférences des deux utilisateurs. Cette priorité est donnée au niveau du simulateur, *Amadeus* n'a bien sûr pas connaissance de cette priorité.

Résultats Tout d'abord, nous illustrons le nombre d'actions réalisées par les deux utilisateurs, d'une part sur la lampe figure 5.29, et d'autre part sur le store électrique figure 5.30, en l'absence d'*Amadeus*. Concernant le store, nous pouvons observer une forte augmentation du nombre d'actions dans la seconde moitié des simulations. En effet, les deux utilisateurs ont des préférences similaires dans la première moitié des simulations, qui font qu'ils n'ouvrent et ne ferment les stores à peu près qu'une fois par jour. En revanche, sur la seconde moitié, les préférences des deux utilisateurs étant radicalement différentes, ils ouvrent et ferment bien plus souvent le store.

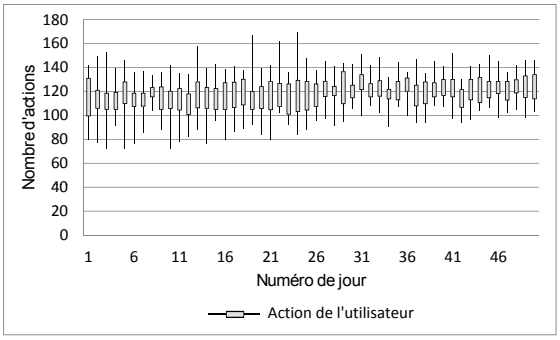

Figure 5.29 – Nombre d'actions de l'utilisateur sur la lampe

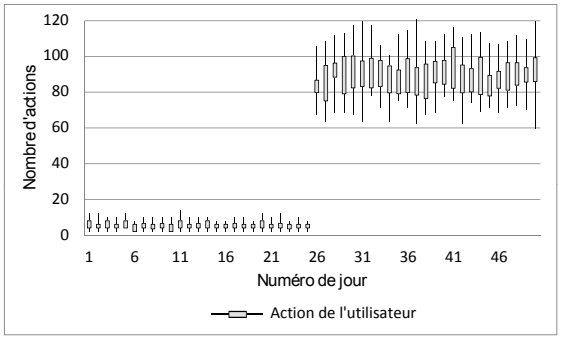

Figure 5.30 – Nombre d'actions de l'utilisateur sur le store électrique

Nous associons ensuite un AMAS *dispositif* aux différents dispositifs du système ambiant, et nous relançons les simulations. Le nombre d'actions réalisées par les utilisateurs sur la lampe et sur le store électrique sont représentés sur les figures suivantes 5.31 (pour la lampe) et 5.32 (pour le store).

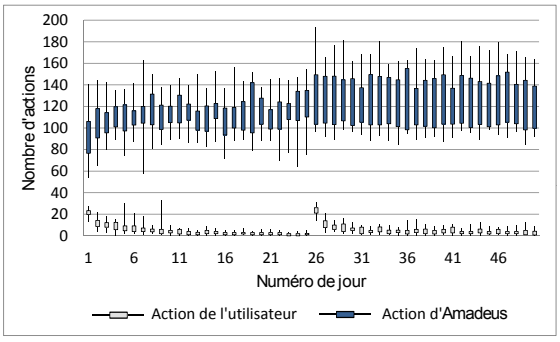

Figure 5.31 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur la lampe

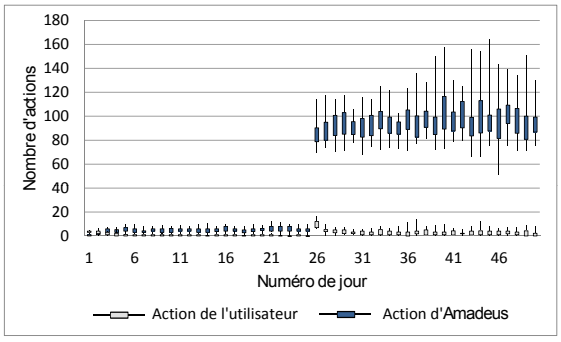

Figure 5.32 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur le store électrique

Le premier jour, le nombre moyen d'actions des utilisateurs passe de 119 sans *Amadeus* à 21.15 avec *Amadeus*, soit une diminution de 82.2% des actions des utilisateurs. Étendue aux 10 premiers jours, cette comparaison montre une diminution de 119.1 actions en moyenne

sans *Amadeus* à 8.7 actions en moyenne avec *Amadeus*. Pour les 15 jours suivants de la simulation, l'apprentissage d'*Amadeus* est quasiment complet, avec un nombre d'actions passant de 119.9 sans *Amadeus* à 2.3 avec *Amadeus*, soit 98% des actions réalisées par *Amadeus* à la place des deux utilisateurs.

Au 26ème jour, *Amadeus* doit modifier une partie de son comportement appris, de façon à satisfaire les nouvelles préférences du second utilisateur. Durant ce 26ème jour, le nombre moyen d'actions des utilisateurs remontent à 23.8 avec *Amadeus*, tandis que 201.9 actions en moyenne sont réalisées par les utilisateurs sans *Amadeus*. Le taux d'actions réalisées par *Amadeus* descend donc à 88.2%. Sur l'ensemble des 10 jours suivant la modification des préférences du second utilisateur, le nombre d'actions des utilisateurs passe de 210.1 sans *Amadeus* à 8.1 avec *Amadeus*, soit un pourcentage de 96.1% d'actions réalisées par *Amadeus*. Enfin, sur les 15 jours de simulation restants, le nombre moyen d'actions réalisées par les utilisateurs passe de 210.3 sans *Amadeus* à 3.7 avec *Amadeus*. Le taux d'actions réalisées par *Amadeus* à la place des utilisateurs est donc de 98.2%, aussi nous pouvons considérer qu'*Amadeus* a bien pris en compte le changement du comportement du second utilisateur dans son apprentissage.

Les figures 5.33 et 5.34 illustrent plus précisément, parmi les actions réalisées par les utilisateurs, celles qui sont des contradictions. Sur la première figure, nous pouvons observer quelques contradictions ponctuelles du premier utilisateur tout le long des simulations. En revanche, sur la second figure, certaines contradictions ponctuelles du second utilisateur sont visibles durant la première moitié des simulations, mais le nombre de contradictions augmente fortement au 26ème jour, lorsque les préférences de ce second utilisateur ont changé. Puis, au fur et à mesure qu'*Amadeus* adapte son comportement, le nombre de contradictions du second utilisateur diminue progressivement.

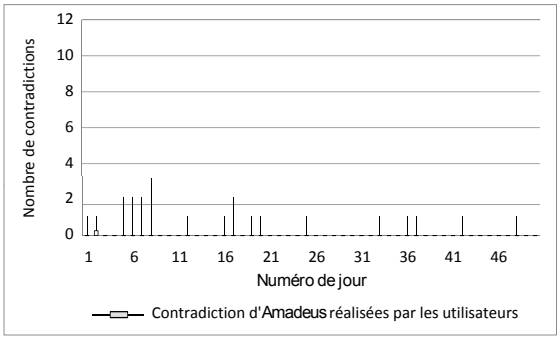

Figure 5.33 – Nombre de contradictions réalisées par le premier utilisateur

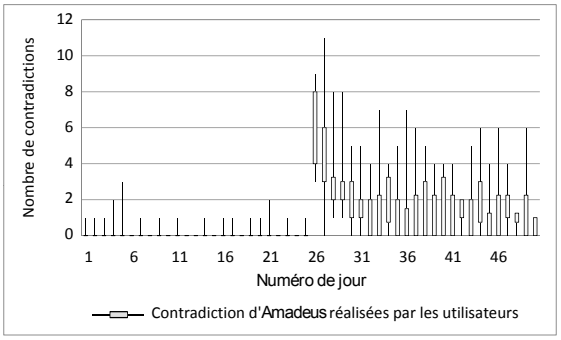

Figure 5.34 – Nombre de contradictions réalisées par le second utilisateur

Discussion L'intérêt de cette étude était de montrer la capacité d'*Amadeus* à apprendre le comportement correct à attribuer aux différents dispositifs d'un système ambiant, même en présence de plusieurs utilisateurs. Pour obtenir ce résultat en dépit des différences de comportements entre les utilisateurs, la seule condition est de posséder une perception de l'identité des différents utilisateurs (dans notre étude, un capteur de présence en fonction de l'utilisateur).

Avec un seul utilisateur, il n'y avait qu'un seul comportement à apprendre. Avec un second utilisateur, nous pouvons considérer qu'il y a trois comportements : un pour chaque utilisateur, et un troisième qui représente le comportement des utilisateurs lorsqu'ils sont présents en même temps dans la pièce. Pour les deux premiers comportements, le traitement réalisé est le même qu'avec un seul utilisateur, et les agents *contexte* associés à ce comportement sont ceux qui sont valides en présence d'un des utilisateurs et en l'absence de l'autre utilisateur. Pour le troisième, il n'y a pas de différence dans le traitement réalisé : les agents *contexte* créés par les actions d'un utilisateur lorsque les deux utilisateurs sont dans la pièce représentent le comportement à attribuer au dispositif en présence des deux utilisateurs.

Cette étude permet d'illustrer la façon dont *Amadeus* gère les conflits entre les préférences de deux utilisateurs. En effet, il serait légitime de se demander comment déterminer quelle action réaliser en présence des deux utilisateurs si nous considérons que le comportement à attribuer au dispositif en présence d'un seul des deux utilisateurs a déjà été appris. Cela semble d'autant plus important si l'action à réaliser est différente en fonction de l'utilisateur présent. Par exemple, quelle action réaliser si les deux utilisateurs sont présents en même temps, sachant que le premier utilisateur allume systématiquement la lampe alors que le second la laisse toujours éteinte ? En l'occurrence, le comportement général d'*Amadeus* reste le même, c'est-à-dire observer les actions des utilisateurs pour déterminer quelle action est la bonne face à une nouvelle situation. Il n'y a aucun connaissance *a priori* sur quelle action satisfera les deux utilisateurs, aussi *Amadeus* observe quelle action est réalisée pour déterminer le bon comportement lorsque les deux utilisateurs sont présents. En d'autres termes, il observe comment les deux utilisateurs agissent lorsqu'ils sont ensemble, et fait de même.

