Université de Montréal

Conception d'un langage de description d'agents cognitifs

par

Charles De Léan

Département d'informatique et de recherche opérationnelle

Faculté des arts et sciences

Mémoire présenté à la Faculté des études supérieures en vue d'obtention du grade de Maître ès sciences (M.Sc.) en informatique

Août, 1998

UNIVERSITÉ DE MONTRÉAL

FACULTÉ DES ÉTUDES SUPÉRIEURES

Ce mémoire intitulé

"Conception d'un langage de description d'agents cognitifs"

présenté par:

CHARLES DE LÉAN

A été évalué par un jury composé des personnes suivantes:

Madame Esma AÏMEUR	Président-rapporteur
Monsieur Claude FRASSON	Directeur de recherche
Monsieur Rudolf KELLER	Membre

Mémoire accepté le : 20.10.1998

Sommaire

Dans cet ouvrage, nous présentons une solution proposée dans un projet de conception de systèmes tutoriels intelligents (STI), le projet SAFARI. Ce projet est complexe et utilise des agents pour faciliter l'implantation et le déroulement de stratégies pédagogiques, c'est-à-dire des façons de présenter la matière à un apprenant. Les agents utilisés dans ce projet s'appellent acteurs et ont des propriétés particulières. Pour remédier aux différentes contraintes du projet SAFARI et de l'architecture des acteurs, nous présentons un langage d'implantation pour les acteurs, ADL que nous discutons en détail. Ce mémoire se termine sur une réflexion portée sur l'apprentissage machine, aspect important pour réaliser une partie vitale des acteurs: la couche cognitive. Ce mémoire cherche à montrer que le langage ADL permet de simplifier la création d'acteurs dans le projet SAFARI, et à proposer une façon d'aborder la couche cognitive des acteurs. Nous montrons aussi que l'utilisation d'algorithmes d'apprentissage n'est pas suffisant: il faut aussi savoir spécifier ce sur quoi on veut apprendre.

Table Des Matières

Table des figures

Remerciements

L'auteur de ce mémoire veut remercier toute l'équipe de SAFARI pour son expérience inoubliable qu'il aura passé avec les différents membres. L'auteur voudrait remercier particulièrement Claude Frasson, son directeur qui lui a fait confiance en lui donnant un sujet portant sur partie vitale de ses recherches et qui lui a permis de vivre une expérience exceptionnelle: celle de faire partie de l'organisation d'une conférence (ITS '96 à l'université de Montréal). Esma Aïmeur a pu faire découvrir à l'auteur le travail de synthèse d'articles et le traitement de connaissances (dans ce cas-ci, l'apprentissage machine). Dans un de ses cours, l'auteur a rédigé une synthèse qui lui fut très utile pour cerner le problème de la couche cognitive (portant sur l'apprentissage multistratégique). L'auteur veut aussi remercier chaleureusement Thierry Mengelle, pour son support académique et psychologique sans compter son sens de l'humour et son réalisme, ainsi que Martine Gemme (secrétaire de M. Frasson) pour son grand sourire, sa bonne humeur, sa disponibilité, sa très grande serviabilité et son sens de la débrouillardise.

L'auteur a beaucoup appris au sein du projet SAFARI, tant au niveau académique qu'au niveau des relations humaines. Son expérience lui apprit beaucoup sur les nombreuses activités qui tournent autour de la recherche, et comment interagir avec les membres d'un grand groupe de travail.

L'auteur voudrait aussi remercier le conseil de recherches en sciences naturelles et en génie du Canada (C.R.S.N.G), pour sa généreuse bourse qui lui permis de faire ses travaux, ainsi qu'au Ministère de l'Industrie, du Commerce, des Sciences et de la Tachnologie du Québec (MICST) qui lui a versé un complément à sa bourse dans le cadre du projet SYNERGIE.

Chapitre 1

Introduction

Après l'apparition de l'intelligence artificielle et des premiers systèmes tutoriels dans les années 50, on s'est intéressé à concevoir des outils de formation. On a rapidement cherché à séparer l'expertise pédagogique de la matière à enseigner d'où la naissance des systèmes tutoriels intelligents (STI) dans les années 80. Les ordinateurs sont devenus de plus en plus puissants et accessibles au grand public. Un large marché de l'information s'est alors rapidement ouvert, dont un marché de la formation, d'où le besoin de développer des STI.

Un projet universitaire cherche à établir des outils permettant à des pédagogues de concevoir des STI. Ce projet appelé SAFARI [Frasson *et* Aïmeur, 96][Frasson *et al.*, 96] (pour Système d'Aide à la Formation basé sur l'Analyse de Raisonnements Interactifs), se déroule au Département d'Informatique et de Recherche Opérationnelle de l'Université de Montréal et implique des partenaires industriels (VPI et Novasys).

Le projet SAFARI offre un curriculum [Nkambou *et al.*, 96] qui structure la matière à enseigner, un planificateur [Lê *et al.*, 98] qui décide des parties de la matière à présenter lors d'une session d'apprentissage, un modèle de l'apprenant [Nkambou *et al.*, 96] contenant les informations sur l'utilisateur (ou apprenant), un ensemble de ressources qui contient le matériel à présenter ainsi que les différents tests pour évaluer les connaissances de l'apprenant, et finalement un ensemble de modules pédagogiques qui peuvent prendre en charge la présentation du cours en suivant une expertise pédagogique. SAFARI donne des outils (i.e. des éditeurs) permettant de faciliter la réunion des différentes expertises permettant la fabrication des différents modules.

[Frasson *et al.*,96] explique que deux grandes caractéristiques sont apparues lors de l'évolution des STI: l'apprentissage dans un STI se fait dans un milieu social (à travers une interaction avec plusieurs partenaires). Cette forme d'apprentissage est appelée apprentissage coopératif.

[Frasson *et al.*,96] propose une architecture de module pédagogique distribuée entre plusieurs entités informatiques appelés agents [Bradshaw, 97][Rossignol *et al.*, 96][Gilbert *et al.*, 95][Cheikes, 95][Tecuci *et* Hieb, 1996][Hayes-Roth, 95 dans Franklin *et* Graesser, 96]. Ces derniers ont chacun leur rôle, et coopèrent avec l'usager dans son cours. Ce module cherche non seulement à coopérer avec l'apprenant dans un cours, mais aussi à faciliter le travail du pédagogue qui doit chercher à mettre en oeuvre des stratégies d'interactions avec l'usager appelées stratégies pédagogiques.

Comme les stratégies pédagogiques ci-dessus mentionées sont complexes, le pédagogue doit pouvoir les concevoir de façon interactive avec des agents, laissant le soin à ces derniers de les apprendre. De plus, ces stratégies peuvent être utilisées dans d'autres STI. Elles doivent alors être génériques.

Les agents s'avèrent prometteurs pour l'établissement de systèmes (ou modules) complexes: des systèmes distribués, des interfaces usagers graphiques et des STI par exemple. De plus, ils possèdent des propriétés intéressantes permettant de répondre aux exigences de SAFARI. Les agents ont la propriété d'éducation (possibilité d'apprendre de nouveaux algorithmes d'un usager), ce qui permet un apprentissage interactif avec un pédagogue pour obtenir un comportement désiré. Cette propriété est utile pour une personne qui n'est pas experte en programmation (i.e. un pédagogue) et qui désire concevoir des agents. Cette propriété répond alors à un besoin de SAFARI (voir plus haut). Les agents peuvent aussi avoir la propriété cognitive, pouvant établir un apprentissage réaliste, c'est-à-dire un apprentissage simulant celui de l'apprenant. Cette

propriété est importante pour déterminer les mécanismes d'apprentissage de l'utilisateur et les annoncer au modèle de l'apprenant (décrit plus haut), permettant ainsi au STI (en particulier aux agents ici) d'adapter son cours pour en faciliter la compréhension.

1.1 But du mémoire

Ce mémoire comporte deux buts. Le premier est de décrire un langage permettant de faciliter la création d'agents dans un module pédagogique. C'est le but principal de ce mémoire. L'autre but est de cerner les difficultés liées à l'établissement des propriétés d'éducation et de cognition des agents dans le cadre du projet SAFARI, sans toutefois donner une solution complète.

1.1.1 L'établissement d'un langage de description d'agents

Ce mémoire cherche à proposer une solution à la création d'agents pour les modules pédagogiques du projet SAFARI. Rappelons que ces agents doivent s'intégrer dans l'architecture bien définie de SAFARI et pouvoir être utilisés dans les différents STI qui seront construits en utilisant les outils de ce projet. De plus, les modules pédagogiques peuvent contenir une communauté d'agents. Nous verrons à travers le mémoire que ces modules doivent être écrits en ANSI C++, peuvent être modifiés à travers un éditeur et doivent s'intégrer facilement au reste du projet SAFARI.

Ces contraintes techniques ont amené l'auteur à concevoir le langage interprêté ADL, pour Actor Description Language. L'interprête est écrit en C++ et se charge des contraintes techniques telles que la gestion des processus parallèles (non inclus dans le C++ standard) et l'intégration avec le reste du projet SAFARI. Le langage lui-même cherche à respecter l'architecture des agents proposée par Frasson [Frasson *et al.*,96] et faciliter l'édition des agents à travers un éditeur [Mengelle *et al.*,97].

Il existe d'autres langages permettant d'établir des communautés d'agents. Un de ces langages est KQML [Finin *et al.*,94]. Nous verrons comment ADL et KQML ont des

buts différents, montrant ainsi l'intérêt et la spécialisation du langage proposé dans ce mémoire.

1.1.2 Reflexions les propriétés d'éducation et de cognition

L'architecture des agents de SAFARI comporte une couche utilisant des algorithmes d'apprentissage machine. Cette couche appelée couche cognitive permet entres autres les propriétés d'éducation (éducable) et de cognition. Nous ne proposons pas une solution complète pour cette couche. Nous décrivons cependant deux projets impliquant des algorithmes d'apprentissage machine pour obtenir ces propriétés. Ces projets utilisent une forme d'apprentissage apellée multistratégique. Nous cherchons à montrer l'intérêt d'une telle forme d'apprentissage machine pour la couche cognitive.

Nous montrons finalement à travers un prototype en ADL que la simple utilisation d'algorithmes d'apprentissage n'est pas suffisante pour l'établissement de la couche cognitive. Il est vital de préciser la sémantique utilisée lors de l'apprentissage, c'est-à-dire les différents termes utilisés pour l'établissement des nouvelles règles.

1.2 Composition du mémoire

Nous décrivons d'abord le contexte de SAFARI avec ses différentes contraintes, l'intérêt d'utiliser des agents pour résoudre le problème complexe de la construction des modules pédagogiques et la description de leur architecture des agents utilisés. Nous examinons ensuite de façon détaillée comment ADL traite les différentes contraintes de SAFARI et permet de réaliser des agents. Quelques prototypes sont discutés pour illustrer l'utilisation de ADL et nous finissons par des réflexions sur la façon de doter les agents d'apprentissage machine, pour établir les propriétés d'éducation et de cognition.

Dans le chapitre 2, nous discutons des agents et de leur utilité pour le développement de systèmes (ou modules) complexes. Nous discutons ensuite plus en détail des différents modules composant le projet SAFARI, dont le module pédagogique, qui contient

l'expertise pédagogique du STI. Ce module a entres autres pour rôle de faciliter le transfert d'expertise pédagogique dans le STI. Nous examinerons comment les agents peuvent faciliter le développement de ce module.

Le chapitre 3 décrit Actor Description Language (ou ADL). Ce langage interprété a pour but de faciliter la création des agents dans le projet SAFARI tout en respectant les différentes contraintes du projet. Nous montrons la spécialisation de ce langage en le comparant avec un autre langage pour agents, KQML. Nous voyons ensuite comment les différents aspects de l'architecture des agents (vus en chapitre 2) sont exprimés à travers ADL. Nous décrivons finalement les grandes lignes du fonctionnement de l'interpète (ou moteur ADL).

Le chapitre 4 décrit pour des fins d'illustration quelques prototypes développés dans le cadre de SAFARI. Nous discutons entres autres d'un prototype de la stratégie du perturbateur où la ressource (ou activité de l'apprenant) offre une analyse des actions de l'apprenant, ainsi que d'un prototype sur l'apprentisssage machine et la couche cognitive. Ce dernier prototype montre l'importance de définir une sémantique pour l'apprentissage.

Le chapitre 5 discute d'une forme d'apprentissage machine appelée multistratégique. Cette forme d'apprentissage prometteuse permet une plus grande flexibilité que les formes d'apprentissage suivant une seule stratégie (i.e. méthode de raisonnement). Nous y discutons de deux projets d'apprentissage multistratégiques, DISCIPLE et META-AQUA, traitant respectivement des propriétés d'éducation et cognitifs d'un agent, propriétés vitales dans le cadre de SAFARI. Ce chapitre cherche à montrer l'intérêt de telles formes d'apprentissage pour l'agent de SAFARI.

La conclusion discute de l'utilité et des limites du langage proposé, et de certaines reflexions sur l'établissement de la couche cognitive avec des algorithmes d'apprentissage.

En résumé, le module pédagogique du projet SAFARI est ambitieux et complexe et nous utilisons des agents pour en faciliter le développement. Ce mémoire traite de ADL, un langage permettant l'implantation d'agents pour développer ce module. Les agents de SAFARI possèdent les propriétés d'éducation et de cognition et nécessitent des outils d'apprentissage machine. Nous proposons des éléments de solutions pour l'implémentation de telles propriétés.

Chapitre 2

Les Acteurs

Ce chapitre cherche à montrer le problème auquel s'adresse ce mémoire, ainsi que son contexte. Ce dernier est l'architecture de SAFARI. Nous allons décrire cette architecture, montrer comment nous pensons gérer un de ses modules (module pédagogique) avec l'aide d'entités informatiques appelés agents et expliquer ce choix. Pour ce faire nous décrivons d'abord ce qu'est un agent et son utilité. L'architecture des agents utilisés dans SAFARI est décrite dans ce chapitre. Une fois le contexte ainsi expliqué, nous posons le problème auquel s'adresse ce mémoire: faciliter l'implantation d'agents dans un module pédagogique de SAFARI.

Ce chapitre explique que la définition d'un agent est difficile à poser et qu'elle n'est toujours pas clairement définie. Les agents y sont alors classés par les différentes propriétés qu'ils peuvent avoir. Ils servent à l'élaboration de modules complexes tels que les systèmes distribués, les interfaces usagers et les STI. Or, le projet SAFARI vise la construction de STI. Ces STI se composent de plusieurs modules, dont le curriculum, le planificateur, le modèle de l'apprenant et les modules pédagogiques. L'expertise pédagogique des STI est gérée par ces derniers modules. Cette gestion s'avère complexe, d'où l'intérêt de l'utilisation d'agents pour les réaliser. Les modules pédagogiques gérés par des agents (appelés acteurs dans le projet SAFARI) forment un ensemble appelé module des acteurs. Ces derniers possèdent une architecture permettant de résoudre les différents difficultés de la fabrication des modules pédagogiques. Dans ce mémoire, nous allons nous attarder à l'architecture des acteurs au sein de SAFARI et montrer ainsi les contraintes traitées dans ce mémoire.

2.1 Les Agents

Le terme d'agent est à la mode dans les milieux de recherche et de l'industrie, où chacun possède sa propre définition.

Bradshaw [Bradshaw, 97] explique que la difficulté d'avoir une définition claire d'un agent vient du fait que ce concept est en fait un outil permettant aux programmeurs de modéliser des programmes complexes plus facilement. Nous nous retrouvons dans une situation similaire aux agents avec programmation orientée objet qui permet de concevoir des programmes volumineux et complexes, prévoir la réutilisation de code, etc, de façon plus aisée que pour d'autres formes de programmation. Le terme orienté objet est populaire dans les milieux de recherche et de développement. Il en est de même avec le terme d'agent. Ce dernier est utilisé abusivement et un consensus s'avère difficile. Nous allons donc voir différentes définitions des agents, les classer selon différentes propriétés que ces définitions leur confèrent et montrer leur utilité dans différents domaines, dont les STI.

2.1.1 Définitions

Bradshaw [Bradshaw, 97] propose la définition de Shoham d'un agent: *a software entity which functions continuously and autonomously in a particular environment, often inhabited by other agents and processes*. Un agent est donc un logiciel dont le déroulement est continu et autonome, c'est à dire qu'il est toujours en fonction dans son environnement et qu'il ne nécessite pas l'intervention ou le guidage d'un humain pour son fonctionnement. Plusieurs agents peuvent cohabiter dans un même environnement. Bradshaw explique aussi que l'activité d'un agent doit être flexible et intelligente, c'est à dire qu'il doit pouvoir s'adapter de façon adéquate aux changements de son environnement.

[Rossignol *et al.*, 96] relate plusieurs autres définitions. Parmi ces définitions, nous pouvons en proposer quelques-unes qui peuvent complémenter celle de Shoham.

- l'agent d'IBM [Gilbert *et al.*, 95]: "*Les agents intelligents sont des entités informatiques qui effectuent une série d'opérations au nom d'un usager ou d'un autre programme et qui, pour ce faire, utilisent des connaissances ou représentations des buts ou désirs de l'usager*";
- l'agent de GIA [Cheikes, 95]: "*Un agent est un programme qui communique par le moyen de messages encodés à l'aide d'un langage de communication d'agents (ACL)*";
- l'agent Disciple [Tecuci *et* Hieb, 1996]: "*Nous définissons un agent autonome comme un système à base de connaissances qui peut interagir avec un usager (ou d'autres agents) et l'assister de différentes façons*";
- l'agent de Hayes-Roth [Hayes-Roth, 95 dans Franklin *et* Graesser, 96]: "*Les agents intelligents effectuent continuellement trois fonctions: percevoir les états dynamiques et raisonner afin d'interpréter les perceptions, d'effectuer des inférences, de résoudre des problèmes et de déterminer les actions à entreprendre*".

À travers ces différentes définitions, nous pouvons voir que les différents auteurs définissent un agent de façon à les adapter à leurs travaux respectifs. Nous voyons ici que les agents ont un fonctionnement continu et autonome, peuvent être des délégués de l'usager pour faire un travail, avoir un langage de communication entre eux, avoir à interagir avec l'usager, avoir une base de connaissances et faire des inférences pour déterminer leurs actions en fonction de leur environnement.

2.1.2 Propriétés d'un agent

Étant donné que les définitions des agents varient beaucoup, nous cherchons à les classer selon plusieurs propriétés qu'ils peuvent avoir.

Frasson [Frasson *et al.*,96] propose plusieurs propriétés d'agents.

- **réactivité**: réagir aux changements de l'environnement sans faire de raisonnement: la réaction est immédiate au changement et ne nécessite pas l'utilisation de processus complexe de prise de décision tel que la planification. Un robot qui se déplace dans

un couloir et qui tourne à gauche dès qu'il rencontre un mur peut être qualifié de réactif. Ceci ne veut pas dire que ce comportement n'est pas intelligent. En effet, Rossignol [Rossignol *et al.*, 96], explique que Brooks indique qu'un comportement intelligent peut se faire sans raisonnement abstrait et peut même être une propriété émergente d'un système complexe [Brooks, 90 dans Wooldridge *et* Jennings, 95];

- **planification**: déterminer une séquence d'actions pouvant mener à un certain but;
- **prédiction**: le fait de pouvoir déterminer à l'avance son comportement, c'est-à-dire de planifier son comportement à partir d'une hypothèse (données de départ);
- **diagnostic**: possibilité de déduire quelque chose à partir d'une hypothèse, ou d'une série d'hypothèses;
- **éducation**: possibilité de recevoir de nouveaux algorithmes de l'usager ou du système et la possibilité d'analyser les actions passées pour assister à ce travail d'apprentissage;
- **adaptatibilité (ou apprenant)**: possibilité de modifier lui-même sa façon de réagir à son environnement et voir même sa base de règles, donc son propre raisonnement;
- **cognitif**: pouvoir modéliser un humain dans une situation d'apprentissage, c'est-à-dire de faire un apprentissage de façon réaliste. En apprenant de l'usager, ses actions peuvent être prédites. Avec un tel modèle, l'agent peut, par exemple, imiter l'apprenant par la suite.

À ces propriétés, nous pouvons ajouter d'autres propriétés rapportées par [Rossignol *et al.*, 96].

- **sociabilité**: agir avec d'autres agents ou humains à travers un langage de communication;
- **initiative**: déclencher des actions autrement que par réaction à son environnement. Il peut aussi prendre ses décisions à partir de buts;
- **mobilité**: possibilité de se déplacer dans un réseau électronique. Un agent peut ainsi gérer les ressources d'un autre ordinateur par exemple;
- **véracité**: l'agent ne va pas volontairement transmettre de fausses informations;
- **bienveillance**: l'agent n'a pas de buts en conflit et cherchera à faire ce pour quoi il a

été conçu;

- **rationalité**: l'agent cherche à atteindre des buts.