Enfin, cette étude a montré comment *Amadeus* est capable d'adapter une sous-partie de sa fonctionnalité en cas de changement de préférences chez un utilisateur, sans perturber pour autant la sous-partie de sa fonctionnalité qui concerne l'autre utilisateur.

5.6 Filtrage des variables inutiles

Dans cette dernière évaluation, nous nous intéressons à la capacité d'un AMAS *dispositif* à filtrer les variables perçues inutiles à son processus d'apprentissage. Notre objectif est d'observer l'effet néfaste des variables inutiles sur le processus d'apprentissage d'*Amadeus*, et d'étudier comment le processus de filtrage de ces données variables inutiles réalisé par les agents *donnée* (avec l'aide des autres agents) permet de fortement diminuer ce phénomène.

5.6.1 Effet des variables inutiles

Cette première étude a pour objectif de montrer la nécessité d'un filtrage des variables inutiles. Pour cela, nous partons d'un cas d'étude plutôt simple, et nous réalisons plusieurs fois cette même étude en rajoutant des variables inutiles. Ainsi, nous pouvons observer l'impact qu'a l'ajout d'une variable inutile même sur un exemple simple.

Cadre d'étude Nous reprenons ici le cadre d'étude optimal de la première évaluation, avec un unique utilisateur et un unique dispositif (une lampe). L'utilisateur se déplace dans son

appartement, en s'assurant que lorsqu'il rentre dans la pièce où se trouve la lampe, il l'allume si la luminosité est insuffisante. De plus, si la luminosité devient suffisante pour qu'il puisse se passer de la lampe, alors il l'éteint. Il fait de même systématiquement dès qu'il sort de la pièce.

Comme pour la première étude, nous associons un AMAS *dispositif* à la lampe. Nous ajoutons aussi un capteur de luminosité et un capteur de présence dans la même pièce que la lampe, ainsi qu'un capteur d'état au niveau des deux portes permettant d'entrer dans la pièce. L'AMAS *dispositif* associé à la lampe perçoit donc l'état de sa lampe, l'état des deux portes (ouvertes ou fermées), le niveau de luminosité de la pièce, et la présence de l'utilisateur dans la pièce.

Nous ajoutons ensuite des variables inutiles parmi les variables perçues par l'AMAS *dispositif* de la lampe. Nous étudions donc les capacités d'apprentissage d'*Amadeus* en fonction du nombre de variables inutiles perçues, sachant que les capacités de filtrage des variables inutiles des agents *donnée* ont été désactivées. Nous générons donc un total de 80 simulations de 20 jours chacune. Pour les 20 premières simulations, nous n'ajoutons aucune variable inutile, puis nous en ajoutons une pour les 20 simulations suivantes, une seconde pour les 20 simulations d'après, et enfin une troisième pour les 20 dernières simulations.

Ces variables inutiles génèrent des données numériques aléatoires comprises entre 0 et 100. Elles prennent initialement une valeur aléatoire, puis sélectionne aléatoirement une valeur cible à atteindre. Toutes les cinq minutes, elles s'incrémentent ou se décrémentent afin de tendre vers cette valeur cible (la valeur d'une donnée est cyclique, si elle descend en dessous de 0, elle remonte à 100 , et inversement).

Résultats Nous présentons ici les résultats obtenus en fonction du nombre de variables inutiles présentes dans les perceptions de l'AMAS *dispositif*. Sur la figure 5.35(a), nous pouvons observer la réalisation des actions de l'utilisateur et d'*Amadeus* sur la lampe. En l'absence de variables inutiles, *Amadeus* réalise son apprentissage rapidement, réalisant dès le second jour la majorité des actions à la place de l'utilisateur.

En revanche, dès l'ajout de la première donnée inutile, nous pouvons constater sur la figure 5.35(b) que les performances d'*Amadeus* ont diminuée. Cette dégradation des capacités d'apprentissage d'*Amadeus* s'amplifie avec l'ajout de la seconde variable inutile, comme nous pouvons le voir sur la figure 5.35(c). Enfin, avec la troisième variable inutile, nous pouvons voir sur la figure 5.35(d) qu'*Amadeus* n'est pratiquement plus capable de réaliser d'actions à la place de l'utilisateur, à l'exception de quelques actions occasionnelles.

Cette nécessité d'un plus grand nombre d'exemples pour couvrir les différentes situations possibles est aussi visible en observant le nombre d'agents *contexte* créés au cours des différentes simulations. La figure 5.36(a) représente le nombre d'agents *contexte* sans variable inutile. Les figures 5.36(b), 5.36(c) et 5.36(d) représente aussi le nombre d'agents *contexte* mais dans les simulations avec respectivement une, deux et trois variables inutiles.

En l'absence de variable inutile, nous pouvons observer que le nombre d'agents *contexte* se stabilise au bout du 6ème jour. En revanche, plus on rajoute de variables inutiles, plus le nombre d'agents *contexte* devient important. De plus, ce nombre a bien plus de mal à se stabiliser si nous ajoutons des variables inutiles, ce qui montre la difficulté d'*Amadeus* à

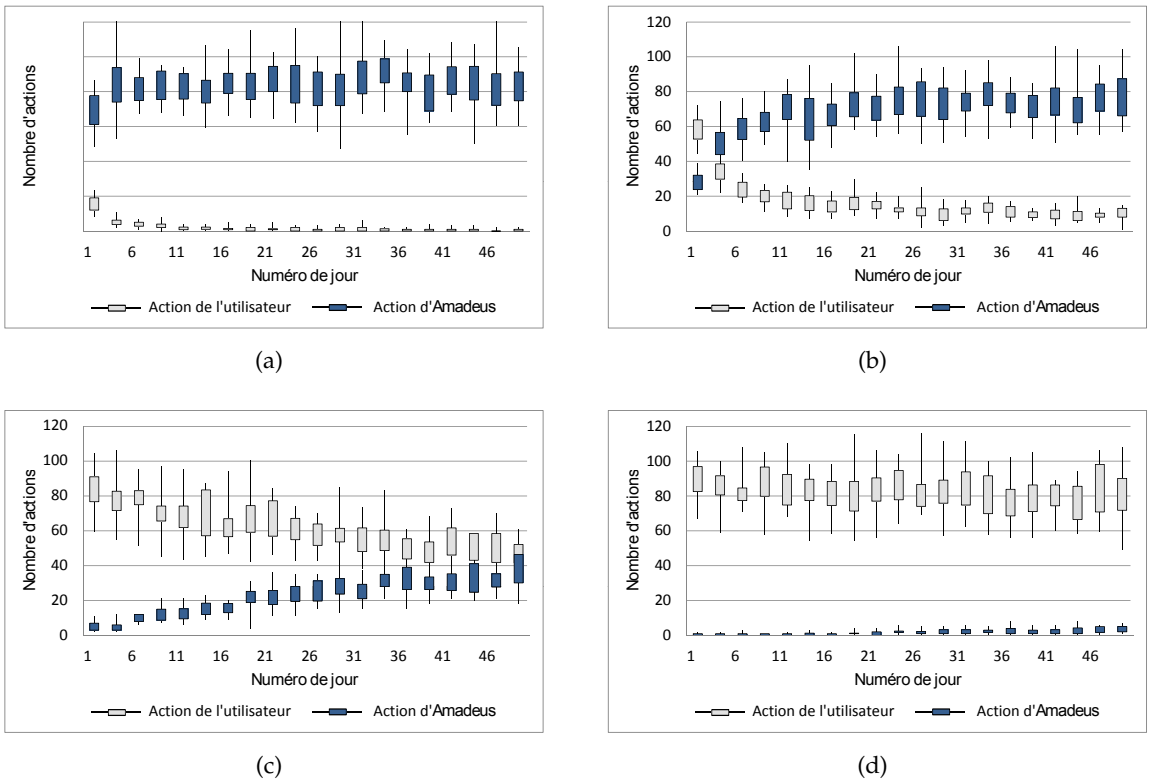

Figure 5.35 – Nombre d'actions réalisées par l'utilisateur et par *Amadeus* en fonction du nombre de variables inutiles

réaliser son apprentissage dans ces conditions.

Plus précisément, le nombre moyen d'actions réalisées par l'utilisateur et *Amadeus* par simulation en fonction du nombre de variables inutiles est représenté par le tableau 5.1. Alors que 97.5% des actions sont réalisées par *Amadeus* en l'absence de variables inutiles, ce nombre diminue à 80.3% avec une variable inutile, puis à 26.5% avec deux variables inutiles, et enfin seules 2% des actions sont réalisées par *Amadeus* avec trois variables inutiles. En revanche, avec une moyenne de 4.7 actions contredites avec 0 variables inutiles, 3.8 actions contredites avec 1 variables inutiles, 2.1 actions contredites avec 2 variables inutiles et 1.6 actions contredites avec 3 variables inutiles, nous pouvons constater que la présence de variables inutiles diminue légèrement le nombre de contradictions que reçoit *Amadeus*. Cela s'explique simplement par le nombre d'actions d'*Amadeus* qui diminue : s'il réalise moins d'actions il est logique qu'il soit moins souvent contredit.

Discussion Cette étude a permis de montrer l'importance d'ajouter à *Amadeus* des capacités de filtrage des variables inutiles. En effet, de telles données dégradent fortement la capacité des agents *contexte* à déterminer les situations pour lesquelles ils sont valides. Ces mauvaises performances sont donc le résultat de la non résolution de la SNC présentée au chapitre 3.2.4.8, qui a donné lieu à la mise en place du filtrage des variables inutiles.

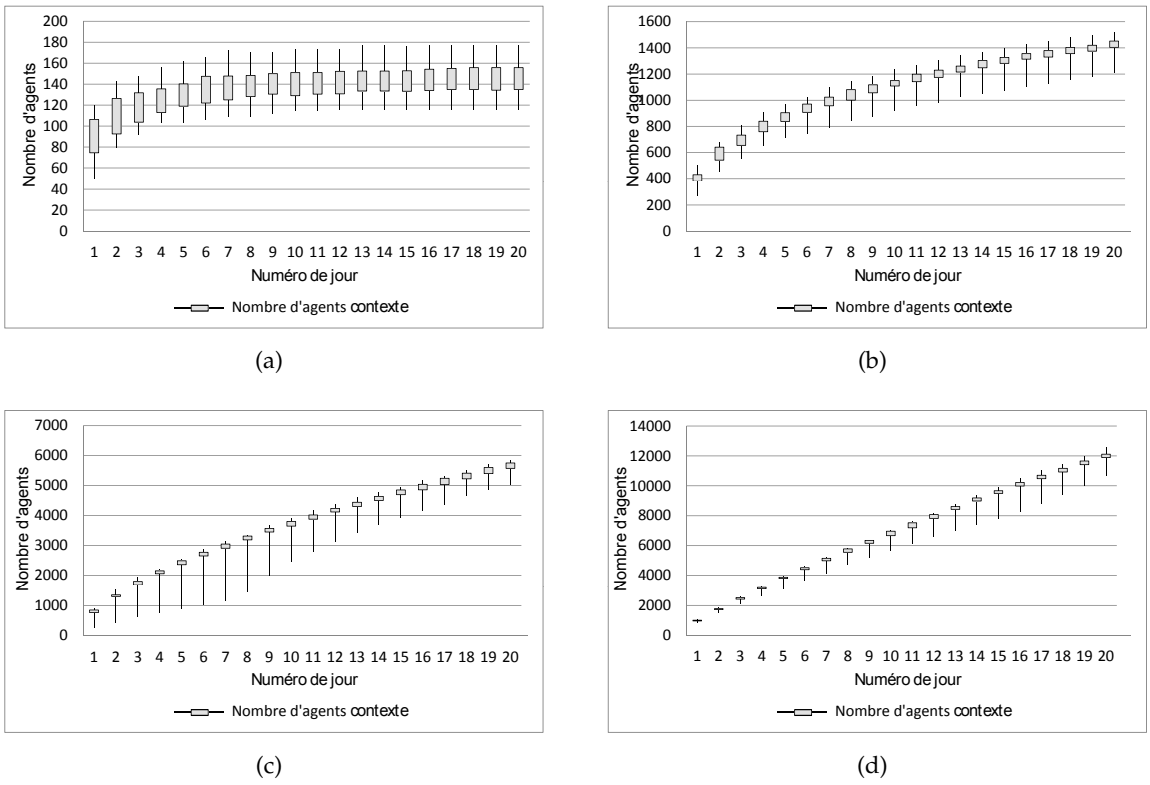

Figure 5.36 – Nombre d'agents *contexte* créés en fonction du nombre de variables inutiles

Nombre de variables inutiles	0	1	2	3
Nombre moyen d'actions de l'utilisateur par simulation	41.7	331.9	1224.3	1636.7
Nombre moyen d'actions d'*Amadeus* par simulation	1652.4	1352.7	441.8	33.2
Taux d'actions de l'utilisateur par simulation	2.5%	19.7%	73.5%	98%
Taux d'actions d'*Amadeus* par simulation	97.5%	80.3%	26.5%	2%
Nombre moyen d'actions de l'utilisateur contredites	4.7%	3.8%	2.1%	2%

Tableau 5.1 – Illustration de la dégradation des performances d'apprentissage d'*Amadeus* en fonction du nombre de variables inutiles

5.6.2 Filtrage de variables aléatoires

Notre objectif est ici d'évaluer la capacité d'un AMAS *dispositif* à filtrer des variables inutiles dont les valeurs évoluent de façon aléatoire. Il s'agit donc de filtrer des variables qui sont, par essence, totalement inutiles.

Cadre d'étude Pour cette étude, nous reprenons la simulation avec un unique utilisateur et un unique effecteur (une lampe). L'AMAS *dispositif* associé à la lampe perçoit des données utiles à son raisonnement : les capteurs de présence et de luminosité situés dans la même pièce que la lampe, les capteurs d'états des portes donnant sur la pièce où se situe la lampe, et enfin l'état de la lampe elle-même. De plus, il perçoit trois variables inutiles générant des données numériques aléatoires comprises entre 0 et 100, et évoluant suivant le même

processus que dans l'étude précédente.

Nous évaluons donc les capacités d'apprentissage d'*Amadeus* en activant les capacités de filtrage des agents *donnée* (voir section 3.2.4.8). Cette évaluation se réalise au travers de 20 simulations de 50 jours.

Résultats A chaque simulation, considérant que 2 actions sont réalisables sur la lampe et qu'il y a 3 variables inutiles, cela représente 6 paires <action,variable> à filtrer ; l'inutilité d'une variable est évaluée en fonction de l'action à réaliser. La figure 5.37 représente le nombre de paires <action,variable> filtrées chaque jour, tandis que le nombre moyen restant de paires <action,variable> à filtrer à la fin de chaque journée est représentée figure 5.38.

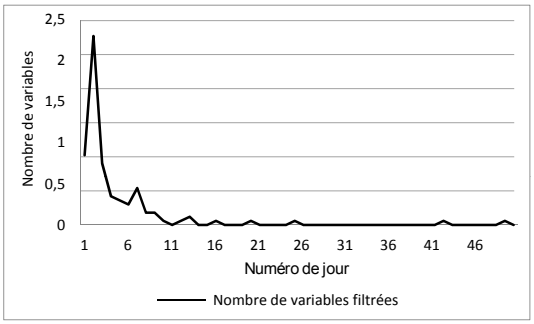

Figure 5.37 – Nombre moyen de paires <action,variable> inutiles filtrées

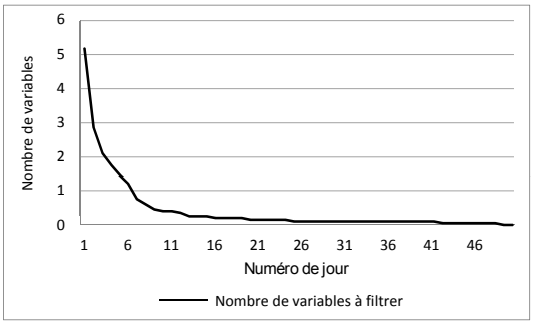

Figure 5.38 – Nombre moyen de paires <action,variable> inutiles à filtrer restantes

Quelle que soit la simulation, la totalité des paires <action,variable> inutiles sont filtrées une fois que la simulation se termine. Plus précisément, 14.2% des paires <action,variable> sont filtrées à la fin du premier jour, puis 52.5% à la fin du second. Nous atteignons le résultat de 90% des paires <action,variable> inutiles filtrées à la fin de la huitième journée. Au final, toutes les paires <action,variable> inutiles sont filtrées à la fin des différentes simulations. De plus, nous observons que sur les 6 paires <action,variable> inutiles, 5 sont filtrées sur la totalité des 20 simulations au bout du seizième jour, tandis que seules 2 simulations sur les 20 réalisées ont encore une dernière paire <action,variable> inutile à filtrer à la fin du vingt-quatrième jour.

La figure 5.39 représente alors le nombre d'actions réalisées sur la lampe par l'utilisateur et par l'AMAS *dispositif* associée à cette lampe, avec le filtrage des variables inutiles d'*Amadeus* activé. Contrairement à ce qui a pu être observé dans l'étude précédente où le filtrage des variables inutiles avait été désactivé, la présence des 3 variables inutiles a ralenti le processus d'apprentissage mais ne l'a pas empêché.

Discussion Nous obtenons pour cette étude un excellent résultat vis-à-vis du filtrage des variables inutiles. Cela est principalement dû au fait que les données de ces variables inutiles varient fortement, et sans aucun lien avec la réalisation des actions de l'utilisateur ou de l'AMAS *dispositif* sur la lampe. Ce type de données est donc facile à filtrer. Nous pouvons à présent nous intéresser à un cas plus complexe, avec des variables inutiles pour certains dispositifs mais pas pour d'autres.

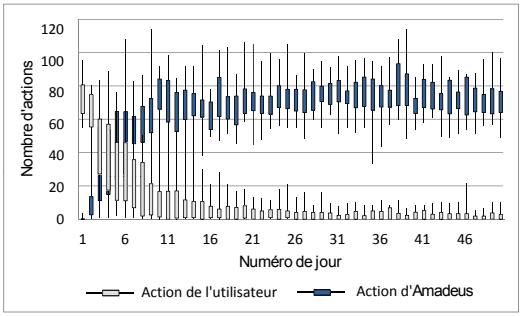

Figure 5.39 – Nombre d'actions de l'utilisateur et d'*Amadeus* sur la lampe en présence de 3 variables inutiles à filtrer

5.6.3 Filtrage de variables inutiles pour différents dispositifs

Pour cette troisième étude pourtant sur le filtrage des variables inutiles, nous cherchons à évaluer la capacité de plusieurs AMAS *dispositif* à filtrer, parmi l'ensemble des données perçues, celles qui sont inutiles à leur processus d'apprentissage. Mais contrairement à l'étude précédente où les variables inutiles possédaient des valeurs aléatoires (elles étaient donc inutiles par essence), il s'agit ici de filtrer des variables qui sont inutiles à certains AMAS *dispositif* et utiles à d'autres. Chaque AMAS *dispositif* se retrouve alors à devoir filtrer les paires <action,variable> qui sont inutiles pour leur traitement, indépendamment du fait que ces paires <action,variable> puissent être potentiellement utiles à d'autres AMAS *dispositif*.

Cadre d'étude Pour cette étude, nous élargissons la simulation de l'étude précédente, en plaçant une lampe et un radiateur dans 4 pièces de l'appartement. Une lampe peut être dans deux états (éteinte ou allumée) tandis que le radiateur peut être dans trois états (éteint, puissance moyenne ou puissance max). De plus, dans chacune de ces pièces, nous plaçons aussi un capteur de présence et un capteur de luminosité. Nous plaçons aussi un capteur de température à l'extérieur de la maison. Enfin, nous plaçons un capteur d'état sur chacune des 6 portes de l'appartement. La topologie de l'appartement simulé est représentée par la figure 5.40, produite grâce à l'interface du simulateur présenté en section 5.1.