Nous proposons aussi de nouvelles propriétés qui ont été définies à l'intérieur de SAFARI.

- **caractère, personnalité, ou sentiments**: l'agent peut avoir une personnalité réaliste et peut avoir un état émotionnel. Ceci peut devenir très utile lorsque l'agent possède une interface graphique avec l'usager. [Rossignol *et al.*, 96] rapporte que Bates [Bates dans Woolridge *et* Jennings, 95] appelle ce type d'agent *Believable agents*. En effet, pour l'usager, un agent montrant des sentiments est beaucoup plus vraisemblable parce que l'interaction ressemble beaucoup plus à celle d'un humain;

- **orienté par un but**: l'agent se définit par un but décomposable en sous-buts. Son travail est d'atteindre ce premier but, en parcourant l'arborescence de sous-buts par exemple. Généralement, les agents ont un but implicite, mais cette propriété les force à le définir de façon explicite.

[Frasson *et al.*,96] explique que les agents ont pour propriétés élémentaires d'être réactifs et autonomes [Castelfranchi, 95]. Ils agissent lorsque certains changements se font dans leur environnement tout suivant certains buts et peuvent s'exécuter sans contôle direct d'un humain. Frasson explique aussi qu'ils peuvent communiquer avec d'autres agents en utilisant un langage d'agents. Les agents sont donc sociables. Il explique que l'évolution des agents se fait en trois étapes donnant trois catégories d'agents. La première catégorie contient les agents réactifs, pouvant planifier, prédire et diagnostiquer. La deuxième ajoute la propriété d'éducation, et la troisième, d'adaptation.

2.1.3 L'utilité les agents

[Rossignol *et al.*, 96] rapporte la possibilité manipuler de plus grandes quantités d'informations venant d'activités diverses de façon plus efficace grâce aux agents. Nous pourrions ainsi demander à nos agents de faire des recherches d'informations pertinentes

sur un certain sujet tout en respectant les goûts de l'usager et en apprenant à mieux chercher d'une fois à l'autre. Nous voyons ici un outil qui peut s'avérer extrêmement utile dans le domaine de la recherche. De la même façon, Rossignol rapporte que le contôle aérien [Wooldridge *et* Jennings, 95], l'administration de réseau, la recherche des biens de consommation et le filtrage du courrier électronique (i.e. éliminer tout courrier inintéressant) sont bouleversés par l'arrivée des agents.

Les projets utilisant des agents sont multiples. Cependant, nous ne voulons pas trop aborder le sujet dans ce mémoire. Bradshaw [Bradshaw, 97], indique que les agents sont utiles pour deux grands domaines d'application: les systèmes distribués et les interfaces graphiques (GUI).

2.1.3.1 Les systèmes distribués

Avec l'arrivée de services de réseau indépendants des systèmes d'exploitation dont l'exécution est distribuée (exploration de répertoires sur une autre machine, le Web, les programmes en Java), on envisage l'utilisation d'agents comme gestionnaires de ressources [Bradshaw, 97]. Ils peuvent traiter les requêtes des usagers, communiquer avec le réseau et planifier l'utilisation des différentes ressources (i.e. une imprimante). Ainsi, l'usager n'a plus à s'occuper des protocoles de communication ou des conflits d'utilisation de ressources, mais simplement de donner ses requêtes. Les agents peuvent communiquer entre eux et planifier les actions nécessaires pour faire le travail.

2.1.3.2 Les interfaces graphiques

Nous commençons à atteindre les limites des interfaces graphiques [Bradshaw, 97]. Elles sont très utiles pour faire des actions simples, mais posent des problèmes pour des tâches plus complexes. En effet, dans le cadre où nous avons affaire à des systèmes distribués, que faire pour explorer des milliers d'items en même temps (i.e. de fichiers dans un répertoire), différer une action (agenda ou réaction à un événement particulier), composer des actions de base en actions plus complexes (série d'actions répétées plusieurs fois dans traitement de texte qui pourraient devenir une seule opération par exemple), ou améliorer

le comportement de notre interface (si une même opération est répétée souvent, il faudrait qu'elle s'automatise, donc à ce qu'un agent propose d'ajouter une nouvelle fonctionnalité au logiciel).

L'utilisation d'agents peut réduire ces difficultés liées aux interfaces. Bradshaw indique qu'ils sont capables de filtrer les informations accessibles, de déclencher une action devant un certain événement, ils peuvent expliquer comment faire certaines opérations et pourquoi l'usager ne réussit pas à faire ce qu'il veut faire (diagnostic), ils peuvent résoudre des problèmes non prévus (avec une base de connaissances), et ils peuvent apprendre de nouveaux algorithmes (à partir des commandes de l'usager par exemple, pour proposer l'ajout d'une nouvelle fonctionnalité dans un logiciel).

Bradshaw propose que toute composante d'une interface graphique devienne un agent. Un agent peut s'occuper de la présentation graphique, un autre être un assistant, d'autres peuvent gérer les logiciels et les différentes composantes (pour reconfigurer automatiquement un périphérique utilisé d'un logiciel à l'autre par exemple).

2.1.4 Les agents dans un STI

Un STI est un système complexe à mettre en oeuvre. Comme les agents sont utiles pour simplifier la production de systèmes complexes (modularisation et réutilisation de composantes), ils peuvent simplifier la production d'un STI [Cheikes, 95].

Un exemple d'utilisation d'agent pour construire un STI se trouve dans un projet d'exploration d'espaces virtuels [Doyle *et* Hayes-Roth, 96]. Ce projet fait partie d'un plus grand projet appelé Virtual Theater qui permet la conception des agents interagissant avec des humains pour les assister dans leurs tâches. On cherche à produire des agents avec des sentiments à cause de l'interaction importante avec les humains. Dans le projet d'exploration, nous avons l'équivalent d'un système tutoriel simple. Ce système aide un apprenant, un enfant, à utiliser un environnement rejoignable par réseau (i.e. internet).

De tels environnements peuvent être des classes virtuelles, un corps humain que explorable de l'intérieur, des musées, des jeux, etc. Le système se compose de deux agents. L'un s'appelle Merlyn et l'autre Wart. Ces noms sont basés sur *The Once and Future King* de T.H. White, où le fameux personnage de Merlin (en anglais, Merlyn) va donner des cours au petit roi Arthur (qu'il appelle Wart).

Merlyn est un agent capable d'explorer l'environnement et de trouver les différentes actions possibles ainsi que toutes les informations qui s'y trouvent. Wart est un agent qui personnifie l'enfant dans le monde virtuel. Merlyn se comporte comme un personnage sympathique envers Wart. On le dote d'une personnalité assez complexe pour qu'il paraisse comme un véritable humain envers l'enfant. Ainsi, nous avons un personnage qui va susciter une interaction avec l'enfant. Merlyn doit chercher à proposer à l'enfant certaines activités à partir des informations trouvées dans l'environnement. Idéalement, Merlyn doit même être capable de jouer avec l'enfant à travers l'environnement.

Wart est un agent qui en fait est une marionnette pour l'enfant. Les actions de l'enfant dans l'environnement (tel que parler et bouger) se font à travers l'agent. Merlyn peut proposer des commandes de haut niveau à l'enfant telles que se rendre à un certain endroit dans l'environnement, ce qui revient à faire une série de commandes de déplacement. Ainsi, l'enfant peut interagir avec Merlyn à travers des commandes de plus haut niveau que ce que l'environnement propose. Wart présente à l'enfant différentes commandes qui peuvent être exécutées. Ces commandes seront une sélection des commandes possibles dans l'environnement et les commandes que Merlyn propose à l'enfant. Ceci n'empêche pas l'enfant de faire lui-même ses actions au lieu de choisir une des options qui lui sont proposées. L'enfant peut en tout temps envoyer ses commandes directement à l'environnement. Il peut aussi décider du comportement social que Wart a envers Merlyn. Il peut par exemple lui être sympathique, antipathique ou même lui jouer des tours.

Nous avons dans Virtual Theater un exemple de système tutoriel très simple. En effet, la

description du domaine est limitée à ce que l'environnement peut donner comme information aux agents (nous souhaitons que cette description soit suffisante). Le modèle de l'apprenant est contenu dans les agents (surtout Merlyn) qui peuvent se souvenir des actions passées de l'enfant et déduire ses préférences. L'activité d'apprentissage se fait avec les agents à travers l'espace virtuel.

2.2 Le projet SAFARI

Le projet SAFARI (Système d'Aide à la Formation basé sur l'Analyse de Raisonnement Interactif) [Frasson *et* Aïmeur, 96][Frasson *et al.*, 96], financé par le programme SYNERGIE du Ministère de l'Industrie, du Commerce, des Sciences et de la Technologie du Québec (MICST), comporte deux partenaires industriels associés au projet, VPI inc. et NOVASYS inc. et 36 chercheurs et étudiants à travers plusieurs universités: Université de Montréal, Université du Québec à Montréal, McGill University, Bishop University et Université du Québec à Chicoutimi. Son but est de fournir des outils permettant de construire des STI s'appliquant à un cours complet (avec des activités d'acquisition d'informations et d'habiletés de résolution de problèmes) et pouvant être basés sur des simulateurs. Cette nouvelle génération de systèmes tutoriels vise les entreprises intéressées à faire de la formation (d'où l'intérêt des partenaires industriels dans ce projet).

2.2.1 Architecture générale

[Nkambou *et al.*, 96] explique qu'un STI se compose de trois modèles [Burns *et* Capps, 88][Self, 97]: un modèle de l'expert, un modèle du pédagogue et un modèle de l'apprenant. Le premier représente l'expertise du domaine à enseigner. Il doit être capable de résoudre les problèmes proposés à l'apprenant à partir de ses connaissances. Le deuxième contient l'expertise qui permet de corriger l'apprenant (méthodes de sélection d'exemples et d'analogie entres autres), ainsi que de choisir le moment où corriger l'apprenant. Le troisième modélise les connaissances de l'apprenant sur la matière, ainsi que son comportement.

Dans l'architecture du projet SAFARI (Figure 1), le module de l'expert se trouve dans un module appellé curriculum, le modèle de l'apprenant conserve son nom et le modèle du pédagogue se trouve dans différents modules dits pédagogiques.

Le modèle de l'apprenant [Nkambou *et al.*, 96] est un module contenant trois sortes d'informations sur l'apprenant:

- cognitives: ce que l'apprenant connaît déjà sur le cours. Ces informations sont en fait une sous-partie du curriculum (appelée *overlay*);
- conatives: comment l'apprenant travaille, raisonne et apprend;
- affective: l'état émotionnel dans lequel trouve l'apprenant.

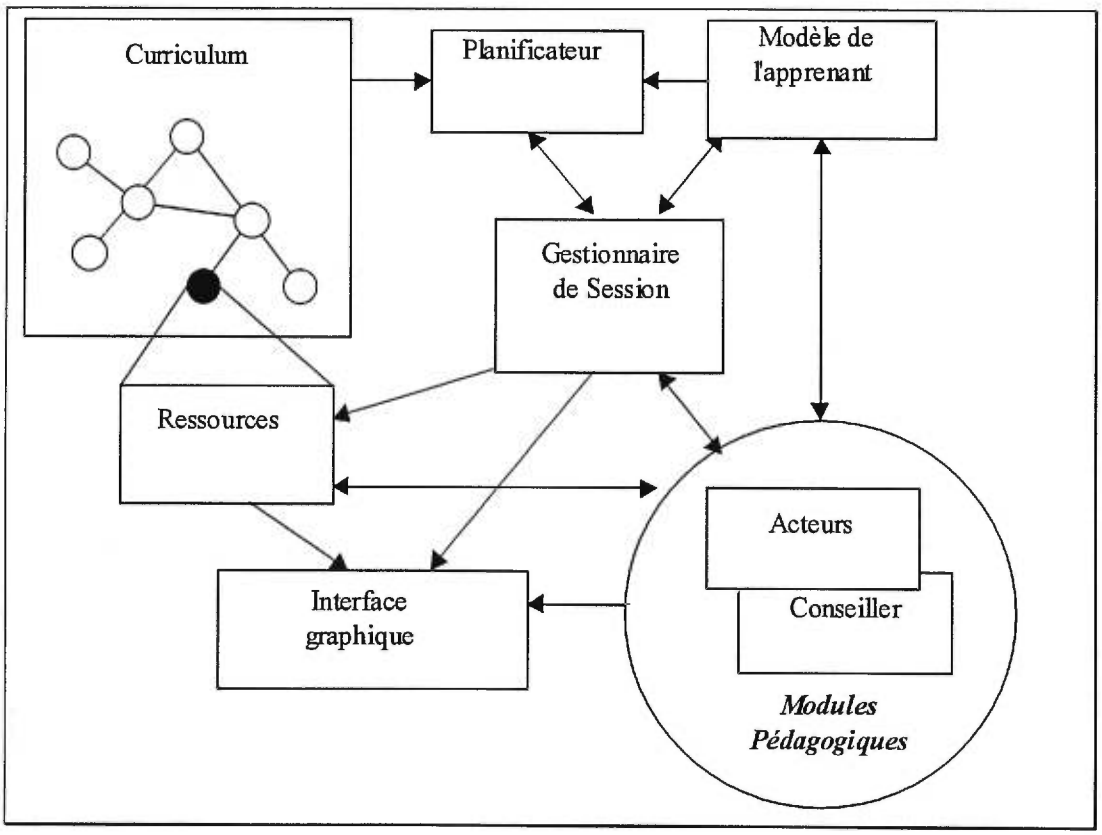

Figure 1. Architecture générale d'un STI de SAFARI

Les ressources sont les activités à travers lesquelles l'apprenant va acquérir des

connaissances et sera évalué. Elles sont définies dans le curriculum et sont de deux types, didactiques ou adidactiques.

- les ressources didactiques sont indépendantes du reste du système tutoriel. Le contrôle de leur déroulement est considéré comme une boîte noire pour le système;
- les ressources adidactiques contiennent les connaissances du domaine permettant d'analyser les actions de l'apprenant. Cependant, elles vont laisser à d'autres modules le soin de faire les différentes interventions pédagogiques auprès de l'apprenant.

Les modules pouvant choisir les interventions pédagogiques pour les ressources adidactiques se nomment modules pédagogiques. Parmi ces modules, nous trouvons le conseiller [Serroud *et al.*, 95] et les acteurs. Les modules pédagogiques sont indépendants des ressources adidactiques et peuvent être réutilisés d'un STI à l'autre. Ils peuvent consulter et mettre à jour le modèle de l'apprenant, recevoir des informations sur les actions de l'apprenant et contrôler le déroulement des ressources. Nous allons nous intéresser plus particulièrement au module des acteurs dans ce travail.

En fonction du modèle de l'apprenant, et du curriculum, le planificateur choisit quelles ressources doivent être présentées à l'apprenant et dans quel ordre. Ce choix s'appelle un plan. Ce plan et donné au gestionnaire de session, qui se charge de le suivre, de présenter les ressources et de déclencher un module pédagogique quand la ressource à présenter est adidactique. Le gestionnaire de session va aussi se charger de mettre à jour de modèle de l'apprenant à la fin de la présentation d'une ressource et de demander au planificateur un nouveau plan quand l'apprenant se trouve dans une impasse dans le déroulement du cours. Le processus du planificateur qui génère un nouveau plan s'appelle replanification.

Finalement, nous avons une interface graphique qui permet la présentation du cours, c'est-à-dire la présentation ordonnée selon le plan des différentes ressources. Cette interface permet aussi aux modules pédagogiques d'interagir avec l'apprenant pour l'assister avec les différentes ressources. Le gestionnaire de session traite les commandes plus générales portant sur la session elle-même, tel qu'une demande de replanification

venant de l'apprenant.

2.2.2 Le curriculum

Le curriculum permet une structuration de la matière de manière à simplifier le traitement d'un cours complet et complexe [Nkambou, 96]. SAFARI vise des STI pouvant traiter de tels cours, d'où l'importance du curriculum dans son architecture.

Le curriculum découle de l'apparition de la partie intelligente des systèmes tutoriels. Nkambou date cette apparition en 1970, où Carbonnell [Carbonnell, 70] propose le système tutoriel SCHOLAR: Les connaissances liées à l'enseignement (connaissances pédagogiques) se séparent de celles relatives à l'expertise du domaine à enseigner et des techniques d'intelligence artificielles pour les définir sont utilisées.

Nkambou définit le curriculum comme suit: *Une représentation structurée de la matière à enseigner en termes de capacités au sens de Gagné, d'objectifs dont l'atteinte contribue à l'acquisition de capacité et de ressources* [...]. *Tous ces éléments sont organisés dans des structures de connaissances pour soutenir l'enseignement d'une matière*. On peut ainsi *organiser un environnement riche et varié pouvant supporter la génération de cours, la planification et le déroulement de l'enseignement, et certains aspects reliés à la modélisation de l'apprenant.*

Il définit les types de composantes comme suit:

- une capacité est une connaissance permettant à une personne de réussir dans l'exercice d'une activité. Cette information est emmagasinée dans la mémoire à long terme de l'apprenant;
- un objectif décrit le résultat que doit atteindre l'enseignement (performances de l'apprenant ou ensemble de capacités obtenues);
- une ressource est une activité à travers laquelle l'apprenant va acquérir des connaissances et sera évalué. Elle permet donc d'atteindre une capacité (i.e. atteindre un objectif).

L'organisation du curriculum est *une structure essentiellement déclarative*. Elle se fait en trois réseaux interreliés. Un pour chaque type de composante dans le curriculum.

2.2.3 Le planificateur

[Lê *et al.*, 98] explique qu'un système tutoriel doit tenir compte de plusieurs contraintes comme le temps alloué à une session d'apprentissage, les connaissances de l'étudiant sur le domaine et ses préférences. De plus, l'apprenant peut se trouver dans plusieurs formes d'impasses telles que la confusion (ne pas comprendre malgré les ressources présentées), l'oubli des préalables au cours et la désorientation (ne plus savoir où on est rendu dans le cours). Lê propose un planificateur pouvant traiter ces différents problèmes.

Une session d'apprentissage est planifiée de façon dynamique. Au début de la session, le planificateur a pour travail de choisir quelles seront les ressources présentées et dans quel ordre, en utilisant une expertise (des règles) indépendante du domaine pour répondre aux différentes contraintes de la session. Un tel choix est appelé plan.

Une session est planifiée en quatre phases.

- une phase d'introduction, qui va indiquer à l'apprenant le but de la session, faire des rappels et introduire la matière;
- une phase de présentation de la matière;
- une phase de test, qui va présenter les exercices, évaluer l'apprenant et lui donner les résultats;
- une phase de récapitulation où la matière est brièvement revue, où l'usager est informé de ses résultats et où ce qui sera présenté à la prochaine session est annoncé.

Si un problème survient lors de la session (l'apprenant est dans une impasse), le planificateur va en chercher la cause, basée entres autres sur l'état des connaissances de

l'apprenant, et générer un nouveau plan permettant de continuer la session. Ce nouveau plan prend en compte les difficultés survenues et choisit des ressources permettant de remédier à la situation. Ce processus est appelé replanification.

2.2.4 La création des modules pédagogiques

Rappelons que les modules pédagogiques ont pour fonction de gérer les ressources adidactiques. La conception de tels modules peut s'avérer ardue. En effet, comment peut-on acquérir une expertise pédagogique dans un STI? Une telle expertise peut être complexe et peut venir d'un expert qui n'a pas nécessairement d'aptitudes pour la programmation (i.e. un pédagogue).

Un éditeur [Mengelle *et al.*,97] a été développé dans SAFARI pour permettre à un expert pédagogique d'implanter des modules pédagogiques en utilisant des agents. Ces derniers apportent plusieurs solutions au problème d'implantation de module pédagogique tel que la réutilisation de code et le fait qu'ils puissent être éducables et cognitifs.

Dans le projet SAFARI, ces agents s' appellent des acteurs. Une stratégie pédagogique est une sélection (ou société) d'acteurs. Le module des acteurs proposé dans l'architecture de SAFARI devient alors une collection de stratégies pédagogiques. Plusieurs stratégies pédagogiques sont déjà définies dans la littérature tels que la stratégie du compagnon [Chan *et* Baskin, 90] et celle du perturbateur [Aïmeur *et al.*, 95].

Notre éditeur crée alors des stratégies pédagogiques et établit le cycle de production de stratégie proposé dans la Figure 2. Ce cycle se décrit comme suit:

- L'expert pédagogique génère les acteurs faisant partie de sa stratégie en leur donnant une expertise de base. Ceci revient à leur donner des règles. Ce processus s'appelle instruction classique.