Nous intégrons deux utilisateurs dans cette simulation. Ces deux utilisateurs ayant les mêmes préférences en terme de luminosité et de température, il n'est pas nécessaire de placer des capteurs de présence distinguant quel utilisateur est dans la pièce, ni même s'il y en a plusieurs. Tous deux agissent sur la lampe de la même façon que dans les études précédentes. De plus, ils allument le radiateur en fonction de la température extérieure : lorsqu'il fait très froid, ils mettent le radiateur à 2, ils le mettent à 1 s'il fait relativement froid, et le laissent éteint s'il ne fait pas froid.

Nous associons un AMAS *dispositif* à chaque dispositif de l'appartement. Chacun de ces AMAS *dispositif* doit donc apprendre localement quel comportement attribuer à l'effecteur qui lui est associé. Au total, un AMAS *dispositif* associé à une des 4 lampes ou à un des 4 radiateurs perçoit un total de 23 variables :

- l'état de l'effecteur auquel elle est associée ;
- l'état des 7 autres effecteurs ;

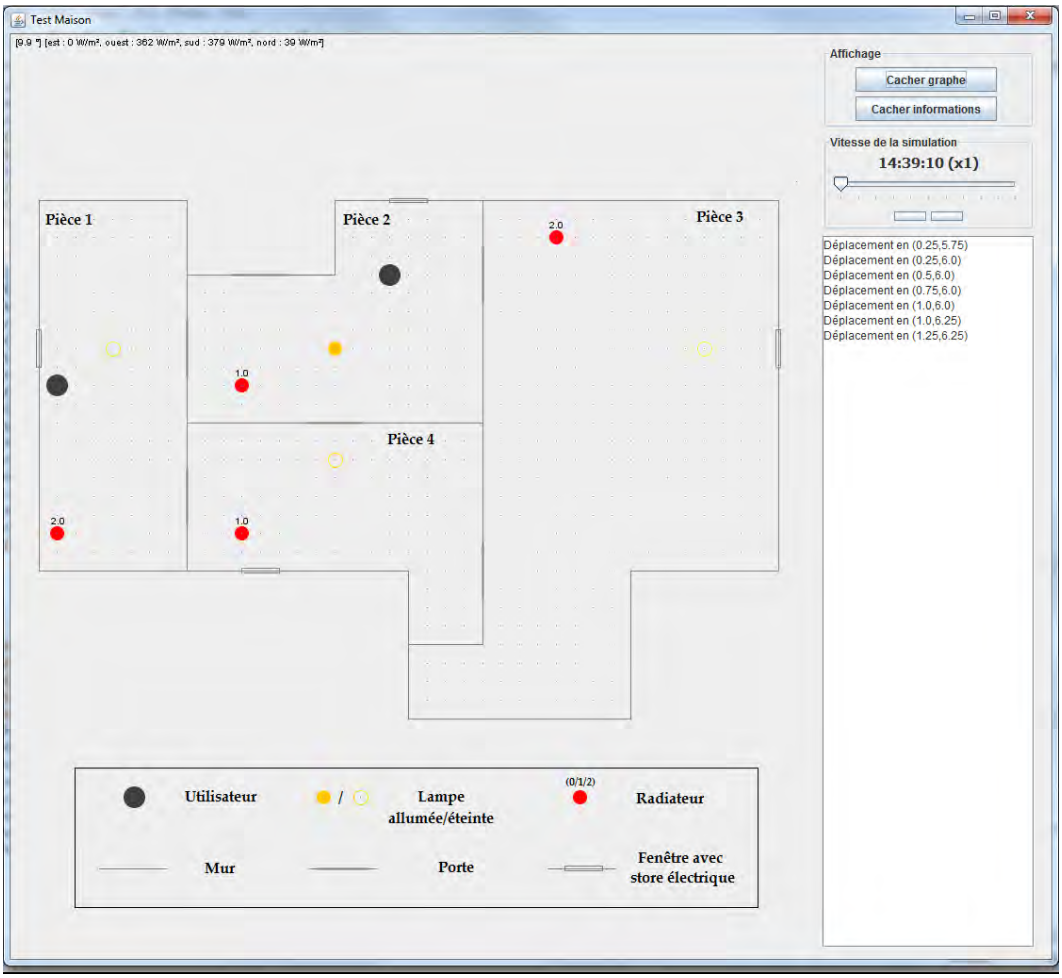

Figure 5.40 – Interface du simulateur

- la luminosité des 4 pièces ;
- la présence d'un utilisateur pour les 4 pièces ;
- la température extérieure ;
- l'état (ouvert ou fermé) pour les 6 portes.

Pour réaliser son apprentissage, chaque AMAS *dispositif* associé à une lampe a besoin de connaître l'état de sa lampe, le niveau de luminosité de sa pièce, la connaissance de la présence d'un utilisateur dans sa pièce, et l'état des portes donnant sur sa pièce. Toutes les autres variables sont inutiles. Sachant qu'il y a deux portes donnant sur la première pièce, 4 sur la seconde, 2 sur la troisième et 3 sur la quatrième, et sachant qu'il n'y a que deux actions réalisables sur une lampe, cela implique donc un total de 10 paires <action,variable> utiles pour la première lampe, 14 pour la seconde, 10 pour la troisième et 12 pour la quatrième.

Un AMAS *dispositif* associé à un radiateur n'a besoin de connaître que l'état de son radiateur et la température extérieure. Sachant qu'un radiateur peut être dans 3 états possibles, cela fait donc un total de 6 paires <action,variable> utiles pour chaque radiateur.

Le tableau 5.2 récapitule le nombre de paires <action,variable> utiles pour chaque dispositif, ainsi que le nombre de paires <action,variable> à filtrer. Au total, cela représente donc 256 paires <action,variable> à filtrer à chaque simulation.

Dispositif	Lampe1	Lampe2	Lampe3	Lampe4
Nombre de paires <action,variable> utiles	10	14	10	12
Nombre de paires <action,variable> à filtrer	26	22	26	24

Dispositif	Radiateur1	Radiateur2	Radiateur3	Radiateur4
Nombre de paires <action,variable> utiles	6	6	6	6
Nombre de paires <action,variable> à filtrer	40	40	40	40

Tableau 5.2 – Nombre de paires <action,variable> utiles et à filtrer en fonction du dispositif

Nous réalisons donc 20 simulations de 50 jours selon 4 configurations :
– Sans *Amadeus* ;
– Avec *Amadeus* mais sans filtrage des variables inutiles ;
– Avec *Amadeus* incluant un filtrage préétablit des variables inutiles ;
– Avec *Amadeus* exploitant la capacité de filtrage de ses agents *donnée*.

Concernant la troisième configuration, nous informons explicitement les agents *donnée* de leur utilité (ou inutilité) en fonction de chaque paire <action,variable>. Ainsi, le filtrage des variables inutiles est déjà réalisé au démarrage de chaque simulation.

Résultats Tout d'abord, la figure 5.41 illustre le nombre d'actions réalisées chaque jour par les deux utilisateurs au travers des 20 simulations en l'absence d'*Amadeus*. Ce nombre varie entre 602 et 835, avec une valeur moyenne de 711.4 actions par jour.

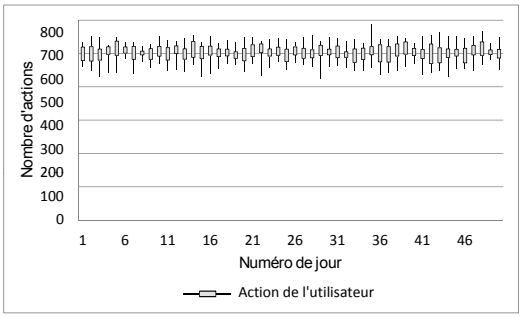

Figure 5.41 – Nombre d'actions des utilisateurs sur l'ensemble des lampes et des radiateurs sans *Amadeus*

A présent, nous relançons les 20 simulations, mais en intégrant un AMAS *dispositif* à chaque dispositif. Cependant, nous désactivons les capacités de filtrage des agents *donnée* (autrement dit, nous empêchons les agents *donnée* de traiter la SNC décrite au chapitre 3.2.4.8). La figure 5.42 représente alors le nombre d'actions réalisées chaque jour par les deux utilisateurs et par *Amadeus* au cours des 20 simulations.

Nous pouvons observer que, bien que le nombre d'actions d'*Amadeus* augmente progressivement au cours du temps, un grand nombre d'actions restent encore réalisées par les utilisateurs à la fin des simulations. Plus précisément, le nombre d'actions réalisées par jour par les utilisateurs passe de 711.4 en moyenne sans *Amadeus* à 163.4 en moyenne avec *Amadeus*, soit une diminution de 77% d'actions des utilisateurs, tandis qu'*Amadeus* réalise en

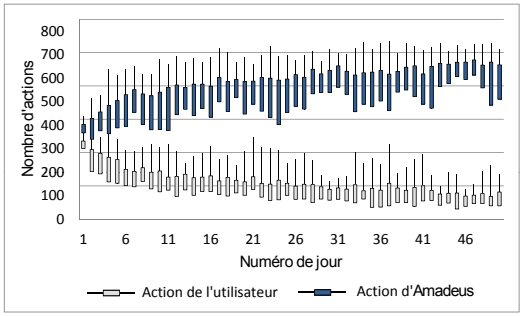

Figure 5.42 – Nombre d'actions des utilisateurs et d'*Amadeus* sur l'ensemble des lampes et des radiateurs, sans filtrage des variables inutiles

moyenne 594.9 actions par jour, ce qui représente 78.5% des actions totales réalisées. Si nous observons les résultats obtenus sur les 10 derniers jours de chaque simulation avec *Amadeus*, nous pouvons voir que les utilisateurs réalisent encore 111.6 actions en moyenne par jour contre 658.9 actions en moyenne par jour réalisées par *Amadeus*. Cela représente 14.5% des actions qui sont encore réalisées par les utilisateurs. Nous pouvons donc considérer que l'apprentissage n'est toujours pas terminé, même après 50 jours de simulations.

A titre de comparaison, nous réalisons les mêmes simulations, mais en intégrant un préfiltrage des données inutiles dans chaque AMAS *dispositif*. Le résultat obtenu est illustré figure 5.43 ; nous pouvons voir le nombre d'actions réalisées par les utilisateurs et par *Amadeus* sur l'ensemble des dispositifs au cours des 20 simulations. Dès lors que seules les variables utiles sont perçues par les différents AMAS *dispositif*, les performances de leur processus d'apprentissage augmentent fortement. Ainsi, le nombre moyen d'actions réalisées par jour avec *Amadeus* est maintenant de 4.7 pour les utilisateurs et 777.1 pour *Amadeus*, ce qui représente une diminution de 99.3% des actions des utilisateurs grâce à *Amadeus*. Sur les 10 derniers jours, ces nombres sont ramenés à 0.9 pour les utilisateurs et 786.2 pour *Amadeus*, soit une diminution de 99.8% des actions des utilisateurs.

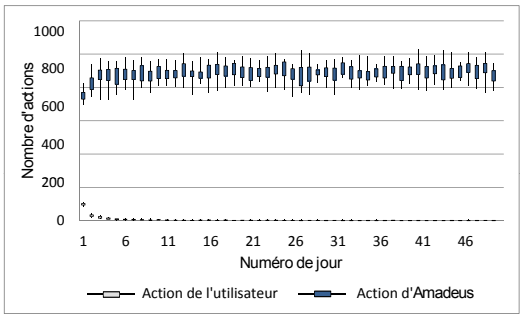

Figure 5.43 – Nombre d'actions des utilisateurs sur l'ensemble des lampes et des radiateurs avec *Amadeus* incluant un préfiltrage des variables inutiles

Enfin, nous réalisons une dernière fois la série des 20 simulations, mais en intégrant les capacités de filtrage des agents *donnée* dans chaque AMAS *dispositif*. Le résultat obtenu est illustré figure 5.44 ; nous pouvons voir le nombre d'actions réalisées par les utilisateurs et par *Amadeus* sur l'ensemble des dispositifs au cours des 20 simulations.

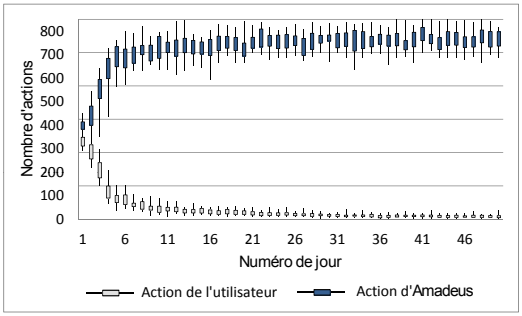

Figure 5.44 – Nombre d'actions des utilisateurs et d'*Amadeus* sur l'ensemble des lampes et des radiateurs, avec filtrage des variables inutiles

Les performances d'*Amadeus* sont fortement améliorées grâce à son filtrage des variables inutiles. En effet, le nombre moyen d'actions réalisées par jour par les utilisateurs passent cette fois de 711.4 sans *Amadeus* à 44.7 avec *Amadeus*, tandis qu'*Amadeus* réalise en moyenne 728.2 actions par jour. Cela représente donc une diminution de 93.7% des actions des utilisateurs en présence d'*Amadeus* doté de capacité de filtrage de variables inutiles. Si nous ne considérons que les 10 derniers jours de chaque simulation, le nombre moyen d'actions par jour des utilisateurs est de 13.9 contre 766.7 pour *Amadeus*, ce qui représente un taux de 98% d'actions réalisées par *Amadeus*. Grâce à son filtrage des variables inutiles, *Amadeus* est donc capable de réaliser son processus d'apprentissage avec des résultats très proches de ceux obtenu avec le préfiltrage.

A présent, nous nous intéressons aux résultats obtenus en terme de variables inutiles filtrées. Initialement, 256 paires <action,variable> doivent être filtrées par l'ensemble des AMAS *dispositif*. Le nombre cumulé de paires <action,variable> filtrées au cours du temps est représenté par la figure 5.45, tandis que la figure 5.46 illustre le nombre de paires <action,variable> filtrées par jour. Nous pouvons observer que sur l'ensemble des simulations, la première journée se termine sans qu'aucun filtrage n'ait encore été réalisé. En revanche, le filtrage commence le second jour avec 8.4 paires <action,variable> filtrées. Ce nombre augmente très fortement le troisième jour, avec 35.5 paires <action,variable> filtrées ce jour là, soit un total 43.9 paires <action,variable> filtrées à la fin du troisième jour. Au final, nous obtenons un nombre moyen de 98.4 paires <action,variable> filtrées à la fin du 10ème jour, puis finalement un nombre moyen de 183.3 paires <action,variable> filtrées à la fin des 50 jours de simulations. Cela représente donc un filtrage de 38.4% des paires <action,variable> inutiles en moyenne à la fin des 10 premiers jours, et un filtrage de 71.6% des paires <action,variable> inutiles en moyenne à la fin des simulations. Enfin, l'ensemble des paires <action,variable> filtrées sont des paires <action,variable> inutiles, aucune paire <action,variable> utile à un AMAS *dispositif* n'a été filtré à tort.

Discussion Cette étude a permis de montrer la forte amélioration des capacités d'apprentissage d'*Amadeus* qu'apporte le filtrage des variables inutiles, avec en moyenne 71.6% des paires <action,variable> filtrées à la fin des différentes simulations. Certes, il reste encore en moyenne 28.4% des variables inutiles qui n'ont toujours pas été filtrées, mais nous pouvons observer que l'important filtrage déjà réalisé a été suffisant pour permettre à *Amadeus*

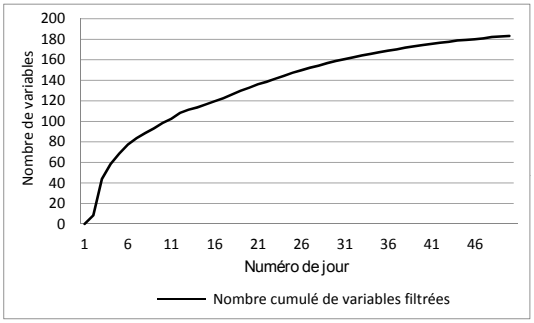

Figure 5.45 – Nombre cumulé de paires <action,variable> filtrées par jour

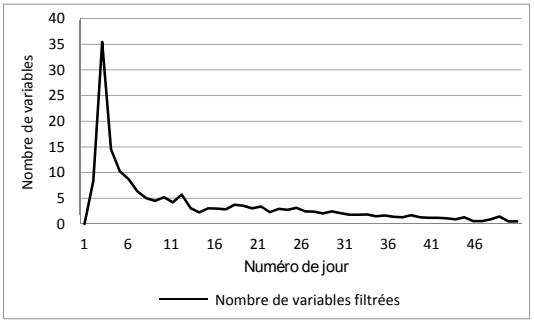

Figure 5.46 – Nombre de paires <action,variable> filtrées par jour

de réaliser efficacement son processus d'apprentissage. De plus, bien qu'il reste encore des paires <action,variable> à filtrer, l'ensemble des paires <action,variable> filtrées étaient bien des paires <action,variable> inutiles.

En diminuant la valeur du seuil à partir duquel un agent *donnée* considère qu'il est inutile pour la réalisation d'une action par un des agents *contrôleur* de son voisinage, il est possible d'augmenter davantage le taux de paires <action,variable> filtrées. Cependant, cela peut entraîner une augmentation du taux d'erreurs, autrement dit du nombre de paires <action,variable> utiles filtrées à tort. Or, considérant qu'en l'état, le filtrage réalisé par *Amadeus* est suffisant pour qu'il puisse réaliser son apprentissage, il vaut mieux se contenter de son taux actuel de paires <action,variable> filtrées, plutôt que de risquer un filtrage incorrect d'une paire <action,variable> utile.

Conclusion

5.7 Bilan

Ce travail de thèse s'inscrit dans le domaine des systèmes ambiants, et a mené à la conception du système multi-agent adaptatif *Amadeus*. Plus précisément, il s'agit d'un AMAS composé lui-même d'AMAS *dispositif* : un AMAS *dispositif* est associé à chaque dispositif présent dans le système ambiant, et l'ensemble des AMAS *dispositif* forme l'AMAS *Amadeus* dont les agents se transmettent des informations utiles à leurs fonctionnements respectifs. Chaque AMAS *dispositif* a pour objectif local de déterminer le comportement à attribuer à son dispositif. Pour cela, il observe les actions des utilisateurs, et apprend dans quelle situation il est possible de les réaliser à leur place.

Pour qu'un système soit capable d'adapter le comportement d'un système ambiant en fonction du contexte de ses utilisateurs, nous avons dressé une liste de propriétés qu'il doit posséder ; il s'agit de la généricité, de l'expressivité, de la distribution, de l'ouverture, de la proactivité, de la confidentialité et de l'explicabilité.