- Ensuite, il va les tester sur un exemple concret, avec un apprenant qui suit un certain

cours par exemple. Ce test nécessite d'autres modules de SAFARI tels que le gestionnaire de session, le modèle de l'apprenant et le curriculum pour l'exécution du test et pour permettre à l'expert de recueillir les résultats.

- L'expert peut chercher à améliorer les acteurs pendant leur exécution. Cette deuxième forme d'instruction est appelée dynamique car elle se fait pendant l'exécution de l'agent. Cette instruction dynamique est en fait l'éducation de notre agent. L'expert peut alors retourner à la phase de tests. Ce type d'instruction est utilisé dans DISCIPLE [Tecuci *et* Hieb,96].

- L'expert peut aussi choisir de retourner à la phase d'instruction classique pour modifier directement le code des acteurs.

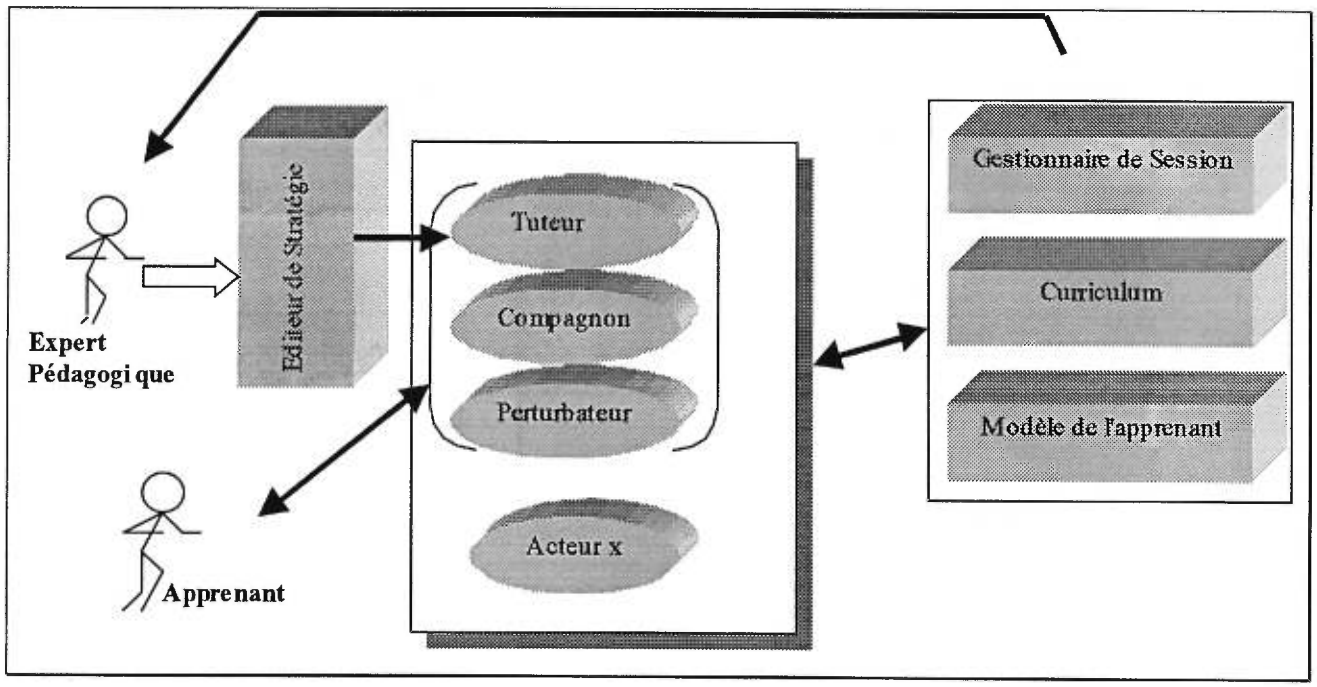

Figure 2. La formation des Stratégies

On peut remarquer que la présence d'un apprenant n'est pas toujours nécessaire. En effet,

comme les acteurs sont cognitifs, ils peuvent apprendre à simuler l'apprenant. Ainsi, nous pouvons remplacer l'apprenant par un agent dans le cycle de production et accélérer le processus de création de stratégies.

Le cycle de production résoud le problème d'acquisition d'expertise pédagogique dans un STI [Mengelle *et* Frasson, 96][Mengelle *et al.*,97]. En donnant aux acteurs les propriétés de cognition et d'éducation, ils peuvent commencer avec une expertise simple être enrichis par la suite. Il faut aussi pouvoir modifier facilement les stratégies (i.e. leur composition), les réutiliser d'un STI à l'autre et d'une ressource à l'autre.

2.3 Les Acteurs

Nous décrivons ici à quoi ressemble un agent dans SAFARI. Nous discutons de son architecture et des différentes propriétés d'agent dont il est doté.

2.3.1 Définition

[Frasson *et al.*, 96] définit un acteur comme un agent réactif, éducable, adaptatif et cognitif. Par les propriétés proposées en 2.1.2 p.9, les acteurs sont aussi sociaux.

2.3.2 Architecture d'un acteur

Comme expliqué dans [Mengelle *et* Frasson, 96], l'architecture d'un acteur (Figure 3) est assez générale pour servir à plusieurs formes d'expertises, mais une expertise pédagogique est visée,entres autres pour la construction des modules pédagogiques.

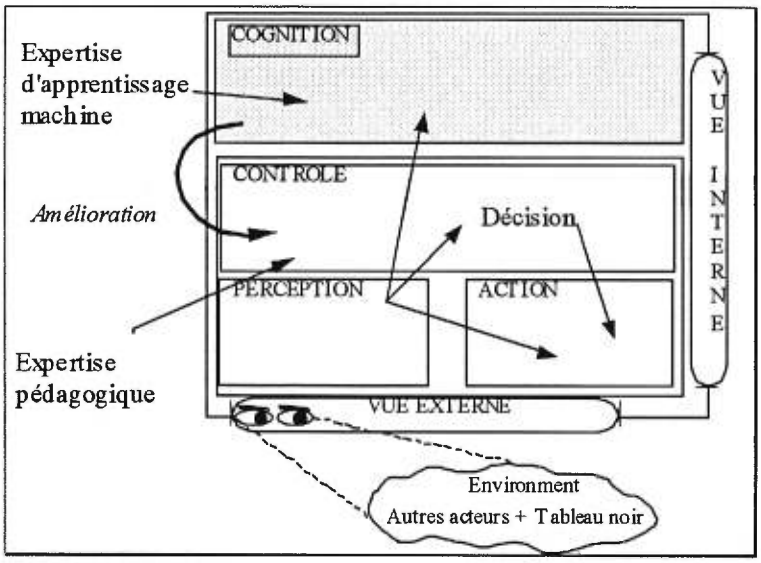

Figure 3. Architecture d'un acteur, extrait et adapté de [Mengelle et al.,97]

Elle est réalisée par deux couches: la couche d'expertise, qui contient les différentes connaissances pédagogiques de l'acteur, et la couche cognitive, qui permet à l'acteur d'apprendre.

2.3.2.1 La couche d'expertise

Elle contient trois modules: la perception, l'action et le contrôle. Le module de *perception* s'occupe de déterminer quand l'acteur peut **réagir** à une certaine situation. Le module de *contrôle* contient des connaissances permettant d'analyser la situation plus profondément pour pouvoir prendre une **décision** (faut-il donner la réponse ou une suggestion à l'apprenant, par exemple). Le module d'action contient les différentes actions que l'acteur peut utiliser pour manipuler l'environnement.

Le module de perception contient un ensemble de *situations types*. Elles vont contenir une condition basée sur l'environnement de l'acteur (tableau noir, actions passées des différents acteurs), ainsi que la *tâche* (ensemble d'instructions pour accomplir un travail) qu'il faudra déclencher lorsque la condition sera vérifiée. Ces situations-types seront constamment évaluées.

Dans cette couche, des objets (structures) sont définies pour permettre d'échanger des informations. Ces structures permettent de définir des messages envoyés à d'autres modules du STI (nous en verrons un exemple avec les ressources 3.2.4.4, p.46).

2.3.2.2 La couche cognitive

Cette couche est composée de tâches *cognitives* qui vont chercher à améliorer les performances de l'acteur. Elle va contenir des outils d'apprentissage machine permettant d'acquérir de l'expérience à partir des différentes situations d'apprentissage avec différents apprenants. Cette couche permet entres autres d'établir les propriétés d'adaptation, de cognition et d'éducation.

2.3.2.3 Les propriétés des acteurs

D'une combinaison des modules de perception de d'action, les acteurs obtiennent la propriété réactive. En effet, la combinaison de ces modules représente des tâches déclenchées à partir de certaines conditions de l'environnement. Ces tâches vont manipuler l'environnement des acteurs. Les situations types doivent être évaluées même lorsqu'une autre tâche est exécutée. Ceci implique l'introduction de parallélisme et l'interruptabilité des tâches.

[Mengelle *et al.*, 98] explique que le module de contrôle détermine le comportement social des acteurs, c'est-à-dire, des tâches de planification, décision et demandes de services à d'autres acteurs. Deux protocoles de communication y sont proposés: l'utilisation du tableau noir pour émettre une information ou une requête, ou la possibilité d'envoyer une requête directement à un ou plusieurs acteurs. Les acteurs possèdent donc un langage simple de communication et peuvent alors se voir attribuer la propriété de sociabilité.

2.4 Le problème

Le problème auquel s'adresse ce mémoire est de faciliter le développement du module pédagogique en utilisant des sociétés d'acteurs. À chaque société d'agent correspond une stratégie pédagogique possible. SAFARI a plusieurs exigences envers les acteurs:

- On leur veut des caractéristiques particulières (réactif, autonome, communiquant, adaptatif (ou apprenant) , cognitif et éducable) de façon qu'ils puissent être modifiés facilement (par un expert pédagogique au moyen de la propriété d'éducation pour faciliter le transfert de l'expertise, au moyen d'un éditeur, et éventuellement par eux-mêmes de par leur propriété de cognition);
- Ils doivent suivre une architecture clairement définie (d'où leur nom d'acteur);
- On doit pouvoir les ajouter et les retirer facilement des stratégies pédagogiques;
- Ils peuvent vérifier leur situation types même s'ils sont en train d'exécuter une tâche;
- Ils ont à exécuter plusieurs tâches en même temps que d'autres agents;
- Les partenaires industriels de SAFARI ont demandé que le module des acteurs soit écrit en C++ portable sur plusieurs plateformes. Cette contrainte interdit alors d'utiliser tout librairie C permettant de gérer le parallélisme des tâches des acteurs en utilisant des threads (processus parrallèles à l'intérieur d'un même programme).

Pour remédier au problème et aux contraintes ci-dessus mentionnés, nous avons choisi de construire un langage interprété (ainsi que l'interprète appelé Moteur ADL qui est écrit en C++), suivant l'architecture, les besoins et les contraintes des acteurs. Nous le voulons assez simple pour pouvoir spécifier rapidement des acteurs, les modifier facilement, et les exécuter dans l'architecture SAFARI. De plus nous voulons qu'il soit fidèle à l'architecture des acteurs pour en réduire et faciliter leur expression. L'interprète doit assurer une gestion des tâches parallèles. L'expérience a montré que l'introduction de tâches parallèles permet de simplifier le code des acteurs.

Chapitre 3

Le langage ADL

Le chapitre 2 donne les raisons du choix de créer un langage interprété. ADL (pour Actor Description Language) est ce langage servant à spécifier des acteurs. L'objet de ce chapitre est de décrire ce langage, et de montrer comment il répond aux différentes exigences de l'architecture des acteurs et de SAFARI.

Une comparaison entre ADL et KQML [Finin *et al.*,94] montre que le langage ADL est spécialisé. Il couvre les différents aspects de l'architecture des acteurs proposée au chapitre 2 et offre en plus une transparence avec les différents modules de SAFARI.

L'interprète du langage (moteur ADL) intègre les acteurs aux autres composantes de SAFARI, soit les ressources, l'interface graphique avec l'apprenant, le modèle de l'apprenant et le gestionnaire de session. De plus, le projet SAFARI demande à ce que le moteur ADL soit écrit en C++ portable (rappelons que c'est une contrainte des partenaires industriels). Or, cette contrainte empêche une implémentation facile du parallélisme des acteurs. Le moteur gère donc lui-même ce point.

3.1 Comparaison de ADL avec KQML

Dans la littérature sur les agents, dès qu'on parle de langage d'agent, on cite KQML [Finin *et al.*,94]. Une description rapide de ce langage montre qu'il n'est pas applicable pour les acteurs.

KQML veut dire Knowledge Query and Manipulation Language. Ce langage est un protocole d'échange d'informations et de connaissances entre agents. Il définit le format

de messages et la façon de les traiter. Ce langage est né de l'emploi des agents pour gérer un système d'information distribué. En effet, un service qui peut prendre l'initiative pour nous présenter certaines informations est souhaité. Le modèle client-serveur retrouvé typiquement dans Internet ne permet pas ce type de comportement. De plus, il existe plusieurs types de plateformes, format de données et services d'information. KQML sert à faciliter l'établissement d'une communauté d'agents pouvant offrir ces services.

Ces agents doivent pouvoir communiquer avec un langage, travailler en commun pour atteindre un but (une tâche par exemple), prendre l'initiative, et pouvoir gérer les ressources locales pour pouvoir répondre aux requêtes des autres agents. Nous avons donc affaire ici à des agents dont la propriété sociale est prédominante.

KQML définit des protocoles de communication. Les différents agents qui vont communiquer ensemble peuvent être codés en plusieurs langages (notamment en C et Lisp) et être exécutés sur des plateformes bien différentes. Plusieurs laboratoires ayant des projets différents s'en sont servis. Ce langage est un standard de communication entre agents sur plusieurs plateformes permettant une transparence des différents services qu'ils peuvent offrir.

Dans le projet SAFARI, il ne s'est pas avéré nécessaire de faire fonctionner les acteurs sur plusieurs ordinateurs à travers un réseau. Ainsi, KQML est inutile et ardu à utiliser pour l'implantation d'acteurs.

Le langage ADL est un langage interprété dont le but est de faciliter l'implantation d'un type particulier d'agent appellé acteur. Ce langage est spécialement pensé pour épouser l'architecture des acteurs et pour permettre certaines contraintes, tel que la possibilité de modifier facilement le code à l'aide d'un éditeur ou à la main, voire même par l'acteur lui-même (avec sa couche cognitive). Ce langage est interprété par un moteur qui est portable d'un système d'exploitation à l'autre.

En conclusion, KQML est un protocole général de communication d'agents, alors qu'ADL est un langage de description d'un type particulier d'agent, les acteurs. KQML s'intéresse ainsi à un problème différent (et plus général) d'ADL et ces deux langages n'ont en commun que de l'usage des agents.

3.2 *Les acteurs en ADL*

Dans cette section, nous voyons comment nous traitons les différents aspects de l'architecture des acteurs à travers le langage ADL. La grammaire complète des acteurs est donnée en annexe (Annexe A).

3.2.1 La grande variété des acteurs et des stratégies pédagogiques

Une stratégie pédagogique est composée d'un certain nombre d'acteurs. Voici quelques exemples d'acteurs [Frasson *et* Aïmeur, 96]:

- tuteur: professeur qui pose les questions et qui corrige les réponses;
- apprenant: simulation d'un étudiant. Cet acteur peut être utile lors de la phase de test de stratégie dans le cycle de fabrication de stratégie pédagogique (voir 2.2.4 p.20);
- compagnon: étudiant accompagnant l'apprenant. Il cherche à l'aider, pose des questions et émet des suggestions;
- perturbateur: compagnon qui cherche à dérouter l'apprenant par des suggestions bien choisies (vraies ou fausses);
- cotuteur: tuteur qui doit travailler avec un autre tuteur complémentaire pour donner un cours dont il ne connaît qu'une partie de la matière;
- superviseur: acteur qui peut corriger un tuteur.

On peut construire différentes stratégies pédagogiques par combinaison de ces acteurs. En voici quelques exemples:

- un tuteur et un compagnon (stratégie du compagnon);
- un tuteur et un perturbateur (stratégie du perturbateur);

- un tuteur et un cotuteur (stratégie de cotutorat);
- un tuteur et un superviseur (stratégie de supervision);

Ces différentes stratégies ont chacune une variante où un acteur apprenant est ajouté. Cet acteur simule un véritable apprenant. Ces variantes sont très utiles pour tester les stratégies dans le cycle de production de l'éditeur de stratégies par exemple (voir 2.2.4 p.20).

Ces différents acteurs ont tous des expertises bien différentes. Comme nous voulons tous les coder à travers un même langage (ADL), ce dernier se doit être général et ne s'appliquer qu'à l'architecture des acteurs.

3.2.2 Structure et moyens de communication

Les acteurs sont tous structurellement identiques: ils communiquent tous entre eux de la même façon, et possèdent la même structure. Seuls leur comportement et leurs connaissances sont différents.

Un acteur se divise en cinq composantes principales (Figure 4):

- perception: collection de *situations types* déclenchant des *tâches*. Un outil y permet de restreindre les informations accessibles à l'acteur: le *focus général*. Cette composante examine dans une trace des actions passées de l'acteur (appelée vue interne), celle des autres acteurs, et le *tableau noir*. Elle est aussi sensible à des événements venant des ressources (i.e. actions de l'usager);
- action: ensemble de *tâches opératoires, tâches abstraites*. Ce sont les actions de base d'un acteur;
- contrôle: ensemble de *tâches de contrôle*. C'est ici que se prennent les décisions (donner un indice ou la réponse à l'apprenant à la suite d'une erreur par exemple);
- cognition: ensemble de *tâches de cognition*. Ces tâches assurent le bon fonctionnement de la couche cognitive, dont les propriétés de cognition et d'éducation des acteurs. Certaines de ces tâches peuvent ainsi utiliser des outils d'apprentissage

machine et peuvent communiquer avec le pédagogue lors d'apprentissage de nouvelles tâches dans le cycle de production de stratégies (2.2.4 p. 20);

- vue interne (ou comportement passé pour la session): trace des *tâches* accomplies.

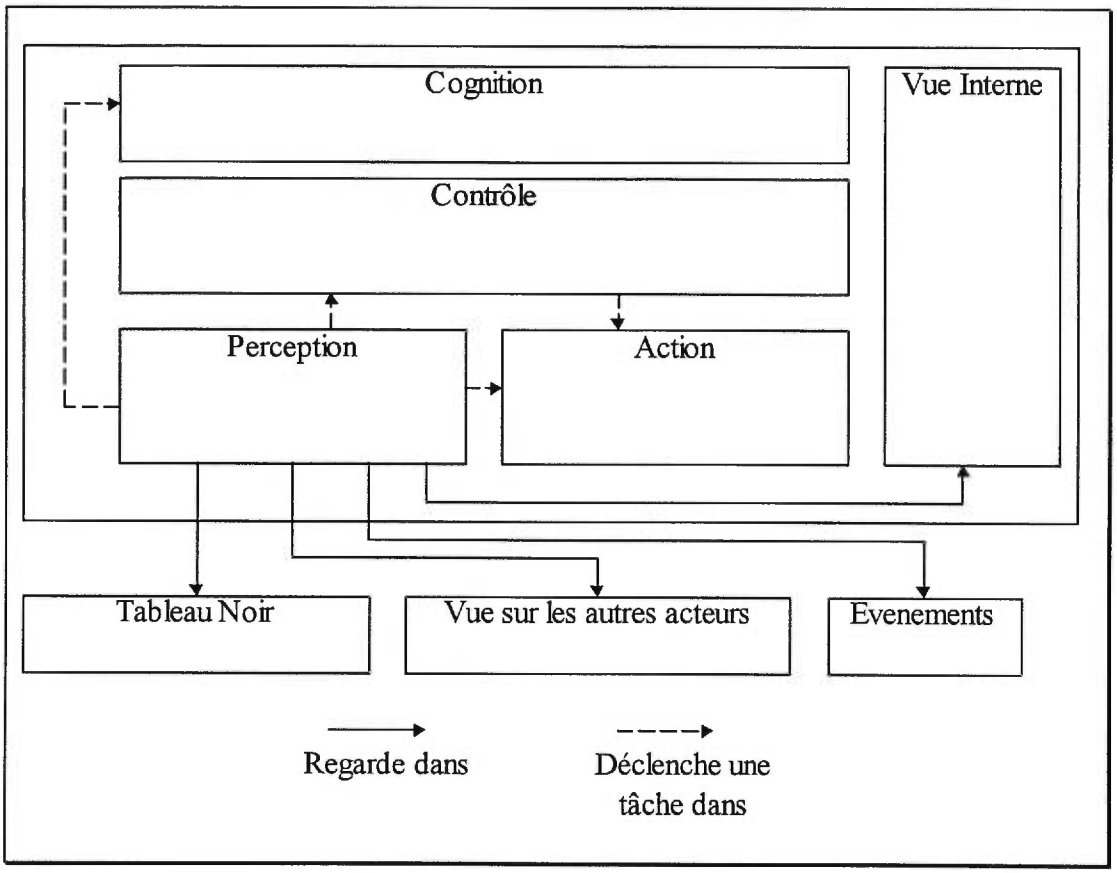

Figure 4. Les composantes d'un acteur

Rappelons que les *tâches* sont interruptibles et peuvent fonctionner en parallèle. Il y a quatre sortes de *tâches*

- opératoire: c'est une séquence de primitives. Répondre "non, la réponse est 4" est une *tâche opératoire*;
- abstraite: Une séquence de *tâches abstraites*, opératoires et/ou primitives. "Mentir" en est une;
- contrôle: *Tâche abstraite* qui peut appeler d'autres *tâches de contrôle*, avoir des variables locales et des fonctions de contrôle (if then else, while ...);

- cognition: *Tâche de contrôle* pouvant appeler une autre *tâche de cognition*.