La mise en oeuvre d'*Amadeus* peut se faire en tant qu'application fonctionnant sur un gestionnaire de contexte ; *Amadeus* accèderait alors aux données en passant par le gestionnaire de contexte, qui prendrait en charge les échanges de données entre les AMAS *dispositif*. Dès lors, nous pouvons lister les propriétés possédées par ces deux systèmes complémentaires :

– **Généricité** et **Expressivité** : La généricité et l'expressivité sont deux propriétés généralement antinomiques. En effet, une donnée très expressive nécessite un format trop complexe (sous forme d'ontologie par exemple) pour être générique, tandis qu'une donnée complètement générique est plutôt sémantiquement pauvre. Dans notre cas, nous avons conçu *Amadeus* pour fonctionner en exploitant les données sous leur forme la plus générique possible, à savoir directement avec leur valeur numérique, ce qui lui permet de prendre en compte la majorité des données. Cependant, rien n'empêche que ces données soient malgré tout enrichies d'une sémantique qui permettrait éventuellement de réaliser des prétraitements (un filtrage supplémentaire des variables inutiles basées sur la sémantique des données, par exemple). *Amadeus* est conçu pour fonctionner même en l'absence de toute sémantique associée aux données. *Amadeus* est donc doté de la propriété de généricité, tout en étant capable de fonctionner sur un gestionnaire de contexte doté de la propriété d'expressivité ;

- **Proactivité** : La proactivité de notre système est basée sur sa capacité à observer les actions des utilisateurs afin de déterminer quelle action réaliser face aux différentes situations qui se présentent. Un AMAS *dispositif* observe donc les actions que réalisent les utilisateurs sur le dispositif qui lui est associé, et apprend à réaliser ces actions à la place de l'utilisateur. Le chapitre 5.2 a permis de mettre en avant les capacités d'apprentissage d'*Amadeus* dans un cas ne comportant qu'un utilisateur ayant des préférences stables. Le chapitre 5.4 a ensuite permis d'observer sa capacité à adapter son apprentissage en cas de changement dans les préférences de cet utilisateur. Enfin, dans le chapitre 5.5, nous avons pu montrer sa capacité à réaliser son apprentissage même avec plusieurs utilisateurs, l'un d'eux ayant changé ses préférences en cours de fonctionnement. Notre système est donc capable d'apprendre un comportement à attribuer aux différents dispositifs d'un système ambiant afin d'agir de façon proactive sur ses dispositifs ;
- **Ouverture** : Si nous considérons l'utilisation d'un gestionnaire de contexte ouvert - comme c'est le cas de la plupart d'entre eux), l'application fonctionnant sur ce gestionnaire de contexte doit être capable d'exploiter cette propriété. Les évaluations du chapitre 5.3 ont permis de confirmer que c'était le cas d'*Amadeus*, chaque AMAS *dispositif* étant capable de continuer son processus d'apprentissage et de contrôle de son dispositif en s'adaptant progressivement à l'apparition et/ou à la disparition d'autres dispositifs ;
- **Distribution** : La transmission des données entre les différentes AMAS *dispositif* est bien sûr à la charge du gestionnaire de contexte. Mais utiliser le gestionnaire de contexte comme simple *middleware* n'est pas suffisant pour assurer la propriété de distribution, car les échanges de données entre AMAS *dispositif* doivent être pertinents. Le chapitre 5.6 a permis de montrer les capacités d'*Amadeus* à filtrer les variables inutiles de son processus d'apprentissage, afin que progressivement les échanges entre AMAS *dispositif* se limitent aux données utiles ;
- **Confidentialité** : La confidentialité est une propriété que doit posséder un gestionnaire de contexte. Cette propriété est notamment étudiée par [Oglaza *et al.*, 2013] afin d'être intégrée au gestionnaire de contexte du projet Income. Nous considérons que la localité de l'apprentissage réalisé par *Amadeus* est un premier pas vers cette propriété. En effet, les traitements concernant chaque dispositif peuvent être réalisés localement à ce dispositif sans avoir besoin d'être centralisés sur un unique dispositif, ce qui facilite le contrôle des utilisateurs sur la diffusion de leurs informations ;
- **Explicabilité** : Nous avons considéré que s'il est nécessaire qu'un utilisateur puisse comprendre les raisons qui ont poussé le système à agir, cela ne signifie pas qu'il ait besoin de comprendre l'intégralité du processus, ni que cette justification doive être fournie en continu. Dans sa version actuelle, *Amadeus* ne propose pas de mécanisme permettant d'expliquer ses actions. En revanche, l'intégration d'un tel mécanisme pourrait être envisagée, en ajoutant aux agents *contexte* une mémoire leur permettant d'enregistrer un historique des situations dans lesquelles ils ont été créés puis sélectionnés, afin de présenter aux utilisateurs une justification basée sur leurs actions et leurs acceptions des actions d'*Amadeus* passées.

Amadeus possède la majorité des propriétés nécessaires pour adapter un système am-

biant au contexte de ses utilisateurs que nous avions listées. Il vérifie en particulier la généricité vis-à-vis du type de données qu'il peut prendre en compte, l'ouverture lui permettant de gérer l'apparition et la disparition de dispositifs en cours de fonctionnement, la distribution pertinente des traitements et des données, et enfin la proactivité qui lui permet d'attribuer un comportement à un système ambiant sans connaissance *a priori*. Les propriétés d'expressivité et de confidentialité ne sont pas intégrées au fonctionnement d'*Amadeus*, mais son fonctionnement n'empêche pas pour autant leur mise en place.

5.8 Perspectives

Le développement d'Amadeus et ses expérimentations nous ont permis de définir les orientations de travaux futurs, tant théoriques que pratiques.

5.8.1 Résolution de SNC supplémentaires

Bien que la conception d'un AMAS par la méthode ADELFE aide le concepteur à déterminer un certain nombre de SNC, il est très rare que la totalité des SNC que peuvent rencontrer les agents soit spécifiée lors de la première phase de conception. Dans le cadre d'*Amadeus*, certaines SNC ont été trouvées par observation du comportement obtenu lors d'évaluations intermédiaires. Pour améliorer notre système, il est donc nécessaire de rechercher quelles sont les SNC encore non résolues, puis de les étudier afin de les résoudre.

Nous avons d'ores et déjà déterminé certaines SNC à résoudre pour apporter quelques améliorations à notre système à court terme. Par exemple, s'il nous semble que la majorité des données perceptibles peuvent être modélisées et utilisées sous la forme de valeurs numériques, il existe des données pour lesquelles ce format n'est pas approprié. Ainsi, un agent *capteur*, percevant des données pour lesquelles le format sous forme de valeurs numériques n'est pas approprié (les données énumérées par exemple), sera face à une SNC d'incompréhension. La résolution de cette SNC impliquerait donc d'étendre les formats de données perceptibles (par les agents *capteur* notamment) tout en restant le plus générique possible.

A plus long terme, il sera nécessaire de résoudre la SNC suite à l'absence d'une variable utile parmi les variables perçues par un AMAS *dispositif*. Il s'agit d'une SNC d'improductivité pour les agents *contexte* qui, considérant les données perçues, ne sont pas capables de fournir les propositions d'actions correctes à leur agent *contrôleur*. Une première voie à exploiter pour détecter cette SNC consiste à se baser sur un trop grand nombre d'autocontradictions, qui montrerait l'impossibilité d'un AMAS *dispositif* à converger vers un comportement correct à cause d'au moins une variable manquante. La résolution de cette SNC ne se limite pas à *Amadeus* ; ce problème de détection de données perçues manquantes se retrouve dans l'instanciation de l'approche par AMAS à plusieurs autres problèmes (voir par exemple les travaux de [Brax, 2013]).

5.8.2 Amélioration du filtrage des variables inutiles

Le processus d'apprentissage d'*Amadeus* intègre un mécanisme de filtrage des variables inutiles. L'intégration d'un tel mécanisme nous semble primordiale au vu du grand nombre de variables présentes dans un système ambiant. Cependant, il nous semble présenter des limitations dans son état actuel : il permet bien de filtrer la majorité des variables inutiles dans les expérimentations que nous avons réalisées, mais une mise en application dans un système ambiant réel peut amener un AMAS *dispositif* à percevoir un nombre très largement supérieur de variables inutiles à celui que nous avons étudié dans nos expérimentations (allant jusqu'à dépasser les milliers de variables perceptibles). Un tel traitement de filtrage nous semble encore au-delà des capacités actuelles de filtrage d'*Amadeus*.

Nous avons montré au chapitre 2.3 que pour réaliser un filtrage des variables inutiles, il existe généralement deux possibilités : partir d'un ensemble vide de variables puis étendre cet ensemble en intégrant progressivement des variables utiles, ou partir de l'ensemble des variables perçues puis réduire cet ensemble en supprimant progressivement des variables inutiles (cette dernière solution étant la solution actuellement employée dans *Amadeus*). Outre ces deux possibilités, il nous semble pertinent d'en envisager une troisième, située à mi-chemin entre ces deux solutions : partir d'un sous-ensemble des variables perçues, puis d'une part éliminer les variables inutiles de cet ensemble, et d'autre part intégrer les variables utiles extérieures à cet ensemble.

Concernant l'initialisation de ce sous-ensemble des variables perçues, plutôt que de partir d'un sous-ensemble sélectionné au hasard, il nous semble intéressant d'envisager l'utilisation d'un système complémentaire pour améliorer le fonctionnement d'*Amadeus* par un prétraitement des variables en utilisant la sémantique associée à ces variables. En particulier, nous nous intéressons au travail réalisé par [Olaru, 2011], qui propose de structurer des informations contextuelles grâce à une architecture multi-agent. Ce système permet de considérer cinq types de données contextuelles : les données spatiales, les données computationnelles, les données temporelles, les données d'activités et les données sociales. Il permet ensuite de hiérarchiser les différentes données grâce à différents types de relations, telles que "*est-dans*" pour les données spatiales, ou "*fait-partie-de*" pour les données d'activités. Ce système permet d'exploiter la sémantique des données contextuelles pour relier certaines données entre elles, et ainsi établir la notion de proximité entre deux données. Cette proximité n'est pas uniquement spatiale, elle porte aussi sur les autres aspects gérés par ce système (proximité temporelle, proximité sociale, etc.).

L'utilisation d'un tel système pour établir la proximité entre différentes données contextuelles impliquerait donc une intervention du concepteur afin d'apporter quelques connaissances portant sur la proximité entre les variables perçues par les différents AMAS *dispositif*. Cependant, ce système permettrait d'améliorer la présélection des variables apparemment utiles au fonctionnement d'un AMAS *dispositif*. Il sélectionnerait alors les variables qui lui semblent proches de celles à contrôler (l'état de l'effecteur) afin de fournir à l'AMAS *dispositif* un sous-ensemble de variables perçues pertinentes. Puis par la suite, l'AMAS *dispositif* pourra filtrer les variables inutiles dans cet ensemble, ou bien demander des variables supplémentaires à mesure qu'il détecterait une insuffisance dans cet ensemble.