Les acteurs possèdent deux moyens de communication (Figure 5)

- *tableau noir*: informations globales et accessibles à tous les acteurs;
- *requêtes*: un acteur peut demander un service à un autre. Ce dernier peut refuser de rendre ce service.

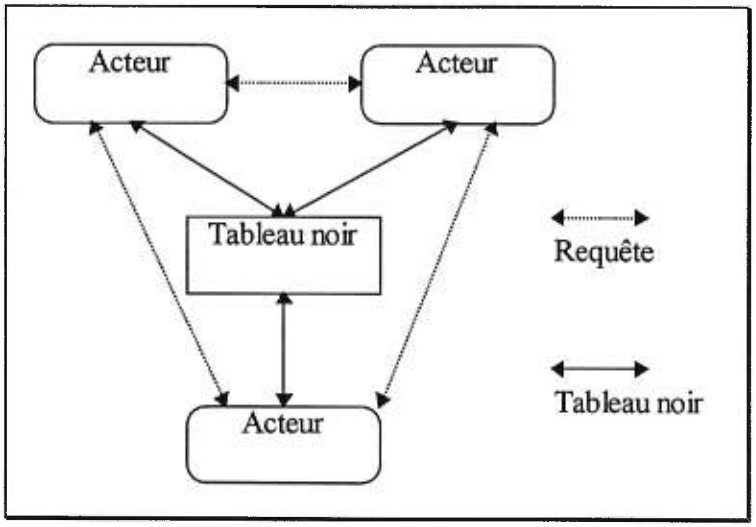

Figure 5. Types de communication

3.2.3 Implémentation des acteurs

Nous voyons ici l'implantation des différentes composantes des acteurs, ainsi que les définitions et restrictions nécessaires. Nous discutons d'abord les sources d'informations directes dont disposent les acteurs pour leur fonctionnement: le tableau noir, les vues sur les autres acteurs, les événements et les variables globales. Nous voyons ensuite comment les acteurs activent des tâches à partir des ces informations. Viennent ensuite les *requêtes*, sources d'informations supplémentaires et déclencheurs de *tâches* pour obtenir des services. Nous finissons par la description des tâches.

Dans les descriptions, nous présentons les différentes primitives du langage ADL permettant d'accomplir les différentes fonctionalités voulues.

3.2.3.1 Sources d'information directes

Un acteur a accès à quatre sources d'informations de façon directe:

- tableau noir: zone d'information accessible à tous en lecture et en écriture;
- vue sur les autres acteurs: une trace de toutes les tâches qu'ils ont exécutées;
- liste d'événements: chaque tâche d'un acteur et chaque action de l'apprenant génère un événement qui est ajouté à une liste;
- variables globales: chaque acteur peut avoir des variables globales et une mémoire associative qui seront partagées par les différentes tâches qu'il exécutera. La mémoire associative est une forme de tableau noir privé pour l'acteur.

Le tableau noir

Le tableau est une collection d'informations avec une clé composée d'un titre, du nom de l'acteur qui a déposé cette information dans le tableau noir et d'une catégorie. Chaque élément ressemble à ceci: **<titre> <nom d'acteur><catégorie><information>**

Les fonctions pour le manipuler sont:

- **putbb** <information> as <titre> [category <catégorie>]

cette fonction ajoute un élément supplémentaire dans le tableau noir. Si un même acteur met deux fois une information sous le même titre et la même catégorie, il y a écrasement de la dernière information. **<information>** est une information quelconque. **<titre>** est le nom (chaîne de caractères) de l'information. **<catégorie>** est un nom supplémentaire pour le classement. La catégorie est optionnelle. Sa valeur par défaut est "NoCategory";

- **getbb** <titre>|* [from <nom d'acteur>] [category <catégorie>]

cette fonction trouve le premier élément dans le tableau noir correspondant aux critères de sélection, le retire et renvoie l'information qui lui est associée. Si aucune information n'est trouvée, le résultat sera *nil*. Un "*" à la place du titre indique qu'il n'est pas un critère de sélection. Il en est de même pour l'absence d'un nom d'acteur. Ne pas spécifier

de catégorie revient à chercher dans la catégorie "NoCategory", qui est la catégorie des informations sans catégories. La catégorie "AllCategories" indique qu'elle n'est pas un critère de sélection;

- **lookbb** <titre>|* [from <nom d'acteur>] [category <catégorie>]

fait la même recherche que **getbb**, sauf que l'information n'est pas retirée du tableau noir;

- **existbb** <titre>|* [from <nom d'acteur>] [category <catégorie>]

fait la même recherche que **getbb**, sauf que la primitive renvoie un booléen indiquant l'existence d'une telle information sans la retirer du tableau noir.

Vue sur les acteurs

Chaque acteur conserve une trace des tâches qu'il a exécutées et laisse la possibilité aux autres de la consulter. Une trace est une liste de blocs d'information de tâche. Le début de chaque tâche est marqué de façon unique dans la trace des acteurs (par un entier qui augmente d'un à chaque nouvelle tâche par exemple). La trace ne sert qu'à connaître l'ordre du déclenchement des tâches d'un acteur à l'autre.

Chaque bloc correspond à l'activation d'une tâche. Ce bloc contient:

- le temps en ordre de déclenchement et en secondes où la fonction a débuté (la référence étant le déclenchement de la stratégie);
- une liste d'informations sur les sous-tâches appelées;
- les paramètres donnés à cette tâche;
- le nom de la tâche;
- un booléen indiquant si la tâche est terminée;
- un booléen indiquant si la tâche a été un succès ou non (i.e. elle a généré une erreur);
- un booléen indiquant si cette tâche a été activée par une *situation type* ou par une *requête*. Ce dernier champ sera caché et ne servira que lors du filtrage par le focus.

Un acteur n'est pas supposé voir tout ce que les autres acteurs ont déjà fait. Pour permettre un apprentissage cognitif, un acteur ne doit voir que ce qu'un autre acteur a fait, pas les décisions qu'il a pris pour accomplir les tâches dans la trace. Un acteur a un apprentissage réaliste (i.e ressemblant à celui d'un humain). Or, un humain n'est jamais au courant des décisions prises des autres. Il ne peut que les déduire à partir des actions qu'il a observées. Ainsi, nous mettons un filtre. Nous l'appelons le *focus*. Sa forme détaillée est décrite plus loin. Un acteur a un *focus général* qui détermine ce que voit l'acteur dans les vues des autres acteurs.

Sous la restriction du *focus général*, un acteur peut utiliser ces fonctions dans toute tâche et dans les évaluateurs:

- **lookact** [<liste de noms d'acteurs>] [for <filtre de tâches>]
[jumptolast | jumpto <temps>][notended]

où **<liste de noms d'acteurs>** indique quels sont les acteurs recherchés. Par défaut, tous les acteurs le sont. **<filtre de tâches>** est une liste comportant une combinaison de type de tâche, soit: **operating** | **abstract** | **control** | **cognitive task**. Ce critère de recherche indique quelles tâches sont voulues. Par défaut, toutes les tâches le sont. **jumpto <temps>** indique qu'il faut tout ignorer ce qui vient avant le temps <temps>. **jumptolast** est un mot clé demandant de regarder à partir du dernier temps que notre acteur a visité par **lookact** ou **multilookact**. Par défaut, si **jumpto** et **jumtolast** ne sont pas spécifiés, la recherche commence à partir du temps le plus récent. **notended** indique que les tâches inachevées sont aussi voulues. Cette fonction va retourner la tâche la plus récente qui remplira les critères de recherche en parcourant la trace. Elle renverra *nil* si elle ne trouve rien. Il faut préciser ici que la trace récupérée peut omettre certaines informations. En effet, si la tâche appelante n'est pas dans le focus, la trace annoncera que cette tâche n'est pas une sous-tâche (directement appelée par une *requête* ou une *situation type*);

- **gettracetaskparam**

le paramètre de cette primitive est ce que retourne un appel de **lookact**. Elle donne les

paramètres ayant servi à l'appel de la tâche sous forme de liste;

- **gettracetime** <élément de trace>

le paramètre de cette primitive est ce que retourne un appel de **lookact**. Elle indique le temps de l'élément de trace. Ce temps peut servir à un appel futur de **lookact** (paramètre **jumpto**);

Les événements

Les actions de l'usager et le déclenchement de tâches sont annoncés aux acteurs par la voie d'événements. Ces derniers sont représentés par des structures. Ceux envoyés par la ressource dépend du protocole du communication (la définition des structures correspondantes se retrouve dans le code ADL), et le déclenchement de tâches est annoncé par une structure prédéfinie appelée *task*, qui contient les trois champs *name*, *actor* et *id*, contenant respectivement le nom de la tâche, celui de l'acteur, ainsi que l'identificateur de temps pour la tâche.

Un acteur peut les filtrer et accéder aux événements avec les primitives suivantes.

- **seteventfilter** {<filtre1>,<filtre2> ... <filtre>}

cette primitive détermine quels sont les événements que l'acteur ne veut pas pouvoir détecter avec la fonction **getevent**. Par défaut, tous les événements sont détectés. Les événements dont les noms sont dans le filtre d'événement sont ignorés par l'acteur;

- **getevent**

récupère l'événement détecté avant la dernière évaluation des situations types. Si aucun événement n'a été détecté, cette primitive retourne *nil*. Elle ne peut pas voir les événements éliminés par **seteventfilter**;

- **matchevent** [<var> =] <type> , [<var> =] <type> ... [<var> =] <type>

cette primitive permet de vérifier si les derniers événements sont des structures ayant les

types <type> tout en respectant l'ordre énoncé. Elle retournera un booléen annonçant si c'est le cas ou non. Si oui, pour tout <type> précédé d'un <var> =, la variable <var> se trouvera l'événement correspondant. Si <type> est "*", tout type conviendra. Les événements vus ne seront que les tâches pouvant passer par le *focus* de l'acteur et par le filtre d'événements (voir **seteventfilter**);

- <expression> **istype** <type>

<type> est un type de structure défini dans le code, ou une des constantes suivantes: **integer**, **boolean**, **string** et **list**. Les dernières constantes tiennent pour un entier, un booléen, une chaîne de caractères et une liste (A.1 p. A-1 pour voir la description des types). Cette primitive permet de vérifier si une valeur est d'un certain type. Elle sert entre autres pour déterminer le type d'un événement (i.e. le type de la structure);

- l'extraction d'un champ de structure. L'extraction d'un champ se fait avec le point (.).

À gauche du point se trouve une variable contenant la structure. À droite, le nom du champ. Ce nom n'est pas délimité par des guillemets. Exemple: MaVariable.UnChamp. Une telle expression est considérée comme une variable.

Les variables globales

Un acteur a deux sortes de variables:

- les variables globales: pouvant contenir une valeur à la fois.
- variables de mémoire associative: ressemblent beaucoup à un tableau noir simplifié et privé. Nous pouvons y mettre des informations associées à des clés.

Ces variables sont manipulées par les primitives suivantes:

- **getglobalvar** <nom>

donne le contenu d'une variable;

- **setglobalvar** <nom> to <expression>

assigne le résultat d'une expression <expression> à la variable <nom>;

- **putmem** <entier> as <titre> in <nom de variable de mémoire>

ajoute un entier dans la mémoire associative de l'acteur;

- **getmem** <titre> in <nom de variable de mémoire>

renvoie l'entier placé par un appel de **putmem** avec les mêmes paramètres et retire cette information de la mémoire;

- **lookmem**

fait la même chose que **getmem** sauf que l'information n'est pas retirée de la mémoire.

3.2.3.2 La perception et les situations types

Chaque acteur peut voir dans le tableau noir et dans l'historique des autres. Dans certains cas, l'acteur aura à réagir à un nouvel état du monde qu'il voit. Le mécanisme de détection et de réaction se trouve dans la partie perception de l'acteur. Pour chaque situation type, un focus supplémentaire est ajouté. Le *focus général* est appliqué sur la vue des autres acteurs avant toute évaluation de situation type. Ainsi, seule la vue filtrée par les deux focus sera prise en compte par les situations types (Figure 6).

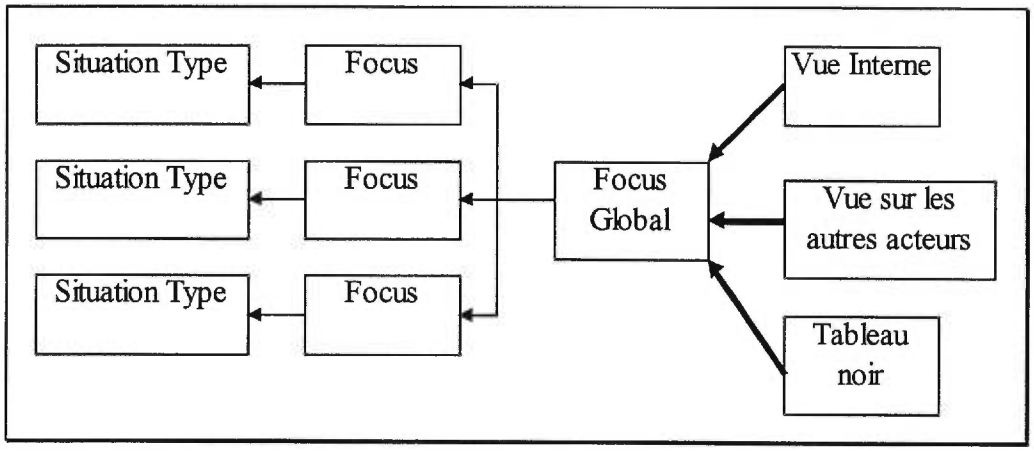

Figure 6. La vue filtrée par deux focus

La description d'un focus est donnée dans celle de la situation type. Nous allons donc nous attarder sur ce dernier point. Une situation type est composée d'un triplet:

- évaluateur
- focus
- alias de tâche.

Évaluateur

Un évaluateur est un bloc de code dont l'évaluation donne un booléen indiquant si la situation type est vérifiée ou non. Il est interdit d'y faire appel à des *tâches* ni faire de *requêtes*. En effet, une situation type indique si une tâche doit être déclenchée. Déclencher une tâche à l'intérieur de l'évaluation de la condition nuit au but même de la situation type.

Focus

Un focus indique quelle est la limite du temps avant laquelle nous ne sommes pas intéressés à voir une trace de tâche, ainsi que les tâches ou les types de tâches qui ne nous intéressent pas. Chaque acteur a son propre focus (appelé *focus général*). Cet outil permet de simplifier l'utilisation de la trace en ignorant certains éléments. Rappelons aussi que le focus devient important pour un apprentissage cognitif.

Le focus de la situation type est un focus supplémentaire qui filtre ce qui est passé à travers le *focus général*. Seul ce qui passera par ce second focus sera pris en considération par l'évaluateur (décrit plus haut). Nous pouvons remarquer aussi que les focus ont deux autres effets sur la perception:

- Les fonctions de recherche dans l'évaluateur sont allégées: il n'est pas nécessaire de répéter sans arrêt le focus à chacune d'entre elles.
- Elles iront plus rapidement: en effet, le focus aura pour effet d'enlever beaucoup d'informations inutiles, accélérant ainsi leur fonctionnement.

Un focus s'intéresse à deux choses:

- Le temps

- Le niveau de perception (sécurité) dans les autres acteurs.

Le temps

L'expression du temps a deux formes:

- begin (aucune restriction temporelle)
- <nom d'acteur> did **<positionnement>** **<description de tâche>**

où **<positionnement>** a une de ces formes:

- first: le temps de la première tâche correspondant à la description
- last: le temps de la dernière tâche correspondant à la description
- <un entier X> times: le temps de la X^e tâche correspondant à la description. Si X a une valeur de 1, c'est l'équivalent de faire "first".

et **<description de tâche>** a une de ces formes:

*: toute tâche correspond à cette définition

un *alias de tâche*

<type de tâche>: operating | abstract | control | cognitive task

Le niveau de sécurité

Elle s'exprime par une liste d'expressions allant comme suit:

<nom d'acteur> at <type de tâche> [request level]

<nom d'acteur> unique à chaque expression. Indique quel acteur dont la vue nous intéresse.

<type de tâche> operating | abstract | control | cognitive task. Profondeur à laquelle un acteur a le droit d'en regarder un autre. Généralement, il ne faut pas dépasser le niveau "operating" en ce qui concerne d'autres acteurs. Un niveau inclut tous ceux qui sont en dessous.

request level est optionnel. Les tâches vues sont celles déclenchées par des situations types ou par d'autres tâches. Cette option indique qu'il faut les tâches qui ont

été déclenchées par des requêtes.

Si jamais un acteur nommé n'est pas dans la distribution, ce n'est pas une erreur.

L'alias de tâche

Un alias de tâche est une référence mobile (pointeur) à une tâche. Il est désigné par une chaîne de caractères. Par contraste, un identificateur de tâche est une référence fixe à une tâche. Par défaut, l'identificateur est aussi considéré comme l'alias. L'acteur doit garder une liste des noms de tâches et les différents alias pour que le tout fonctionne correctement.

Dans le cas d'une situation type, notre alias doit faire référence à une tâche sans paramètres.

3.2.3.3 Les requêtes

Il peut arriver qu'un acteur ait à demander un service à un autre. Pour ce faire, il fait la demande et attend la réponse. Cette réponse peut prendre un temps indéfini. Elle est soit un refus de rendre le service, soit le résultat du service après qu'il ait été rendu. Ainsi, sur une réponse, l'acteur sait à quoi s'en tenir.

Une *requête* ne peut être faite qu'à partir d'une *tâche*.

Nous allons voir les *requêtes* en trois parties:

- syntaxe des primitives liées aux requêtes;
- mécanisme d'envoi et de retour;
- mécanisme de traitement.

Syntaxe des primitives liées aux requêtes

Les requêtes sont manipulées par les primitives suivantes

- **req** <titre> [with <paramètres>][to <liste de noms d'acteurs>] [passive][ifno <que faire si refus>]

  <titre> est le nom de la *requête*. **<paramètres>** sont les paramètres éventuels qui peuvent servir au bon fonctionnement de la *requête*. Par défaut, il n'y a pas de paramètres. **to** est suivi d'une liste de noms d'acteurs à qui cette *requête* est envoyée. Par défaut, est envoyée à tous. **passive** indique si la réponse à cette *requête* est n'est pas impérative pour continuer la *tâche*. Par défaut, la *requête* est considérée comme active. **<que faire si refus>** indique ce qu'il faut faire si la *requête* est refusée. Voici les différents mots clés possibles:

  - **abort**: la tâche doit être arrêtée. Elle résulte en un échec;
  - **ignore**: continuer comme si rien n'était.;

- **getreply** <identificateur> [from <nom d'acteur>]

<identificateur> est ce retourne un appel à **req** lors d'appels passifs. Cette fonction émet le résultat d'une requête ou appel de fonction. Elle restera bloquante tant qu'il n'y aura pas de réponse;

- **ifno**

retourne un booléen indiquant si la dernière requête a été refusée;

- **existreply** <identificateur> [from <nom d'acteur>]

indique s'il y a une réponse à une requête. Elle n'est pas bloquante: elle retourne false s'il n'y a pas de réponse.