5.8.3 Évaluations supplémentaires

La conception d'un simulateur de système ambiant nous a permis de réaliser facilement un grand nombre d'évaluations tout au long de la conception d'*Amadeus*. Cette phase d'évaluation grâce à un simulateur de système ambiant est nécessaire, afin d'évaluer les capacités d'*Amadeus* en cours de conception, jusqu'à ce qu'il soit capable d'être appliqué dans un système réel. Des possibilités d'améliorations de ce simulateur sont actuellement à l'étude, afin de faciliter son utilisation et d'améliorer la complexité et le réalisme des simulations dans le but d'agrandir notablement le corpus des évaluations d'*Amadeus*.

Afin d'améliorer cette phase d'évaluation, une des perspectives de notre travail consiste à intégrer à *Amadeus* la capacité d'expliquer ses actions, par exemple en transformant les connaissances de l'ensemble des agents *contexte* en une base de règles compréhensibles par les utilisateurs. Cette propriété permettrait de compléter les évaluations réalisées sur *Amadeus*. Cependant, la principale perspective à long terme concerne une mise en application d'*Amadeus* dans un système ambiant réel impliquant de vrais utilisateurs. Cette mise en application entraînerait certains besoins : il faudrait disposer d'un environnement doté de capteurs et d'effecteurs, ainsi que d'utilisateurs volontaires pour évoluer dans cet environnement durant une période de temps relativement longue. Une telle évaluation permettrait sans aucun doute de pouvoir établir clairement les capacités, ainsi que l'acceptabilité, de notre système.

Bibliographie

Agnar AAMODT et Enric PLAZA : Case-based reasoning : Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59, 1994.

Jean-Paul ARCANGELI, Victor NOEL et Frédéric MIGEON : Architectures logicielles et systèmes multi-agents. *Dans* Mourad OUSSALAH, éditeur : *Architectures logicielles : Principes, techniques et outils*, chapitre 10. Hermès Science, 2013.

Seung-Ho BAEK, Eun-Chang CHOI et Jae-Doo HUH : Design of information management model for sensor based context-aware service in ubiquitous home. *Dans International Conference on Convergence Information Technology*, pages 1040–1047. IEEE, 2007.

Gérald BAILLARGEON : *Introduction à l'inférence statistique : méthodes d'échantillonnage, estimation, tests d'hypothèses, corrélation linéaire, droite de régression et test du khi-deux avec applications diverses*. Techniques statistiques. Les editions smg édition, 1982.

Louise BARKHUUS : Context information vs. sensor information : A model for categorizing context in context-aware mobile computing. *Simulation series*, 35(1):127–133, 2003.

M Gonzalez BEDIA et Juan Manuel CORCHADO : A planning strategy based on variational calculus for deliberative agents. *Computing and information systems*, 9(1):2–13, 2002.

Abdel BELAÏD et Yolande BELAÏD : *Reconnaissance des formes : méthodes et applications*. InterEditions, 1992.

Carole BERNON, Valérie CAMPS, Marie-Pierre GLEIZES et Gauthier PICARD : Engineering adaptive multi-agent systems : The adelfe methodology. *Agent-Oriented Methodologies*, pages 172–202, 2005.

Jérémy BOES, Frédéric MIGEON et François GATTO : Self-organizing agents for an adaptive control of heat engines. *Dans* Jean-Louis FERRIER, Oleg Yu. GUSIKHIN, Kurosh MADANI et Jurek Z. SASIADEK, éditeurs : *Proceedings of the 10th International Conference on Informatics in Control, Automation and Robotics*, volume 1, pages 243–250. SciTePress, 2013.

Noelie BONJEAN, Carole BERNON et Pierre GLIZE : Engineering development of agents using the cooperative behaviour of their components. *Dans* Giancarlo FORTINO, Massimo COSSENTINO, Marie-Pierre GLEIZES et Juan PAVON, éditeurs : *Proceedings of the Second Multi-Agent Logics, Languages, and Organisations Federated Workshops*, volume 494 de *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

Noelie BONJEAN, Wafa MEFTEH, Marie-Pierre GLEIZES, Christine MAUREL et Frédéric MIGEON : Adelfe 2.0. *Dans* Massimo COSSENTINO, Vincent HILAIRE, Ambra MOLESINI et Valeria SEIDITA, éditeurs : *Handbook on Agent-Oriented Design Processes*. Springer, 2013.

Nicolas BRAX : *Self-Adaptive Multi-Agent Systems for Aided Decision-Making : An application to Maritime Surveillance*. Thèse de doctorat, Université Paul Sabatier, 2013.

Paolo BRESCIANI, Anna PERINI, Paolo GIORGINI, Fausto GIUNCHIGLIA et John MYLOPOULOS : Tropos : An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

Patrick BREZILLON et Jean-Charles POMEROL : Contextual knowledge sharing and cooperation in intelligent assistant systems. *Le travail humain*, 62:223–246, 1999. ISSN 0041-1868.

Peter BROWN : The stick-e document : a framework for creating context-aware applications. *Electronic Publishing*, 8:259–272, 1996. ISSN 0894-3982.

Patrick BRÉZILLON : Modélisation et management des contextes. Rapport technique Mci, Laboratoire d'Informatique de Paris 6, 2010.

Valérie CAMPS : *Vers une théorie de l'auto-organisation dans les systèmes multi-agents baséee sur la coopération : application à la recherche d'information dans un système d'information répartie*. Thèse de doctorat, Université Paul Sabatier, 1998.

Davy CAPERA : *Systèmes multi-agents adaptatifs pour la résolution de problèmes : Application à la conception de mécanismes*. Thèse de doctorat, Université Paul Sabatier, 2005.

Tarak CHAARI, Frédérique LAFOREST et André FLORY : Adaptation des applications au contexte en utilisant les services web. *Dans Proceedings of the 2nd French-speaking conference on Mobility and ubiquity computing*. ACM Press, 2005. ISBN 1595931724.

Harry CHEN, Tim FININ et Amupam JOSHI : Semantic web in the context broker architecture. Rapport technique, DTIC Document, 2005.

Denis CONAN, Romain ROUVOY et Lionel SEINTURIER : Scalable processing of context information with cosmos. *Dans Distributed Applications and Interoperable Systems*, pages 210–224. Springer, 2007.

Diane J COOK, Michael YOUNGBLOOD, Edwin O HEIERMAN III, Karthik GOPALRATNAM, Sira RAO, Andrey LITVIN et Farhan KHAWAJA : Mavhome : An agent-based smart home. *Dans Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 521–524. IEEE Computer Society, 2003.

Antoine CORNUÉJOLS et Laurent MICLET : *Apprentissage artificiel*. Eyrolles, 2011.

Corinna CORTES et Vladimir VAPNIK : Support vector machine. *Machine learning*, 20(3):273–297, 1995.

Massimo COSSENTINO, Giancarlo FORTINO, Alfredo GARRO et Samuele MASCILLARO : Passim : a simulation-based process for the development of multi-agent systems. *International Journal of Agent-Oriented Software Engineering*, 2(2):132–170, 2008.

Joelle COUTAZ, James L CROWLEY, Simon DOBSON et David GARLAN : Context is key. *Communications of the ACM*, 48(3):49–53, 2005.

Michael J COVINGTON, Wende LONG, Srividhya SRINIVASAN, Anind K DEV, Mustaque AHAMAD et Gregory D ABOWD : Securing context-aware applications using environment roles. *Dans Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 10–20. ACM, 2001.

Tom DE WOLF et Tom HOLVOET : Emergence versus self-organisation : Different concepts but promising when combined. *Dans Engineering self-organising systems*, pages 1–15. Springer, 2005.

Olivier DELALLEAU : *Extraction hiérarchique de caractéristiques pour l'apprentissage à partir de données complexes en haute dimension*. Rapport pré-doctoral, University of Montreal, 2008.

Anind K DEY et Gregory ABOWD : Towards a better understanding of context and context-awareness. *Dans CHI 2000 Workshop on The What, Who, Where, When, and How of Context-Awareness*, pages 304–307. Springer, 2000.

Anind K DEY, Gregory D ABOWD et Daniel SALBER : A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human–Computer Interaction*, 16(2-4):97–166, 2001.

Richard O DUDA, Peter E HART *et al.* : *Pattern classification and scene analysis*, volume 3. Wiley New York, 1973.

Tony DUJARDIN, J ROUILLARD, Jean-Christophe ROUTIER, Jean-Claude TARBY *et al.* : Gestion intelligente d'un contexte domotique par un système multi-agents. *Dans* Emmanuel ADAM et Jean-Paul SANSONNET, éditeurs : *Journées Francophones sur les Systèmes Multi-Agents*, pages 137–146. Cepadues Editions, 2011.

Jérome. EUZENAT, Jérome. PIERSON et Fano RAMPARANY : Dynamic context management for pervasive applications. *The Knowledge Engineering Review*, 23(01):21–49, 2008.

Jacques FERBER : *Les sysèmes multi-agents : vers une intelligence collective*, volume 16. Inter-Editions Paris, 1995.

David FRANKLIN et Joshua FLASCHBART : All gadget and no representation makes jack a dull environment. *Dans Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments*, pages 155–160, 1998.

Olivier FRANÇOIS et Philippe LERAY : Etude comparative d'algorithmes d'apprentissage de structure dans les réseaux bayésiens. *Dans Rencontres des Jeunes Chercheurs en IA*, pages 167–180, 2003.

Jean-Pierre GEORGÉ : L'émergence. Rapport technique, Institut de Recherche en Informatique de Toulouse, 2003.

Jean-Pierre GEORGÉ, Marie-Pierre GLEIZES et Valérie CAMPS : Cooperation. *Dans Self-organising Software*, pages 193–226. Springer, 2011.

Jeffrey GOLDSTEIN : Emergence as a construct : History and issues. *Emergence*, 1(1):49–72, 1999.

Christos GOUMOPOULOS, Achilles KAMEAS, Hani HAGRAS, Victor CALLAGHAN, Michael GARDNER, Wolfgang MINKER, Michael WEBER, Yacine BELLIK et Apostolos MELIONES : Atraco : Adaptive and trusted ambient ecologies. *Dans Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 96–101. IEEE, 2008.

Priscilla E GREENWOOD et Michael S NIKULIN : *A Guide to Chi-Squared Testing*. Wiley, 1996.

Richard Moe GUSTAVSEN : Condor–an application framework for mobility-based context-aware applications. *Dans Proceedings of the workshop on concepts and models for ubiquitous computing*, volume 39, 2002.

Isabelle GUYON et André ELISSEEFF : An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

Hani HAGRAS, Victor CALLAGHAN, Martin COLLEY, Graham CLARKE, Anthony POUNDS-CORNISH et Hakan DUMAN : Creating an ambient-intelligence environment using embedded agents. *Intelligent Systems, IEEE*, 19(6):12–20, 2004.

Thomas HOFER, Wieland SCHWINGER, Mario PICHLER, Gerhard LEONHARTSBERGER, Josef ALTMANN et Werner RETSCHITZEGGER : Context-awareness on mobile devices-the hydrogen approach. *Dans System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 10–19. IEEE, 2003.

John H HOLLAND : *Adaptation in natural and artificial systems : An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

John J HOPFIELD : Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

Finn V JENSEN : *An introduction to Bayesian networks*, volume 210. University College London Press, 1996.

Elsy KADDOUM : *Optimisation sous contraintes de problèmes distribués par auto-organisation coopérative*. Thèse de doctorat, Université de Toulouse, 2011.

Lalana KAGAL, Tim FININ et Anupam JOSHI : A policy language for a pervasive computing environment. *Dans 4th International Workshop on Policies for Distributed Systems and Networks*, pages 63–74. IEEE, 2003.

Kenji KIRA et Larry A RENDELL : The feature selection problem : Traditional methods and a new algorithm. *Dans Proceedings of the tenth national conference on Artificial intelligence*, pages 129–134. AAAI Press / The MIT Press, 1992.

Boicho KOKINOV : A dynamic approach to context modeling. *Dans Proceedings of the IJCAI-95 workshop on modeling context in knowledge representation and reasoning*, volume 95, pages 199–209, 1995.

Janet L KOLODNER : An introduction to case-based reasoning. *Artificial Intelligence Review*, 6(1):3–34, 1992.

Mari KORKEA-AHO : Context-aware applications survey, 2000. URL `http://www.cse.tkk.fi/fi/opinnot/T-110.5190/2000/applications/context-aware.html`.

Pat LANGLEY et Stephanie SAGE : *Oblivious Decision Trees and Abstract Cases*. AAAI Press, 1994.

Sylvain LEMOUZY : *Systèmes interactifs auto-adaptatifs par systèmes multi-agents auto-organisateurs : application à la personnalisation de l'accès à l'information*. Thèse de doctorat, Université de Toulouse, Toulouse, France, juillet 2011.

George Henry LEWES : *Problems of life and mind*, volume 2. London : Trubner and co., 1877.

Henry LIEBERMAN et Ted SELKER : Out of context : computer systems that adapt to, and learn from, context. *IBM Systems Journal*, 39(3.4):617–632, 2000.

Deborah L MCGUINNESS, Frank VAN HARMELEN *et al.* : Owl web ontology language overview. *W3C recommendation*, 10(2004-03):10, 2004.

Wolfgang MINKER, Tobias HEINROTH, Achilles KAMEAS, Hani HAGRAS, Joy van HELVERT, Yacine BELLIK, Christos GOUMOPOULOS et Apostolos MELIONES : D05 - project final rwwwweport. Public deliverable, The ATRACO Project (FP7/2007-2013 grant agreement numéro 216837), 2010.

Michael C MOZER, Robert H DODIER, Marc ANDERSON, Lucky VIDMAR, RF CRUICKSHANK et Debra MILLER : The neural network house : an overview. *Dans Current Trends in Connectionism : Proceedings of the Swedish Conference on Connectionism*, pages 371–380. Erlbaum Hillsdale, NJ, 1995.

Michael C MOZER et Debra MILLER : Parsing the stream of time : The value of event-based segmentation in a complex real-world control problem. *Dans Adaptive Processing of Sequences and Data Structures*, pages 370–388. Springer, 1998.

Victor NOEL : *Component-based Software Architectures and Multi-Agent Systems : Mutual and Complementary Contributions for Supporting Software Development*. Thèse de doctorat, Université de Toulouse, 2012.

Victor NOEL, Jean-Paul ARCANGELI et Marie-Pierre GLEIZES : Une approche architecturale à base de composants pour l'implémentation des systèmes multi-agents. *Revue des Nouvelles Technologies de l'Information, Avancées récentes dans le domaine des Architectures Logicielles*, RNTI-L-6:1–26, 2012.

Arnaud OGLAZA, Romain LABORDE et Pascale ZARATÉ : Authorization policies : Using decision support system for context-aware protection of user's private data. *Dans International Symposium on UbiSafe Computing*. IEEE, 2013.

Andrei OLARU : *Un système multi-agent sensible au contexte pour les environnements d'intelligence ambiante*. Thèse de doctorat, Université Pierre et Marie Curie, 2011.

Jason PASCOE : Adding generic contextual capabilities to wearable computers. *Dans Second International Symposium on Wearable Computers*, volume 44, pages 92–99. IEEE Computer Society, 1998.

Juan PAVÓN : Ingenias : Développement dirigé par modèles des systèmes multi-agents. *Habilitationa diriger des recherches de l'Université Pierre et Marie Curie*, 2006.

Gauthier PICARD : *Méthodologie de développement de systèmes multi-agents adaptatifs et conception de logiciels à fonctionnalité émergente*. Thèse de doctorat, Université Paul Sabatier, 2004.

Gauthier. PICARD et Pierre GLIZE : Modélisation et expérimentations d'une décision locale basée sur l'auto-organisation coopérative. *Dans Journées Francophones sur les Systèmes Multi-Agents*, pages 161–174. Hermès-Lavoisier, 2005.

Gaetan PRUVOST : *Modélisation et conception d'une plateforme pour l'interaction multimodale distribuée en intelligence ambiante*. Thèse de doctorat, Université Paris Sud-Paris XI, 2013.

John Ross QUINLAN : *C4. 5 : programs for machine learning*, volume 1. Morgan kaufmann, 1993.

Gaetan REY : *Contexte en Interaction Homme-Machine : le contexteur.* Thèse de doctorat, Université Joseph Fourier, 2005.

Eli ROHN : Predicting context aware computing performance. *Ubiquity*, pages 1–17, 2003.

Daniel ROMERO, Romain ROUVOY, Lionel SEINTURIER, Sophie CHABRIDON, Denis CONAN et Nicolas PESSEMIER : Enabling context-aware web services : a middleware approach for ubiquitous environments. *Enabling Context-Aware Web Services : Methods, Architectures, and Technologies*, pages 113–135, 2010.

Frank ROSENBLATT : The perceptron : a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Sam ROTTENBERG, Sebastien LERICHE, Claire LECOCQ, Chantal TACONET *et al.* : Vers une définition d'un système réparti multi-échelle. *Dans 8èmes journées francophones Mobilité et Ubiquité*, pages 178–183, 2012.

Sylvain ROUGEMAILLE : *Ingénierie des systèmes multi-agents adaptatifs dirigée par les modèles*. Thèse de doctorat, Université Paul Sabatier, 2008.

Stuart Jonathan RUSSELL et Peter NORVIG : *Artificial intelligence : a modern approach, 3rd edition*. Prentice Hall, Person Education Inc., 1995.

Nick S. RYAN, Jason PASCOE et David R. MORSE : Enhanced reality fieldwork : the context-aware archaeological assistant. *Dans Computer applications in archaeology*, pages 1–9. Tempus Reparatum, 1998.

Bill N.. SCHILIT, Norman ADAMS et Roy WANT : Context-aware computing applications. *Dans First Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Comput. Soc. Press, 1995.

Bill N. SCHILIT et Marvin M. THEIMER : Disseminating active map information to mobile hosts. *Network, IEEE*, 8(5):22–32, 1994. ISSN 0890-8044.

Albrecht SCHMIDT, Michael BEIGL et Hans-W GELLERSEN : There is more to context than location. *Computers & Graphics*, 23(6):893–901, 1999.

Zied SELLAMI : *Gestion dynamique d'ontologies à partir de textes par systèmes multi-agents adaptatifs*. Thèse de doctorat, Université de Toulouse, 2012.

Leila S SHAFTI, Pablo A HAYA, Manuel GARCIA-HERRANZ et Eduardo PÈREZ : Evolutionary feature extraction to infer behavioral patterns in ambient intelligence. *Dans Ambient Intelligence*, pages 256–271. Springer, 2012.

Nikolaos I SPANOUDAKIS et Pavlos MORAITIS : Agent based architecture in an ambient intelligence context. *Dans Proceedings of the 4th European Workshop on Multi-Agent Systems*, pages 1–12, 2006.

Thomas STRANG et Claudia LINNHOFF-POPIEN : A context modeling survey. *Dans The Sixth International Conference on Ubiquitous Computing*, 2004.

John STRASSNER, Sven MEER, Declan O SULLIVAN et Simon DOBSON : The use of context-aware policies and ontologies to facilitate business-aware network management. *Journal of Network and Systems Management*, 17(3):255–284, 2009.

Richard S SUTTON et Andrew G BARTO : *Reinforcement learning : An introduction*, volume 1. Cambridge Univ Press, 1998.

Dante I TAPIA, Javier BAJO, Juan M SANCHEZ et Juan M CORCHADO : An ambient intelligence based multi-agent architecture. *Dans Developing Ambient Intelligence*, pages 68–78. Springer, 2008.

Sylvain VIDEAU, Carole BERNON et Pierre GLIZE : Towards controlling bioprocesses : A self-adaptive multi-agent approach. *Journal of Biological Physics and Chemistry*, 10(1):24–32, 2010.

Christopher John Cornish Hellaby WATKINS : *Learning from delayed rewards*. Thèse de doctorat, University of Cambridge, 1989.

M. WEISER : The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991.

Jason WESTON, André ELISSEEFF, Bernhard SCHÖLKOPF et Mike TIPPING : Use of the zero norm with linear models and kernel methods. *The Journal of Machine Learning Research*, 3:1439–1461, 2003.

Sofia ZAIDENBERG : *Apprentissage par renforcement de modeles de contexte pour l'informatique ambiante*. Thèse de doctorat, Institut National Polytechnique de Grenoble-INPG, 2009.