Mécanisme d'envoi et de retour

Le bon déroulement de l'émission de la *requête* est assumé par le moteur ADL. Ce dernier propose la *requête* à tous les acteurs concernés. Deux cas peuvent arriver;

- il y a un seul destinataire. S'il existe, il traite la requête. Sinon, le moteur ADL refuse la requête au nom de ce destinataire inexistant et l'annonce à l'acteur demandant le service (la requête);

- il y a plusieurs destinataires. Le moteur leur demande s'ils reconnaissent cette

requête. Le premier qui la reconnaît répond à cette requête. Si aucun acteur ne peut répondre, le moteur refusera la requête au nom de "AllActors".

Mécanisme de traitement

Lorsqu'un acteur doit reconnaître une *requête*, ce dernier examine une table de requêtes auxquelles il est capable de répondre (un dictionnaire). S'il le reconnaît, il déclenche une tâche pouvant répondre à la requête. Une fois la tâche terminée, il retourne le résultat à l'acteur qui a fait la requête.

3.2.4 L'intégration avec les autres modules de SAFARI

Le langage veut donner une intégration transparente avec les autres modules de SAFARI. Pour ce faire, la communication avec les autres modules fait partie intégrante du langage. Les acteurs peuvent dialoguer avec les autres modules de SAFARI à travers des primitives contenues dans une librairie de services externes (Figure 7). Il suffit qu'un programmeur mette à jour la librairie des services externes au fur et à mesure que d'autres modules s'ajoutent à SAFARI ou que les services changent.

Le moteur est intégré à trois modules externes: une interface graphique, les ressources (adidactiques), et le modèle de l'apprenant.

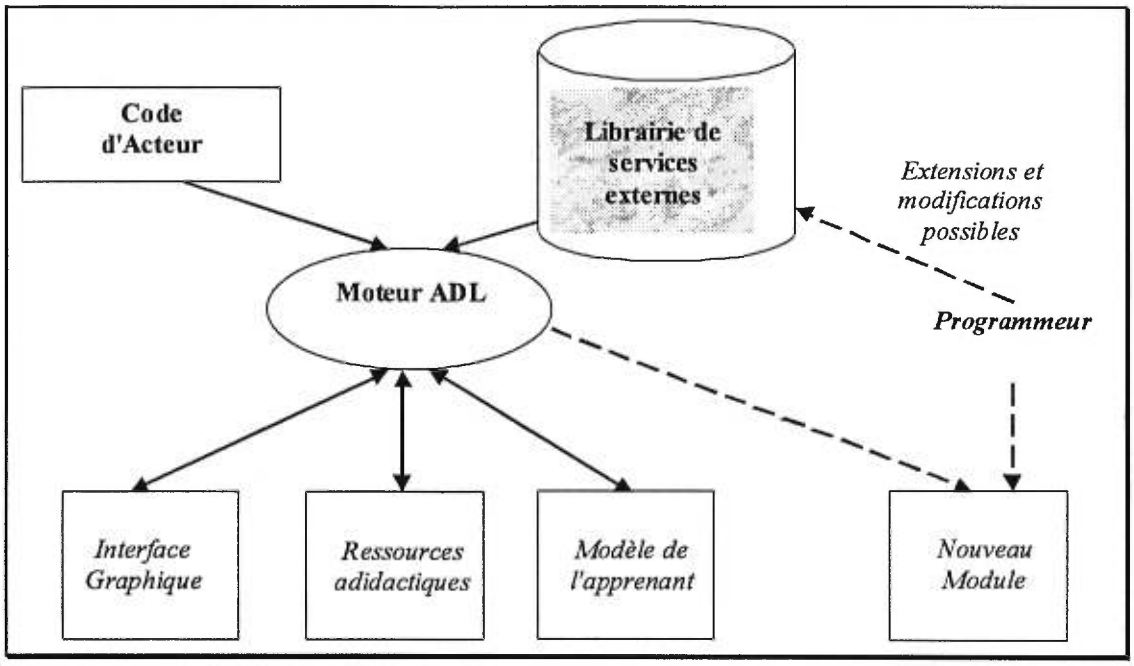

Figure 7. Intégration du moteur ADL avec autres modules de SAFARI

3.2.4.1 L'interface graphique

L'interface graphique s'occupe d'afficher les acteurs ainsi que leurs actions. Ils peuvent ainsi dialoguer avec l'apprenant et changer leurs expressions. Nous avons alors les primitives suivantes:

- **ask** <question>

demande une question à l'apprenant de la part de l'acteur. La valeur de retour sera un booléen si l'apprenant répond (oui ou non) et *nil* sinon;

- **say** <message>

annonce que l'acteur veut donner un message à l'apprenant. Le texte contenu dans <message> lui sera transmis;

- **setexpression** <expression>

change l'expression de l'acteur. Les différentes expressions possibles ne sont pas définies dans ce langage. Elles le sont dans le module de l'interface graphique des acteurs.

3.2.4.2 Les ressources

Pour dialoguer correctement avec les ressources, nous avons établi un protocole de communication à travers des événements qui correspondent à l'arrivée d'une structure (voir 3.2.4.4, p. 46). Les primitives utilisées pour recevoir des événements sont décrites en 3.2.3.1, p.32.

Pour envoyer des messages aux ressources, les acteurs utilisent une autre primitive. Cette dernière génère un événement pour les ressources en leur envoyant une structure. Cet événement n'est pas visible pour les acteurs. Si le message demande une réponse (selon le protocole de communication avec la ressource), une structure est retounée, mais n'est pas perçue comme un événement pour les acteurs.

Seuls les diagnostics de la ressource sur les actions de l'apprenant seront perçus comme des événements. Les primitives nécessaires pour envoyer un message aux ressources sont:

- **make** <type>

<type> est le nom d'une structure déclarée dans le code. Ce nom n'est pas délimité par des guillemets. La fonction retournera une structure du bon type ou *nil* s'il n'existe pas;

- **send** <structure>

Cette primitive envoie la structure <structure> comme message à la ressource. Le type et le contenu de la structure contiennent le message et sa signification. Ce message est considéré comme un événement par la ressource. Si la ressource définit une réponse au message envoyé, cette primitive retourne la réponse, sinon elle retourne *nil*.

3.2.4.3 Le modèle de l'apprenant

Le modèle de l'apprenant possède des information cognitives, conatives et affectives, ainsi que ce que l'apprenant sait du cours (sous forme d'overlay du curriculum). Les primitives utilisées sont:

- **updatecapacity** <idcapacite> to <niveau>

met à jour une capacité <idcapacite> au niveau <niveau>. Ces champs doivent nous être donnés par la ressource à travers un événement;

- **updateobjective** [<idobjective> to] <niveau>

met à jour un objectif <idobjective> au niveau <niveau>. Par défaut, <idobjective> est l'objectif qui est à atteindre par les acteurs. Ces champs doivent nous être donnés par la ressource à travers un événement;

- **updateressource** [<idressource> to] <niveau> [context <contexte>]

met à jour une capacité <idcapacite> au niveau <niveau>. Par défaut, <idressource> est la ressource courante gérée par les acteurs. <contexte> peut contenir des informations sur l'état de la ressource (sauvegarde pour une session ultérieure). Ces champs doivent nous être donnés par la ressource à travers un événement;

- **getstudentinfo** attention | alert | anxious | interest | self-confidence | self-esteem | workmemory | generalknowledge | associativelearning | inductivereasoning | quickness | defaultknowledge | metacognition | impulsive | serial | verbal | passive | random | conformist

récupère l'information affective, cognitive ou conative chez l'apprenant. Les six premiers mots-clés (*attention* à *self-esteem*) donnent une information affective. Les sept suivants (*workmemory* à *metacognition*), une information cognitive et les six derniers (*impulsive* à *conformist*), une information conative. Les valeurs retournées sont des entiers de 0 à 10 indiquant de degré d'importance de l'information demandée sur l'apprenant;

- **getstudentknowledge** objective | resourcepassed | resourcescore | capacity | maxlevelcap <identificateur>

récupère de l'information sur les connaissances de l'apprenant envers le cours. Le premier paramètre indique le type d'information recherché dans l'information désigné par <identificateur>. Si le premier paramètre est *objective*, la primitive indique si l'objectif en

question a été atteint. Il en va de même avec *resourcepassed* pour une ressource. Avec *resourcescore*, un score de 0 à 10 indique la réussite de l'apprenant avec la ressource, et *maxlevelcap* indique le niveau maximum qu'une capacité peut atteindre.

3.2.4.4 Protocole de communication avec les ressources

Le protocole vu ici est un standard qui a été décidé vers la fin de la rédaction de ce mémoire. Même si aucun prototype ne l'a encore utilisé, nous avons choisi ce protocole parce qu'il généralise et étend les protocoles utilisés dans les différents prototypes développés dans SAFARI pour établir la communication entre acteurs et ressources. Un prototype utilisant une version ultérieure de ce protocole est donné cependant en annexe (Annexe D, p. D-32).

Rappelons que le but premier des acteurs dans SAFARI est de composer un module pédagogique pouvant gérer des ressources adidactiques. Ces ressources peuvent avoir des structures internes variées, mais doivent communiquer avec les acteurs selon un protocole bien défini [Mengelle *et al.*, 98]. Ce protocole, en Figure 8, comporte quatre messages:

- SendDescription(<type-ressource>, <domaine-local>)
- Inform(<diagnostic>,<reactions-possibles>)
- Load-Resource(<id-ressource>)
- ApplyDecision(<reaction>)

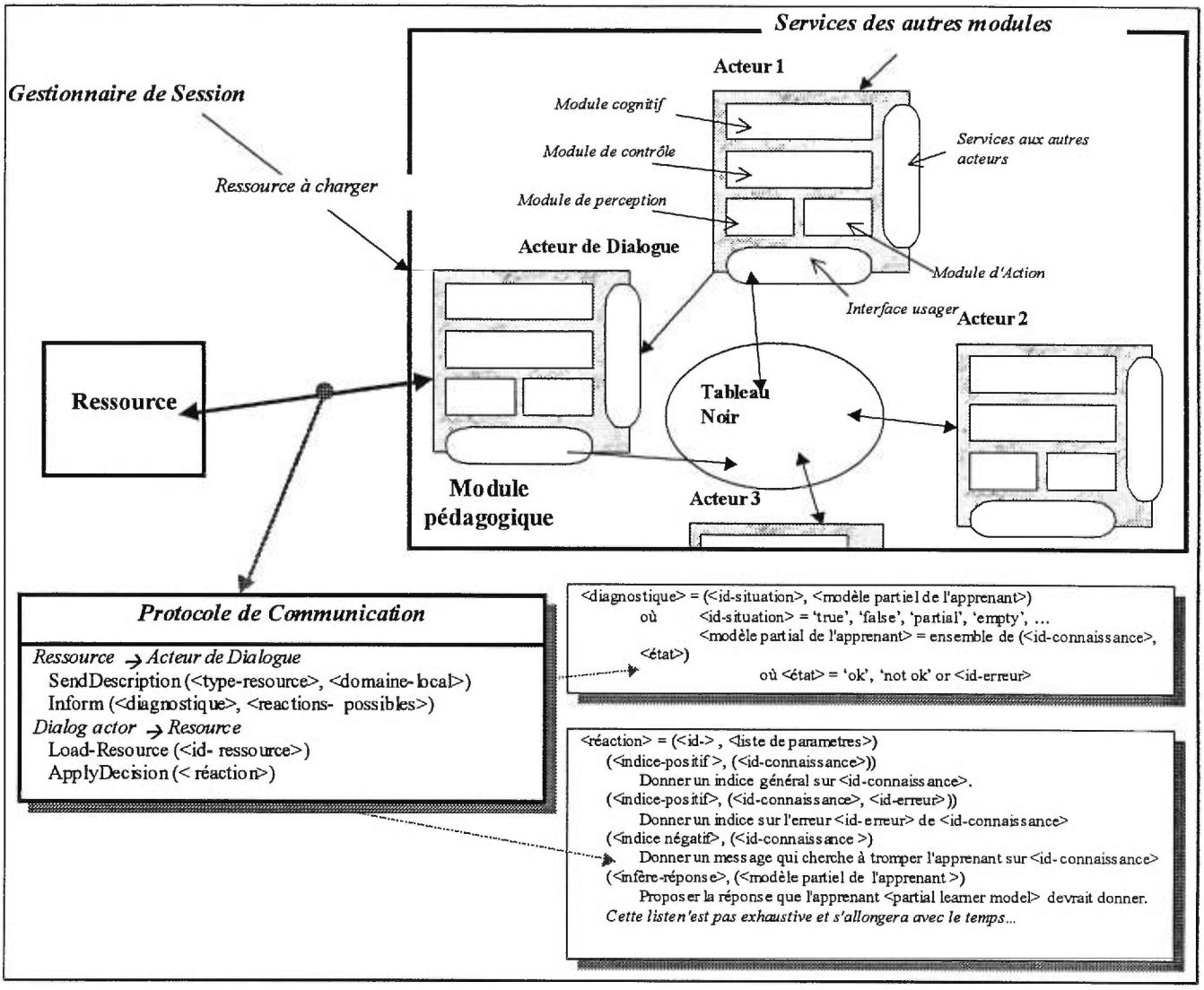

Figure 8. Protocole de Communication avec les ressources, adapté de [Mengelle et al., 98]

À chaque message correspond un type de structure dans le code des acteurs donnant le nom des différents champs. Lors d'échanges d'informations, les acteurs et les ressources se passent une structure d'un de ces types.

En premier lieu, un acteur envoie le message (ou structure) `Load-Ressource` où <id-

ressource> est la ressource que le gestionnaire de session demande de charger.

La ressource donne une réponse dans une structure `SendDescription`. <type-ressource> indique le type de la ressource (si le type de ressource est un QCM ou une page HTML par exemple), et <domaine-local> content une d'extension du curriculum pour avoir un contrôle raffiné sur la ressource. En effet, lors des échanges d'informations entre les ressources et des acteurs, des connaissances locales à la ressource qui n'ont pas été prévues dans le curriculum peuvent être concernées. <domaine-local> est alors un dictionnaire pour que les acteurs puissent dialoguer correctement avec la ressource.

Par la suite, le dialogue s'établit véritablement entre les acteurs et la ressource. À chaque action de l'apprenant, la ressource en informe les acteurs avec une structure `Inform`. Cette structure possède un diagnostic sur l'action de l'apprenant et donne les différentes réactions possibles. Le diagnostic décrit le type de réponse (bonne réponse, mauvaise réponse, réponse partielle ...) et décrit l'interprétation de ce fait en donnant une partie du modèle de l'apprenant. Ce modèle est une liste contenant l'état de chaque connaissance impliquée dans l'activité. Les états peuvent être *mastered*, *not-mastered*, (pour maîtrisé ou non), ou un identificateur d'erreur typique (*frequent mistake*). Les réactions possibles sont les actions que la ressource peut accomplir à ce moment-là pour continuer le cours. Ces réactions peuvent être de reformuler la question, donner un indice sur une connaissance donnée dans le diagnostic (un des <id-connaissance>), chercher à tromper l'apprenant en proposant en donnant un indice qui se trouve à être une erreur typique, donner la réponse, etc. Ce sont les acteurs qui décident quelles sont les actions à prendre à partir de celles que propose la ressource.

La réaction choisie est envoyée à la ressource à travers une structure `ApplyDecision`. La ressource répond par un texte que les acteurs ont à afficher dans leur interface graphique (l'indice demandé par les acteurs par exemple), ou bien par un code indiquant que l'action est accomplie par la ressource (en déplaçant le pointeur de la souris pour attirer l'attention de l'apprenant par exemple). Ces deux réponses peuvent aussi être combinées (les acteurs

donnent un indice et la ressource pointe là où l'indice s'applique).

Dans ce protocole, la ressource s'occupe de faire le diagnostic des actions de l'apprenant alors que les acteurs s'occupent de faire les décisions pédagogiques. Une telle encapsulation de l'expertise sur la ressource permet une véritable modularisation des acteurs et la possibilité de développer l'expertise de la ressource de façon indépendante. En fait, l'expertise du domaine local peut être représentée de toutes sortes de façons, tant que la ressource peut répondre aux différentes requêtes des acteurs et qu'elle peut les informer des différents événements qui se passent.

3.2.5 Les tâches

Rappelons qu'il y a quatre types de tâches et qu'elles peuvent fonctionner en parallèle. Les tâches sont initialement déclenchées par des situations types, d'autres tâches ou des requêtes. Elles ont des variables locales, des structures de contrôle et sont constituées par une séquence de primitives.

Nous allons d'abord parler de la stratégie qui permettra au parallélisme de fonctionner correctement. Ensuite, nous finirons par énoncer les permissions d'utilisation du langage pour les différents niveaux de tâches.

3.2.5.1 Le parallélisme

Pour que les tâches fonctionnent en parallèle (i.e. empêcher des interbloquages éventuels), un solution classique à ce problème est utilisée : la préemption. Dès qu'une tâche doit attendre après un résultat pouvant prendre un temps arbitrairement long, elle est préemptée. Elle l'est aussi à chaque ligne d'exécution pour permettre aux autres tâches de se dérouler.

Les primitives du langage sont atomiques (permettant ainsi les pas atomiques du moteur). Seules les primitives qui doivent attendre après le résultat d'une requête ou une réponse

de l'apprenant font exception car elles doivent se préempter.

Le moteur nous garantit de réveiller la tâche de temps à autre pour continuer son exécution. Cependant, elle peut avoir à attendre qu'une tâche ou une requête active se termine.

La tâche exécute un petit en-tête à chaque fois qu'elle sera réveillée par le moteur. Si elle attend après le résultat requête ou une tâche active, elle regarde dans sa boîte aux lettres et ne continue que s'il y a eu une réponse.

De même, lorsqu'une tâche se termine, elle envoie son résultat à ce qui l'a déclenché. Lorsqu'une tâche se termine (échec ou succès), l'acteur met dans sa boîte aux lettres le résultat de la tâche, en annonçant que c'est lui-même qui l'a posté. Ainsi, déclencher une tâche est l'équivalent de se faire une requête. Si cette tâche est la réponse à une requête, l'acteur va ensuite retourner le résultat à l'acteur qui a demandé le service.

3.2.5.2 Les tâches opératoires

Une tâche opératoire se veut être une tâche visible à tous. Elle peut avoir des paramètres et des variables locales. Nous avons droit aux structures de contrôle (if then else, while, for...) et les fonctions suivantes:

- La fonction d'extraction (.);
- **ask** <question>;
- **existbb** [<titre>] [from <nom d'acteur>][category <catégorie>];
- **existreply** <identificateur> [from <nom d'acteur>];
- **getbb** <titre>|* [from <nom d'acteur>] [category <catégorie>];
- **getevent**;
- **getglobalvar** <nom>;
- **getmem** <titre> in <nom de variable de mémoire>;
- **getreply** <titre> [from <liste de noms d'acteurs>];

- **getstudentinfo** attention | alert | anxious | interest | self-confidence | self-esteem | workmemory | generalknowledge | associativelearning | inductivereasoning | quickness | defaultknowledge | metacognition | impulsive | serial | verbal | passive | random | conformist;
- **getstudentknowledge** objective | resourcepassed | resourcescore | capacity | maxlevelcap <identificateur>
- <expression> **istype** <type>;
- **ifno**;
- **lookbb** <titre>|* [from <nom d'acteur>] [category <catégorie>];
- **lookmem** <titre> in <nom de variable de mémoire>;
- **make** <structure>;
- **matchevent** [<var> =] <type> , [<var> =] <type> ... [<var> =] <type>;
- **putbb** <information> as <titre> [category <catégorie>];
- **putmem** <entier> as <titre> in <nom de variable de mémoire>;
- **req** <titre> [with <paramètres>][to <liste de noms d'acteurs>] [passive][ifno <que faire si refus>];
- **say** <message>;
- **send** <structure>;
- **seteventfilter** {<filtre1>,<filtre2> ... <filtre>};
- **setexpression** <expression>;
- **setglobalvar** <nom> to <expression>;
- **updatecapacity** <idcapacite> to <niveau>;
- **updateobjective** [<idobjective> to] <niveau>;
- **updateressource** [<idressource> to] <niveau> [context <contexte>].

3.2.5.3 Les tâches abstraites

Les tâches abtraites sont des tâches opératoires de plus haut niveau. Ces tâches ne devraient être visibles qu'à l'acteur qui les exécute. Elles annoncent ses intentions (i.e. mentir). Ces tâches peuvent appeler d'autres tâches avec la primitive suivante.

calltask <alias de tâche> [with {<param1>, <param2> ... <param>}][passive][ifno <que faire si échec>]
fait appel à une sous-tâche. L'option passive permet à la fonction d'être passive au lieu d'être active. Si l'option est utilisée, alors la primitive retournera un identificateur au lieu du résultat de l'appel. Il faut alors utiliser **existreply** et **getreply** pour récupérer le résultat. **<que faire si échec>** peut être deux choses:

- ignore: continue d'exécuter la tâche (comportement par défaut);
- abort: arrête la tâche et génère un échec à son tour.;

On peut aussi utiliser la primitive **ifno** plus loin dans la tâche à la place de l'option du même nom. Cette primitive fonctionne de la même façon que pour une requête. Cette primitive est l'équivalent de faire une requête à soi-même à la différence près que la tâche sera ajoutée à la trace de l'acteur comme étant une sous-tâche (tâche apelée par une autre).

3.2.5.4 Les tâches de contrôle

Les tâches de contrôle sont des tâches abstraites qui prennent des décisions. Ce sont elles qui vont choisir de mentir par exemple. Elles peuvent utiliser

- **lookact** [<liste de noms d'acteurs>] [for <filtre de tâches>] [jumptolast | jumpto <temps>][notended];
- **gettracetaskparam**;
- **gettracetime**;
- **getevent**;

- **matchevent** [<var> =] <type> , [<var> =] <type> ... [<var> =] <type>;
- **aliastask** <identificateur de tâche>|<alias de tâche> to <alias de tâche>

où le premier paramètre désigne le nom d'une tâche lors de son implantation, ou un alias déjà défini. Le deuxième désigne un autre nom. Lors de **calltask**, l'acteur cherche d'abord parmi ses alias pour trouver la tâche. Si l'alias n'est pas trouvé, la recherche se fera parmi les véritables noms de tâches;

unaliastask <alias de tâche>
détruit l'alias (fonction inverse d'aliastask);

3.2.5.5 Les tâches de cognition

Les tâches de cognition sont des tâches de contrôle qui s'occupent de réaliser la couche cognitive de l'acteur. Elles peuvent utiliser:

- **createtask** <identificateur de tâche> <code de la tâche> oftype operative | abstract | control | cognitive;
  Ajoute une nouvelle tâche lors de l'exécution. Son nom, son code et son type sont spécifiés en paramètre.
- **createTS** <nom de situation type> <code d'évaluateur> <alias de tâche>;
  Ajoute une nouvelle situation type lors de l'exécution. Son nom, son code et l'alias de tâche qui lui est associé type sont spécifiés en paramètre.

- **setglobalfocus** <expression de focus>
  Modifie le focus général.

Ces différentes primitives n'existent pas dans l'implémentation présente du moteur parce qu'elles n'ont pas été jugées nécessaires pour les différents prototypes qui ont été implantés. De plus, la couche cognitive n'a pas encore été spécifiée assez clairement pour des prototypes ayant besoin de ces fonctionalités. À cause de cela, les primitives de cognition (ainsi que **aliastask** et **unaliastask**) ne seront implantés que lorsque les

décisions nécessaires seront prises.

3.3 Le moteur ADL

Le moteur ADL a pour but d'interpréter le code décrivant une stratégie pédagogique (société d'acteurs) et d'intervenir auprès des autres modules de SAFARI. Les acteurs pourront être ainsi en mesure d'assister l'apprenant dans le déroulement des ressources et de prendre les décisions nécessaires pour que l'apprentissage puisse être fait suivant le but de la stratégie pédagogique choisie par le gestionnaire de session.

3.3.1 Communications avec les autres modules de SAFARI

Le moteur ADL n'a pas d'interface graphique en tant que tel. Son travail est essentiellement de compiler du code, de l'exécuter et de communiquer avec les autres modules de SAFARI: les ressources, l'interface, le modèle de l'apprenant et le gestionnaire de session. Ces modules ont déjà été discutés plus tôt dans ce mémoire (voir chapitre 1,2.2.1 et 3.2.4).

Les moyens de communication entre le moteur et les autres modules varient d'un prototype à l'autre. Un prototype utilise une communication pour les ressources utilisant des sockets (voir A.9, p. A-15 pour la grammaire du protocole) et CORBA (contrainte de SAFARI) pour dialoguer avec les autres modules. Dans le dernier prototype de SAFARI cependant, le moteur dialogue avec les autres modules directement puisqu'ils ont été compilés dans un même exécutable.

3.3.2 Exécution du code ADL dans ses grandes lignes

On divise toute description de stratégie pédagogique écrite en ADL en quatre types de composantes (Figure 9) :

- la stratégie: détermine les différents acteurs qui composent la société et peut contenir du code permettant de résoudre des conflits de situations types;
- l'acteur: description d'un agent de module pédagogique. Cette description contient

entre autres les situations types;

- la tâche: description d'une action qu'un acteur peut accomplir. Elle contient l'expertise des acteurs. Les tâches sont partagées entres les acteurs. Elles peuvent être exécutées en parallèle en autant d'instances voulues;
- la structure: définit un message avec ses différents champs. Ce message fait partie d'un protocole de communication entre les acteurs et les ressources.

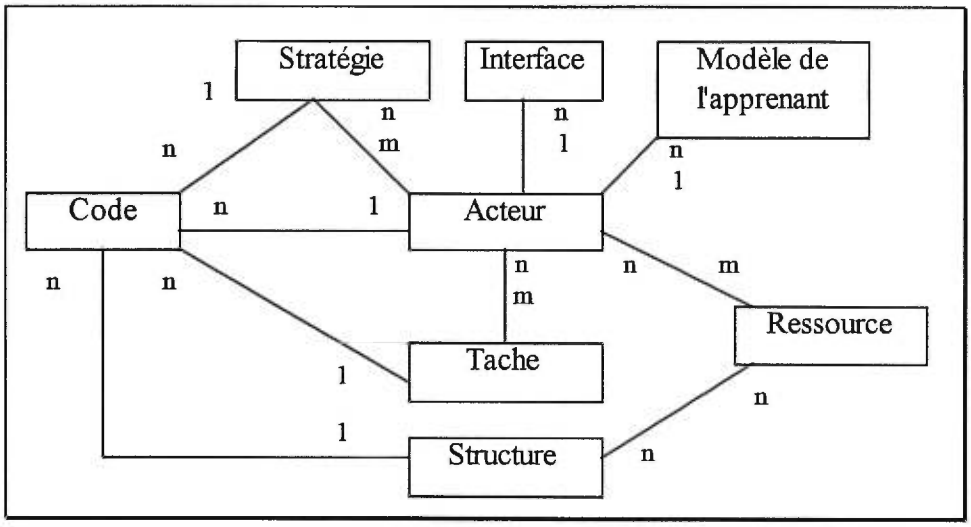

Figure 9. Organisation des éléments du code ADL des acteurs

Plusieurs stratégies pédagogiques peuvent être décrites dans un même source (ou code) ADL et partager des acteurs ainsi que des tâches. L'exécution d'une stratégie active les différents acteurs qui en font partie. Ils peuvent interagir avec les différentes ressources, le modèle de l'apprenant et sont représentés dans l'interface graphique (modules avec lequel le moteur interagit). La forme de la représentation dépend de l'interface. Mais généralement, elle devrait offrir la possibilité de montrer l'état émotionel de chaque acteur (par des expressions faciales par exemple) et donner à l'apprenant leurs messages. Un visage parlant peut ainsi représenter chaque acteur.

Les acteurs interagissent avec les ressources en réagissant aux différents événements de la ressource (i.e. actions de l'apprenant) et en leur donnant des requêtes (i.e. donner un indice sur une connaissance). Cette interaction se fait par un protocole dont les messages

sont définis par les structures (voir l'énoncé des composantes du code ADL ci-dessus).

Le moteur ADL n'est pas uniquement un interprète. Il passe d'abord par une phase de précompilation du code ADL. Chaque composante du code ADL (voir ci-dessus) est modélisée en un objet C++ dialoguant directement avec le moteur (qui est un autre objet C++). La performance des acteurs est améliorée puisque l'interprétation du code est remplacée par un dialogue entre des objets C++.

3.3.3 L'exécution en parallèle

Le moteur ADL étant écrit en C++ portable (contrainte de SAFARI), il n'y a pas de possibilité d'exécuter du code en parallèle. Or, une société d'acteurs peut exécuter des actions en parallèle (i.e. des tâches). Pour résoudre le problème, le moteur exécute une boucle de pas atomiques, c'est à dire par une série d'actions courtes et qui ne peuvent pas être exécutés en même temps qu'un autre pas. Ainsi, le moteur peut partager le temps d'exécution entres les différents acteurs sans qu'ils puissent se gêner. Cette boucle de pas atomiques permet alors le fonctionnement en parallèle des différentes tâches, des situations types, le dialogue avec les ressources et celui entre les différents acteurs. De plus le moteur peut interrompre et démarrer les acteurs en tout temps puisqu'il contrôle la boucle. Cette dernière fonctionnalité s'avère utile pour le gestionnaire de session.

Le fonctionnement du moteur se résume alors à réveiller les différents acteurs pour qu'ils puissent exécuter leur prochain pas. Le moteur demande à tous les acteurs de vérifier leur situation type. Deux cas peuvent se présenter:

- un seul acteur peut s'activer. Il s'active;
- plusieurs peuvent s'activer. Il y a un conflit à résourdre.

Les différents acteurs sont dans une liste d'évaluation. La politique suivante résoud un conflit : l'acteur pouvant pouvant s'activer se trouvant le plus haut dans la liste est sélectionné. Il déclenche sa situation type et est mis à la fin. Ainsi, celui qui a déclenché sa situation type sera le dernier à vérifier s'il peut en déclencher une la prochaine fois. Par

défaut, la stratégie de résolution de conflit est de prendre le premier acteur dans la liste qui peut s'activer. Cette stratégie peut se remplacer par du code ADL (voir A.5, p. A-12). La stratégie pédagogique exécute alors ce code. Il obtient la liste des acteurs en conflit et détermine lequel peut s'activer.

Après avoir fait examiner les situations types et qu'un acteur ait déclenché sa tâche, le moteur va demander à chaque tâche qu'il a dans sa liste de se réveiller pour faire un pas. Elle se préempte (ou s'interrompt) ensuite immédiatement après. Cette philosophie de réactiver chaque tâche peut poser problème : réactiver plus souvent les tâches récentes peut améliorer le temps d'attente moyen des tâches. Cependant, il n'y a pas de contrôle sur le temps alloué à chaque tâche. Donc, une telle politique ne peut être envisagée.

Les tâches déclenchées peuvent être *actives* ou *passives*. Une tâche active ne permet pas l'évaluation des situations types pendant son exécution et une tâche passive le permet. Une situation type déclenche toujours une tâche active. Une tâche active permet l'exécution d'actions qui peuvent être exécutées en parallèle (i.e. de longs calculs). Dès qu'une tâche active se déclenche, le moteur entre dans un mode "actif" où il change son comportement: il ne demande plus aux acteurs de vérifier leurs situations types. Seules les tâches en attente peuvent se faire réactiver. Ce mode actif se termine lorsque toutes les tâches actives sont terminées.

Le code des acteurs doit pouvoir être changé en tout temps par un éditeur (voir 2.2.4 p.20). Le fait qu'ADL interprété permet de modifier facilement le code des acteurs par un éditeur ou directement par un programmeur dans le code ADL des acteurs. De plus, si jamais une version future du langage permet la modification du code pendant son exécution (ceci peut être demandé lors de certains développements éventuels de la couche cognitive), le fait que le moteur soit un interprète ne peut que simplifier les choses. En effet, il peut interrompre l'exécution des acteurs en tout temps et possède leur état ainsi que celui des tâches.

Maintenant que le langage ADL est défini, le prochain chapitre propose des exemples pour des fins d'illustration.

Chapitre 4

Exemples d'implantation d'acteurs

Ce chapitre décrit quelques prototypes d'acteurs pour soulever certains points discutés dans ce mémoire. Nous illustrons ici comment décrire des acteurs avec ADL. Nous discutons entres autres comment exprimer les différentes composantes des acteurs (les situations types, les tâches, l'interaction avec l'usager et le parallélisme entres autres), l'importance de permettre aux ressources d'analyser les actions de l'usager pour permettre la séparation des expertises: pédagogique, assurée par les acteurs, et du domaine, assurée par les ressources. Nous finissons par un exemple portant sur la couche cognitive: un exemple d'apprentissage chez les acteurs. Nous montrons comment nous pensons établir la couche cognitive, et pourquoi nous jugeons nécessaire d'établir une sémantique dans le processus d'apprentissage. Cet exemple ouvrira la discussion sur l'apprentissage machine. D'autres exemples peu discutés se trouvent à la fin de ce mémoire (Annexe B, Annexe C et Annexe D).

4.1 Un premier exemple de code

Voici un exemple simple de code illustrant le langage des acteurs. Dans cet exemple [Mengelle *et al.*, 98], nous avons trois acteurs. Le premier est le Tuteur. Son travail est de donner le cours. L'Étudiant est un apprenant simulé ou un acteur représentant l'apprenant réel (qui s'occupera alors de traduire en tâches les différentes actions de l'apprenant). Et finalement, le Compagnon, qui agira comme un ami et qui cherche à donner des indices à l'Étudiant. Nous ne présentons qu'un extrait du code ADL pour cet exemple. Il ne cherche pas à expliquer une stratégie, mais à illustrer certains mécanismes de base des acteurs à travers le langage ADL. L'extrait comporte une description simple d'un acteur, et la tâche que sa situation type peut déclencher.

4.1.1 Un premier acteur en ADL

La description de l'acteur Compagnon indique une seule situation type (Figure 10) qui peut éventuellement déclencher une tâche appelée `BesoinAide`.

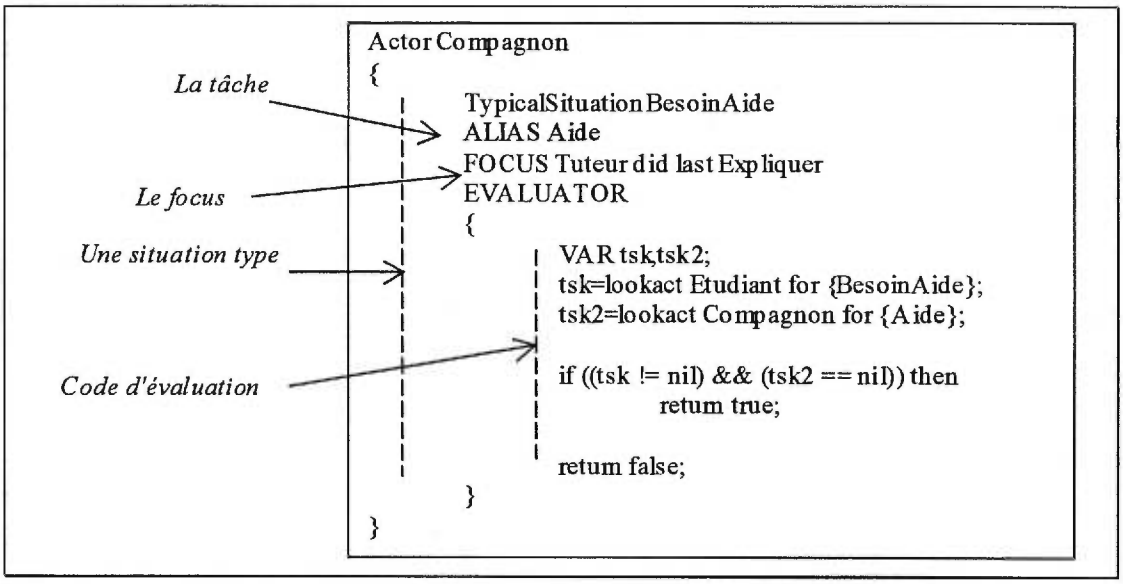

Figure 10. Un exemple d'acteur en ADL

Le focus indique que cette situation type n'est intéressée qu'aux actions accomplies depuis la dernière fois que le Tuteur a accompli la tâche `Expliquer`. Le code d'évaluation va déterminer si la tâche doit être déclenchée ou non. La primitive `lookact` pour récupère les différentes tâches accomplies dans le passé par les acteurs. Il faut se rappeler que cette primitive est sujette au focus de la situation type. Elle permet de récupérer une trace éventuelle de la dernière fois où l'Étudiant à demandé de l'aide (en exécutant la tâche `BesoinAide`), ainsi que la dernière fois où le Compagnon a cherché à lui venir en aide (en exécutant la tâche `Aide`).

La situation type permet de déclencher la tâche `Aide` si l'Étudiant demande de l'aide et que le Compagnon ne lui a pas encore répondu, tout cela, depuis la dernière fois que le Tuteur a expliqué une partie de la matière. Ceci n'est qu'une utilisation simple de la primitive `lookact`.

4.1.2 Une première tâche en ADL

Voici la tâche que le Compagnon peut déclencher avec sa situation type (Figure 11).

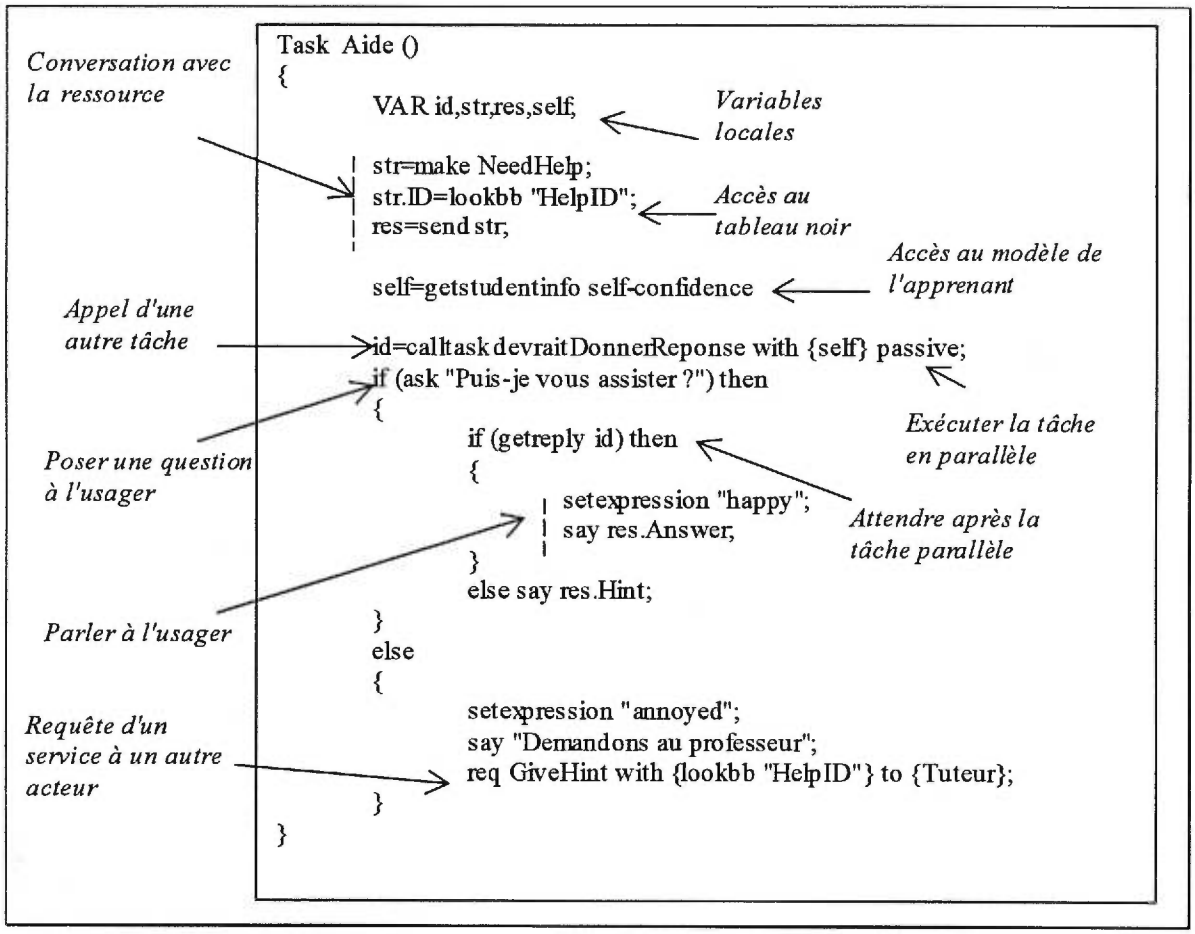

Figure 11. Un exemple de tâche en ADL

Cette tâche demande d'abord à la ressource de suggérer un indice possible au sujet d'un certain détail de la matière désigné par `HelpID` dans le tableau noir. Cet échange n'est pas le protocole de communication avec les ressources discuté en 3.2.4.4, p.46, mais un protocole qui se veut simplifié pour les fins de l'illustration. Dans le protocole de notre exemple, la ressource peut répondre à une requête `NeedHelp`. Pour demander cette requête, l'acteur génère une structure de type `NeedHelp` avec la primitive `make`. Ensuite, il va en remplir les champs des informations nécessaires. Dans notre cas, un seul champ est

demandé. Ce champ contient un identificateur concernant une partie du cours désigné par `HelpID`. Cet identificateur désigne une information dans le tableau noir. Elle y a été déposé par la dernière exécution de la tâche `BesoinAide` de l'Étudiant, d'où l'utilisation de la primitive lookbb pour récupérer l'information. L'acteur envoie la requête par la primitive `send`. Comme le protocole prévoit un retour, la primitive `send` attend après la réponse de la ressource et la retourne. Le résultat est une autre structure. Dans notre cas, la structure contient deux champs. L'un contient un texte indiquant un indice, et l'autre contient la réponse.

Pour exécuter cette tâche, le Compagnon va faire appel à une autre tâche appelée `devraitDonnerReponse` (avec la primitive `calltask`). Cette tâche contient une expertise pouvant déterminer si le Compagnon doit donner une réponse ou un indice à l'Étudiant. Cette tâche prend un paramètre qui se trouve à être le degré de confiance de l'apprenant en lui-même (obtenu par la primitive `getstudentinfo`). Supposons que cette tâche soit complexe et qu'elle prenne quelques secondes. Comme nous devons poser une question à l'apprenant, l'acteur déclenche cette tâche de façon parallèle (tâche passive). La primitive `calltask` retourne alors un identificateur. Si l'Étudiant accepte de recevoir de l'aide du Compagnon, le résultat de la tâche est obtenue avec la primitive `getreply`. Dans des cas plus complexes, il est possible de vérifier si la tâche est terminée sans attendre après son résultat avec la primitive `existreply`.

La question est posée à l'apprenant par la primitive `ask`. Cette primitive permet d'obtenir une réponse de l'étudiant. Si l'étudiant accepte l'aide du Compagnon, ce dernier donne un indice ou la réponse, dépendamment du résultat de l'expertise de la tâche `devraitDonnerReponse`. Sinon, il demande de l'aide. Pour ce faire, il exécute une demande de requête appelée `GiveHint` envers le Tuteur. Ce dernier peut accepter ou refuser de répondre. S'il répond, il exécute une tâche et en retourne le résultat. Ce résultat est retourné par la primitive `req` qui a parti la tâche. Le Compagnon peut réagir avec des émotions envers l'apprenant en fonction de la demande d'aide en utilisant la primitive `setexpression`. Il est heureux si l'étudiant lui demande de l'aide et fâché sinon.

4.2 Stratégie du perturbateur et diagnostic de actions de l'apprenant dans l'analyse d'une mammographie

Voici un exemple utilisant la stratégie du perturbateur. Cet exemple fut un des tous premiers prototypes des acteurs. Nous trouvons trois acteurs: le tuteur et le perturbateur. L'apprenant et le perturbateur vont travailler ensemble sous la supervision du tuteur. Le travail porte sur le diagnostic à partir de l'analyse d'une mammographie [Mengelle *et al.*, 98]. L'apprenant et le perturbateur ont pour but d'encercler certaines zones ayant un intérêt pour le diagnostic et de les nommer (Figure 12). Le code d'une des premières versions de ce prototype se trouve en Annexe (Annexe B, p. B-16). Il ne correspond pas cependant au code plus avancé présenté dans cet exemple. Voici ici un exemple où l'expertise pédagogique est séparé de l'expertise du domaine entre les acteurs et la ressource.

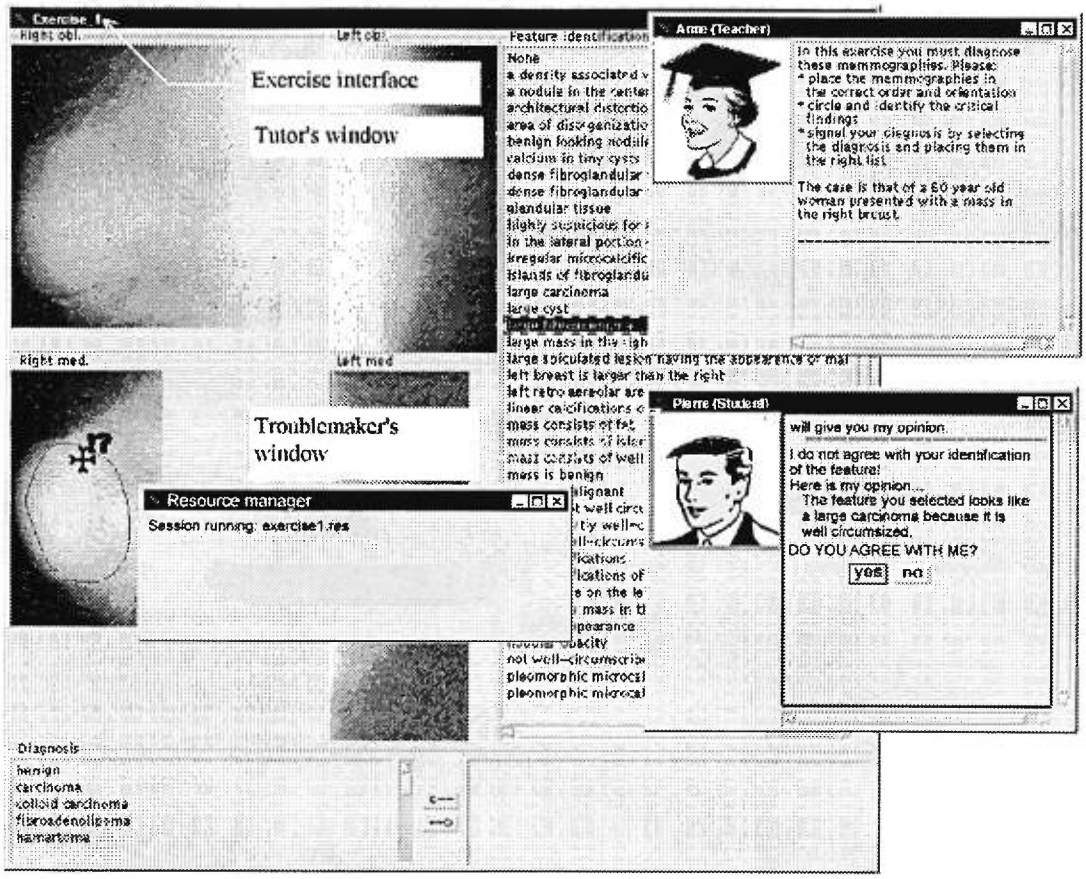

Figure 12. Interface de la mammographie [Mengelle et al., 98]

Rappelons que les acteurs décident les interventions pédagogiques auprès de l'apprenant alors que la ressource (qui inclut l'interface graphique dans cet exemple) prend les décisions ayant rapport au domaine de la mammographie. C'est la ressource qui va interpréter les différentes actions de l'apprenant sur l'interface dans les termes du domaine et qui va déterminer les bonnes et les mauvaises réponses de l'apprenant. Les acteurs sont informés des actions de l'apprenant au fur et à mesure que la session se déroule. Ainsi, le tuteur peut féliciter l'apprenant après la résolution d'une étape du problème, lui donner un indice s'il a des problèmes, etc. Pour ce faire, la ressource envoie un diagnostic aux acteurs à chaque action de l'apprenant. Ce diagnostic décrit l'action accomplie avec les connaissances qui lui sont liées. Les acteurs se basent sur les différents diagnostics pour prendre leurs décisions. Dans l'exemple de la Figure 12, l'apprenant a encerclé une zone sur une mammographie. La ressource envoie un

diagnostic aux acteurs (elle annonce une bonne réponse). Une situation type du perturbateur qui s'appelle `ReactionToLearnerAction` se déclenche à l'arrivée du diagnostic et une tâche de contôle appelée `TroublemakerChooseReaction` est activée (Figure 13). Dans cet exemple, le modèle de l'apprenant se trouve à être géré par un troisième acteur appelé LearnerModelManager. Il peut donner des informations à travers des requêtes. Cet acteur n'apporte rien au niveau pédagogique dans cette stratégie. Il ne sert qu'à faire des requêtes au modèle de l'apprenant et nous pouvons l'ignorer dans la suite de l'exemple.

```
TypicalSituation ReactionToLearnerAction
ALIAS TroublemakerChooseReaction
FOCUS { Troublemaker at control tasks}
// Cette situation typen'a besoin que du comportement du perturbateur
EVALUATOR
// On vérifie l'existence d'un diagnostique possible sur l'apprenantS'il y en
//a un et que le perturbateur n'a pas encore choisi une réaction on part la tâche
 {
  VAR diagnosis, notFirstReaction;
  diagnosis =lookbb "RECENT_DIAGNOSIS";
  if (diagnosis == nil ) then  return false;
  firstReaction = lookact for { "TroublemakerChooseReaction" };
  if ( notFirstReaction  == nil ) then  return true;
 else  return false;
}


ControlTask TroublemakerChooseReaction ( )
// Cette tâche permet au perturbateur de choisir sa réaction
{
  VAR level,which;
// Récupérer le diagnostique dans le tableau noir
  which = lookbb "RECENT_DIAGNOSIS";

  level = (req "ReturnStudentLevel" with { which } to
{"LearnerModelManager"});
// Si l'acteur LearnerModelManager annonce que la connaissance de
// l'étudiant est suffisante, on cherche à le confondre. Sinon,
on
// lui donne un bon indice
  if (level >= 3) then calltask "TroublemakerMislead" with {
which };
  else calltask "TroublemakerHelp" with { which };
}
```

Figure 13. Code de la mammographie, extrait et adapté de [Mengelle et al., 98]

Dans ce cas-ci, le perturbateur considère que l'apprenant a une bonne connaissance sur le sujet du diagnostic et appelle la tâche opératoire `TroublemakerMislead` dans le but de confondre l'apprenant. Cette dernière tâche appartient à la couche de contrôle de l'acteur (remarquons que le type de la tâche est `ControlTask`). En effet, cette tâche se prête bien à

la couche de contrôle. Elle va décider du comportement de l'acteur en fonctions des différentes informations disponibles.

On voit en Figure 12 que Pierre montre son désaccord avec l'apprenant et va même se justifier (voir la fenêtre représentant Pierre). Ici, le texte affiché par Pierre est déterminé par la ressource. Le perturbateur choisit de confondre l'apprenant sur une certaine connaissance, et la ressource choisit alors le texte à afficher. Ainsi, nous séparons bien l'expertise du domaine de l'expertise pédagogique. Le code des acteurs peut se prêter à une situation très différente d'une mammographie.

4.3 L'apprentissage à deux

Voici un exemple illustrant la possibilité d'apprentissage pour les acteurs. Ce n'est pas une proposition de couche cognitive, mais un exemple montrant qu'il est possible de la réaliser en utilisant le langage ADL. En effet, cet exemple montre comment un acteur peut aider à faire un diagnostic sur son comportement. Dans cet exemple, un expert humain (appelé programmeur) fait les modifications nécessaires pour corriger les problèmes détectés par les acteurs. Nous expliquons dans la discussion que rendre le processus de correction automatique (sans intervention humaine) est encore un problème ouvert. Ceci explique pourquoi un expert humain (le programmeur) fait les corrections.

[Mengelle *et al.*, 98] explique que le processus d'apprentissage des acteurs se fait comme suit: le programmeur exécute les acteurs sur une série de problèmes. Un algorithme d'apprentissage tente de diagnostiquer les problèmes en cherchant des critères permettant de séparer les résolutions correctes de problèmes des résolutions incorrectes. Le programmeur interprète ensuite ce diagnostic et améliore l'expertise des acteurs (en ajoutant ou en enlevant des règles par exemple), pour relancer le processus jusqu'à ce que les performances des acteurs soient jugées suffisantes.

Un exemple simple y est proposé. L'exemple se veut le plus simple possible pour éviter

des cas compliqués qui ne feraient que nuire au but de l'illustration. Pour ce faire, l'exemple ne porte pas sur les STI.

Deux acteurs cherchent à fabriquer des mots avec un alphabet de quatre lettres: un carré rouge, un cercle rouge, un carré bleu et un cercle bleu. Les mots sont fabriqués une lettre à la fois, allant de gauche à droite. À chaque tour, les acteurs déterminent la prochaine lettre du mot. Un acteur, appelé le traceur, choisit la forme de la lettre, alors que l'autre, le peintre, en choisit la couleur. Ces acteurs n'ont pas le droit de communiquer leurs intentions. Ils ne peuvent baser leur décision que sur l'état courant du mot, c'est-à-dire les choix posés avant ce tour.

Les mots doivent suivre une grammaire et les deux acteurs n'en commencent qu'avec la connaissance d'une moitié chacun. Ainsi les acteurs sont capables de déterminer en équipe si un mot n'est pas bon, mais ils doivent apprendre à coopérer pour fabriquer des mots de 20 lettres. Le code ADL d'une stratégie correspondant à cette description en annexe (Annexe C, p. C-26).

Les règles du jeu (grammaire):
1) Une lettre rouge doit suivre une lettre carrée.
2) Une lettre bleue doit suivre deux lettres rouges consécutives.
3) Une lettre ronde doit suivre une lettre bleue.
4) Une lettre carrée doit suivre deux lettres rondes consécutives.

Le traceur connaît les règles 1 et 2, tandis que le peintre connaît les règles 3 et 4. Il est possible que les acteurs arrivent à un cul de sac. En effet, si les deux dernières lettres choisies sont rouges et que la dernière lettre est carrée, le peintre se trouve bloqué parce que ses deux règles se contredisent. De même, le traceur se trouvera bloqué si les deux dernières lettres sont rondes et que la dernière est bleue. Si les acteurs se rendent à 20 lettres, ils réussissent. Sinon, c'est un échec. Les acteurs peuvent aussi ne pas être

contraints à un seul choix. Ils choisissent alors au hasard une possibilité.

4.3.1 Le processus général d'apprentissage

Le processus présenté en Figure 14 est un processus général d'apprentissage appliqué à notre problème. Il fonctionne comme suit: nous récupérons d'abord la trace pour chaque séance où les acteurs cherchent à fabriquer un mot en suivant leurs expertises respectives (i.e. un ensemble de règles). Cette trace est ensuite filtrée des tâches qui ne sont pas pertinentes pour l'apprentissage et qui pourraient apporter un bruit inutile au processus. Ainsi, nous pouvons nous intéresser qu'à certaines tâches, des acteurs particuliers, ou à un certain intervalle de temps. Un tel travail est accompli à travers les focus et les options accessibles à la primitive `lookact`. Dans notre cas, nous ne sommes intéressés qu'aux choix posés par les acteurs pour déterminer leurs lettres.

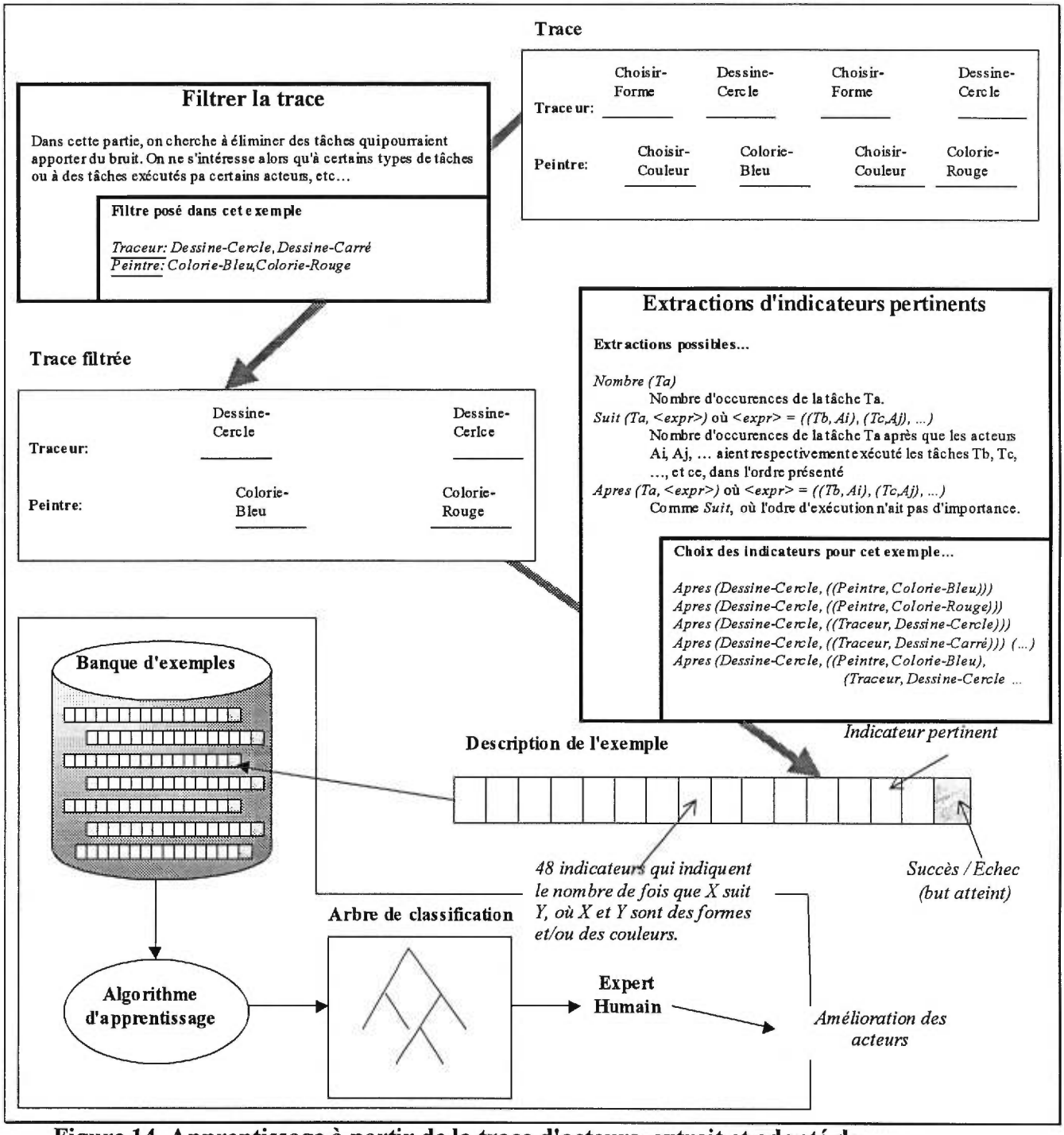

Figure 14. Apprentissage à partir de la trace d'acteurs, extrait et adapté de [Mengelle *et al.*, 98]

Ensuite, il faut extraire des indicateurs pertinents pour obtenir la description d'un exemple (ou séance). Une analyse de la trace est faite pour en récupérer des indices qui

vont nous aider dans le processus d'apprentissage. Les indicateurs sont définis par des extracteurs (i.e. Nombre, Suit et Apres). Rien ne nous empêche de définir d'autres extracteurs, tant qu'ils sont définis pour être indépendants des activités à examiner. Ces extracteurs peuvent ainsi être réutilisés dans d'autres cas d'apprentissage.

Les indicateurs sont en fait des indices pour l'algorithme d'apprentissage. Nous spécifions ici quels sont les concepts à examiner. De tels indices peuvent être le nombre de fois qu'une certaine tâche Ta est représentée dans la trace (appelé Nombre(Ta)), le nombre de fois qu'une certaine tâche Ta suit une séquence de tâches exécutés par des acteurs bien précis, l'ordre d'exécution des tâches devant être respecté ou non. Les deux derniers cas sont exprimés par Suit (Ta, <expr>) et Apres(Ta , <expr>), où <expr> s'exprime par ((Ta , Ai), (Tb,Aj), ...) où les différents acteurs sont désignés par Ai et Aj. Dans notre cas, nous choisissons 49 indicateurs que nous jugeons pertinents. Ces indicateurs sont:

- le nombre de fois qu'une lettre est choisie, sachant la lettre précédente (extracteur Suit). Il y a 16 cas;
- le nombre de fois qu'une forme ou qu'une couleur a été choisie, sachant la couleur et/ou la forme de la lettre précédente (extracteur Suit). Il y a 32 cas;
- un booléen montrant si la scéance est un succès ou un échec. Il suffit de vérifier que le traceur a tracé 20 lettres (extracteur Nombre).

Chaque tentative de création de mot est ainsi analysée et stockée dans une banque de données. Une fois qu'un nombre d'exemples suffisants est accumulé (dans ce cas, nous accumulons 2000 séances), nous exécutons un algorithme d'apprentissage. L'algorithme choisi ici est un arbre de classification. Cet algorithme a été conçu par l'auteur de ce mémoire et un de ses collègues dans le cadre d'un cours sur le traitement de connaissances (dont l'apprentissage machine). L'algorithme ne sera que brièvement discuté pour éviter de déborder du sujet de ce mémoire.

Avec l'arbre de classification, l'expert humain peut déduire de nouvelles règles pour améliorer la performance des acteurs, les ajouter et relancer une autre série d'exemples pour une autre étape d'apprentissage.

4.3.2 La fabrication d'un arbre de classification

L'arbre de classification propose la division de l'ensemble des exemples d'apprentissage en sous-ensembles de plus en plus petits jusqu'à l'atteinte d'une entropie suffisante. La génération des sous-ensembles est faite à l'aide d'une fonction de division des ensembles qui cherche la division la plus rentable du point de vue de l'entropie.

L'algorithme (Figure 15) cherche à séparer, en deux parties de taille significative, un ensemble d'exemples classés dans le but de générer une règle de classification. Les deux parties sont séparées à leur tour et ce, jusqu'à ce qu'une condition d'arrêt soit atteinte.

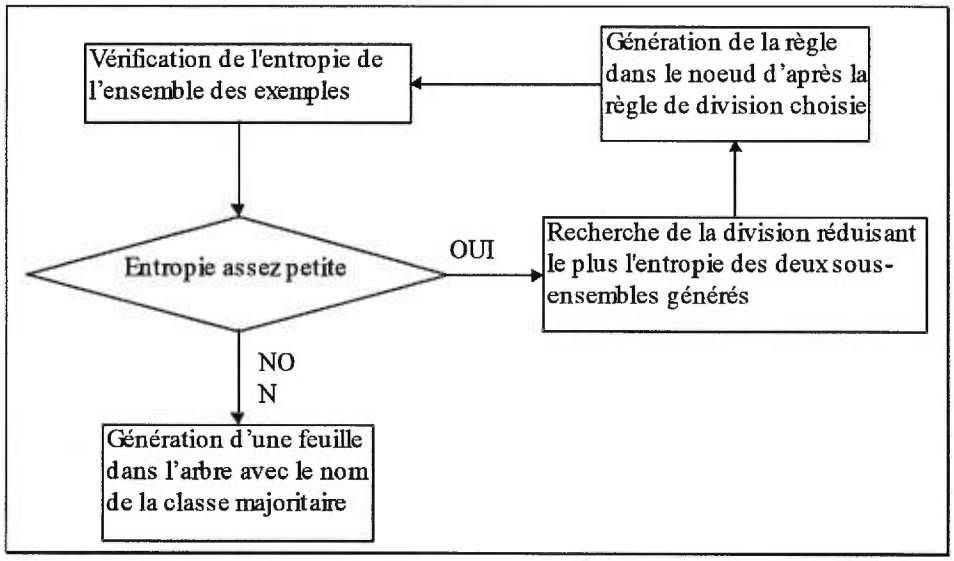

Figure 15. Algorithme de construction de l'arbre de classification

À chaque division, la séparation qui réduit le plus l'entropie est choisie. C'est la division générant les sous-ensembles dont la somme de leurs entropies respectives est minimale. Elle est choisie parmi les divisions possibles à partir de chaque attribut.

L'entropie est définie, comme le degré de non-ressemblance des exemples à l'intérieur d'un même ensemble (i.e. s'ils sont de la même classe). Cette mesure est calculée de cette façon:

$$- \Sigma \, p(j|E) \ln p(j|E)$$

où p(j|E) est la proportion de la classe j dans l'ensemble E. Si cette somme donne zéro, c'est que l'entropie est minimale et nous tombons sur un cas ou tous les exemples sont de la même classe.

Dans le cas de notre expérience, nous avons deux classes: Succès et Échec. Un identificateur contient la classe et tous les autres vont servir d'attributs.

4.3.3 Les résultats

Avec les règles de base des acteurs uniquement, environ la moitié des traces représentent des échecs. L'algorithme d'apprentissage nous génère un arbre (arbre original en Figure 16) dont la tête nous indique que le facteur le plus discriminant entre un échec et un succès est le fait de choisir un carré bleu à la suite d'un cercle rouge. Il faut que ce choix se fasse au moins 5 fois dans le mot pour éviter un échec certain. Le programmeur choisit de corriger un seul acteur à la fois. Il ajoute alors au Peintre l'heuristique suivante: "à la suite d'un cercle rouge, cherche à choisir la couleur bleu". Cet heuristique ne sert qu'à régler le cas où l'acteur n'est pas contraint à un seul choix après un cercle rouge.

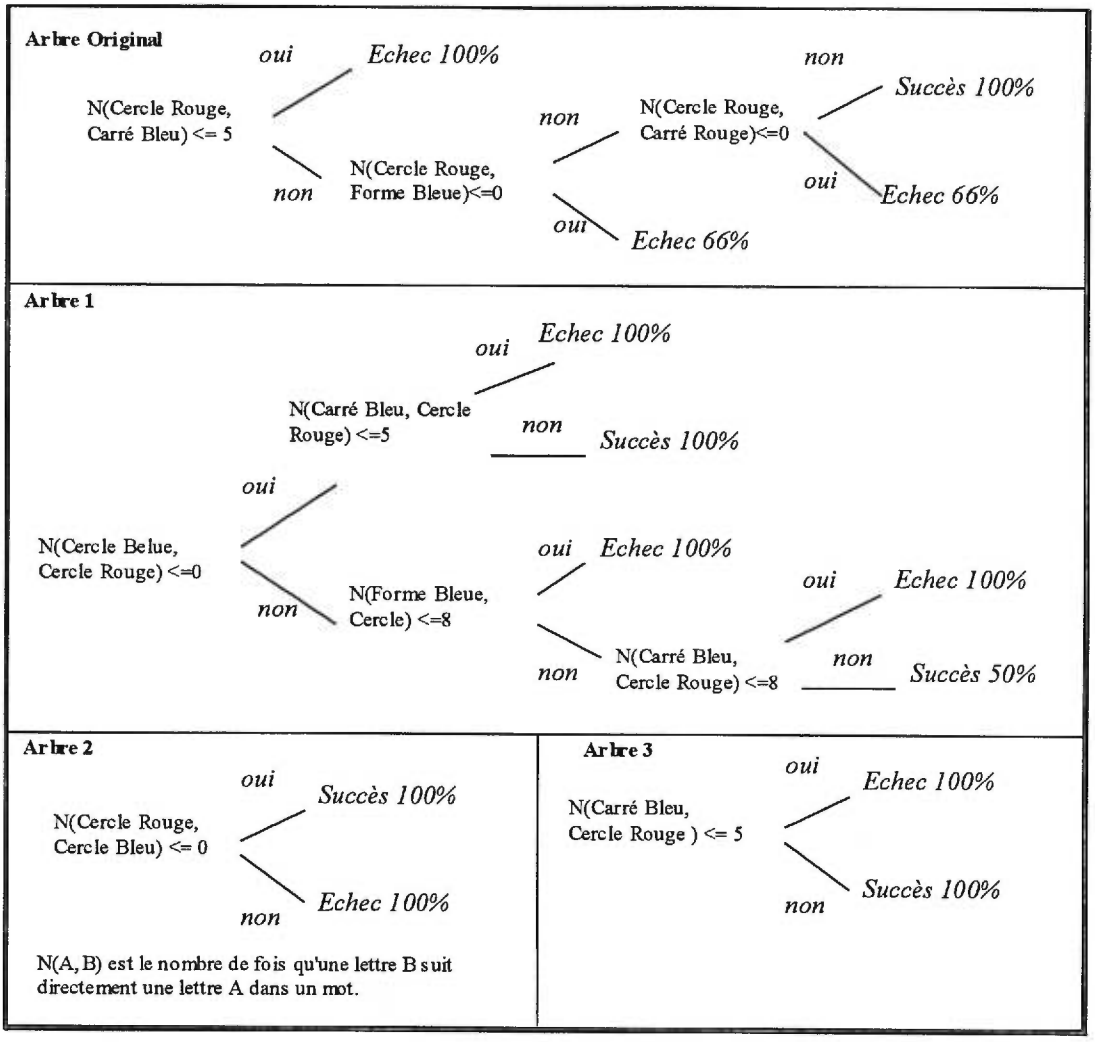

Figure 16. Arbres de classification résultants de l'apprentissage [Mengelle et al., 98]

À l'ajout de cette règle, une autre série de 2000 exemples est produite. Nous n'obtenons un succès qu'environ une fois sur 8. De l'arbre généré (arbre 1), le programmeur déduit et ajoute un nouvel heuristique au traceur: "après un cercle bleu, cherche à prendre un carré". D'après l'arbre original, il aurait fallu dire plutôt de prendre un carré après un cercle rouge, et non bleu, mais l'algorithme d'apprentissage détermine qu'une fois le premier heuristique installé, un autre heuristique (i.e. attribut dans l'arbre de classification) devient plus importante encore.

Maintenant, nous obtenons un succès 2 fois sur 3. L'arbre 2 amène à généraliser

l'heuristique précédent du traceur: "après un cercle, cherche à prendre un carré". Nous n'améliorons pas la performance des acteurs et l'algorithme d'apprentissage ne nous apprend rien (Arbre 3). En effet, les heuristiques qui peuvent en être déduites (i.e. choisir un cercle ou une lettre rouge après un carré bleu) sont déjà couvertes par les règles de base.

Cet exemple nous montre ainsi qu'une trace des tâches permet à un expert humain d'améliorer la performance des acteurs. Dans ce cas-ci, nous sommes passés d'une situation où la moitié des cas sont en échecs à une autre où 2 essais sur 3 sont des succès. Ainsi, l'utilisation d'algorithme d'apprentissage peut être très utile pour assister un programmeur dans l'amélioration d'une société d'acteurs.

Cependant, le choix des différents identificateurs (donc des extracteurs) est important pour non seulement obtenir des résultats pertinents, mais aussi pour pouvoir interpréter correctement les résultats d'apprentissage. Le choix des identificateurs définit ainsi une sémantique au processus d'apprentissage. Avec d'autres identificateurs (donc avec une sémantique différente), nous aurions peut-être pu avoir de meilleurs résultats, mais nous ne nous intéressons pas au meilleur choix possible d'identificateurs. Ceci reste un problème qui dépasse le sujet de ce mémoire. Nous croyons que l'introduction des extracteurs et des identificateurs dans ce processus est un pas important vers l'introduction de l'apprentissage cognitif. En effet, si nous donnons des outils et une sémantique qui permettent à un humain de comprendre les problèmes qu'a une société d'acteurs, nous nous approchons d'un apprentissage cognitif (apprentissage ressemblant à celui d'un humain). De plus ces outils sont indépendants du domaine d'application et peuvent alors être réutilisés dans d'autres cadres d'application.

Même si l'exemple de l'apprentissage à deux discuté apporte éléments de solution pour la couche cognitive, il est un exemple simple d'apprentissage. La couche cognitive vise à suivre l'apprentissage d'un être humain, ce qui est plus compliqué que de contruire des mots. Il est donc nécessaire d'examiner des exemples concrets d'implantation d'agents

ayant les propriétés cognitives et d'éducation. Le chapitre suivant montre qu'ils utilisent une forme d'apprentissage appelé multistratégique qui s'avère prometteuse pour l'établissement de la couche cognitive.

Chapitre 5

Apprentissage machine

L'apprentissage machine a récemment pris un virage important: la combinaison de différentes méthodes d'apprentissage pour s'attaquer à des domaines variés et complexes [Michalski, 94], [Sharp, 95]. Ces systèmes appelés multistratégiques sont très prometteurs dans le milieu de la recherche. Ce chapitre a pour but de décrire l'apprentissage multistratégique et de montrer son intérêt pour l'établissement de la couche cognitive des acteurs. Deux systèmes vont attirer notre attention: Meta-AQUA [Ram *et* Cox, 94][Cox 96a][Cox 96b] et DISCIPLE [Tecuci *et* Hieb, 96]. Meta-AQUA traite de l'apprentissage cognitif et du problème de l'assignation du blâme (i.e. déterminer quelle partie de la connaissance est fautive lors d'un mauvais fonctionnement du système), sujets importants pour la propriété cognitive des acteurs. DISCIPLE traite de l'explicitation de connaissances interactive avec l'expert pour définir un agent, donc sur leur propriété d'éducation. Nous verrons aussi que les agents de DISCIPLE ont aussi la propriété cognitive.

5.1 Apprentissage multitratégique

Dans les premières expériences en apprentissage machine, les chercheurs se sont intéressés à des tâches particulières et ont monté des systèmes limités pour des domaines particuliers. Leurs systèmes apprenaient à mieux exécuter ces tâches. Cependant, en cherchant à résoudre des tâches plus difficiles, s'attaquant à des problèmes dans des domaines plus variés, ces systèmes se sont trouvés inefficaces. On a alors pensé à combiner plusieurs mécanismes d'apprentissage pour faire des systèmes plus flexibles. Ces nouveaux systèmes appelés multistratégiques sont plus adéquats pour des problèmes généraux (s'appliquant à plusieurs domaines) et sont connus pour être souples et robustes.

De nos jours, on croit que l'apprentissage multistratégique permettra d'atteindre les buts présents dans la recherche en apprentissage machine [Michalski, 94], [Sharp, 95].

Voici quelques systèmes multistratégiques prometteurs:

- GEMINI [Danylik, 94], appliqué au diagnostic d'erreurs des réseaux informatiques, de radio militaire et à la compréhension d'histoires sur des terroristes;
- GEST [Segen, 94], système pouvant reconnaître la position de mains en temps réel grâce à une caméra. Ce système sert dans un éditeur graphique en 3D et dans un simulateur de vol;
- MOBAL [Morik, 94], système d'explicitation de connaissances robuste et souple dans son interaction avec l'usager (i.e. expert). Ce système est appliqué au trafique, au mileu médical et à la supervision d'un système distribué;
- MMA [Plaza *et* Arcos, 94a][Plaza *et* Arcos, 94b], système d'apprentissage pouvant modéliser de vastes connaissances, basé sur un langage réflexif appelé NOOS;
- PRODIGY [Veloso *et* Carbonnell, 94], système d'apprentissage par cas pour la résolution de problèmes mathématiques.
- DISCIPLE [Tecuci *et* Hieb, 96], système d'explicitation de connaissances pour un agent, appliqué entres autres à la tactique militaire (projet Captain).
- Meta-AQUA [Ram *et* Cox, 94][Cox 96a][Cox 96b], système pouvant comprendre des histoires et pouvant poser des questions pertinentes.

Dans ce mémoire, nous allons discuter plus en détail de DISCIPLE et de Meta-AQUA parce qu'ils traitent les propriétés d'éducation et de cognition d'un agent.

Michalski [Michalski, 94] présente une théorie en apprentissage machine. Cette théorie, *Inferential Theory of Learning* (ITL) explique des concepts de base permettant d'analyser et d'expliquer des stratégies d'apprentissage. Cette théorie est reconnue dans le monde de la recherche.

ITL considère que l'apprentissage est une exploration *de l'espace des connaissance* (ou *knowledge space*), guidé par un certain but (*learning goal*) pour améliorer le comportement du système. L'exploration cherche à découvrir de nouvelles connaissances à partir des connaissances déjà acquises (*background knowledge*), ou à partir de nouvelles informations du monde extérieur, à travers des mécanismes d'inférence. Ces mécanismes sont utilisés à travers une stratégie d'apprentissage.

Dans ITL, la connaissance est vue sous trois aspects: son *contenu*, son *organisation* et sa *confiance*. Si la modification d'un des trois aspects d'une connaissance améliore la performance d'un système, il y a eu apprentissage. Soit un annuaire de téléphone par exemple. Le contenu est l'ensemble des noms et des numéros qui leur sont associés. L'organisation est l'ordre alphabétique des noms et nous avons une confiance globale sur les numéros indiqués. Ainsi, il y a apprentissage à l'ajout d'une entrée. Dans le cas où nous sommes intéressés à retrouver des personnes par leurs numéros de téléphone, la mise en ordre par numéro de téléphone est un apprentissage. Finalement, la découverte d'un certain nombre de numéros dans l'annuaire qui ne sont pas bons fait perdre confiance sur la pertinence de l'annuaire (i.e. il devient désuet) et implique encore un apprentissage.

La modification des connaissances se fait par des *transmutations*. Il y a deux types de transmutations: les transmutations de manipulation, qui ne font que modifier la structure des connaissances, et celles qui génèrent de la connaissance. Chaque transmutation a sa transmutation opposée.

Les transmutations de manipulation sont:

- insertion / retrait (deletion);
- copie (replication) / destruction;
- ordre (sorting) / désordre (unsorting).

Les transmutations de génération de connaissances sont

- généralisation / spécialisation;
- abstraction / concrétisation;
- similarisation / dissimilarisation;
- explication / prédiction;
- sélection / génération;
- agglomération / décomposition;
- caractérisation / discrimination;
- association / dissociation.

Les transmutations générant la connaissance ont à leur disposition des outils d'inférence. Dans ITL, l'apprentissage comporte deux aspects: *faire des inférences* et *mémoriser*. Les inférences sont de trois types: l'induction, la déduction, et l'analogie. L'analogie est considérée comme une forme intermédiaire entre l'induction et la déduction. L'induction et la déduction peuvent se faire avec un certain niveau de confiance. Une inférence est *conclusive* si sa conclusion est certaine et éventuelle (*contingent*) si elle ne l'est pas. Ces deux termes ne s'appliquent pas vraiment à l'analogie à cause de sa forme intermédiaire.

D'après Michalski, les transmutations peuvent être implémentées à partir de n'importe quelle type d'inférence (déduction, analogie ou induction), sauf pour la similarisation et la dissimilarisation qui sont trop près de l'analogie pour s'appliquer à d'autres formes d"inférence.

On ne connaît pas vraiment de façon générale pour améliorer la connaissance d'un système de connaissances. Cependant, une méthode modifiant de façon utile la connaissance du système est considérée comme une stratégie d'apprentissage.

Une stratégie est sélectionnée pour résoudre une *tâche d'apprentissage*. Une telle tâche est définie par:

- une nouvelle information venant d'un instrument sensoriel, d'une personne, ou d'une

autre étape d'apprentissage;

- une connaissance acquise (*background knowledge*);
- un but d'apprentissage (*learning goal*).

Une importance capitale est portée sur le but d'apprentissage. Il sert à élaguer la recherche. Ce but est d'autant plus important en multistratégie où nous avons à traiter plusieurs sources d'informations. De plus, le but est important pour des raisons "cognitives" dans le système Meta-AQUA, [Ram *et* Cox, 94]. En effet, le but est un état du savoir désiré et est important pour la détermination, l'explication d'un échec de compréhension et la détermination de ce qu'il faut apprendre dans le futur pour éviter de répéter la même erreur dans le futur [Cox, 96a].

[Sharp, 95] rapporte que l'implantation des buts d'apprentissage de façon indépendante du domaine est très difficile. Le problème semble être bien résolu dans Meta-AQUA, NOOS (le langage implémentant MMA), DISCIPLE et GEMINI. Le but d'apprentissage est un domaine pointu et nécessaire de recherche en multistratégie [Michalski, 94],[Veloso *et* Carbonnell, 94], [Plaza *et* Arcos, 94b].

L'implantation des buts d'apprentissage pourrait peut-être aussi apporter une plus grande sensibilité au contexte. En effet, il est discuté que des êtres humains semblent être capables de modifier leur raisonnement dans le but d'atteindre une connaissance à laquelle ils s'attendent à retrouver [Wisniewski *et* Medin, 94]. L'implantation d'un tel but peut apporter des problèmes aussi: le raisonnement pourrait être modifié dans le but d'atteindre le but d'apprentissage. La qualité du raisonnement serait alors discutable.

Un problème qui n'a pas été abordé dans cette théorie est le problème de l'assignation du blâme ou du crédit (*crédit/blame assignation problem*). En effet, dans le cas où il arrive une erreur, comment déterminer la partie de la connaissance incorrecte? Généralement, le problème est laissé ouvert des heuristiques de récupération sont proposés. Meta-

AQUA cependant, s'attarde sur ce problème et possède des mécanismes permettant de résoudre le problème par lui-même. Ce point est aussi considéré comme un point de recherche important.

5.2 Le problème de l'assignation du blâme traité par Meta-AQUA

Meta-AQUA [Ram *et* Cox, 94],[Cox, 96a],[Cox, 96b] est un système qui cherche à comprendre des histoires. Pour permettre une meilleure compréhension dans le futur, Meta-AQUA possède même une habilité particulière: l'introspection. Lors d'un raisonnement, Meta-AQUA garde trace de son raisonnement. Si jamais il y a une erreur (i.e. le système ne comprend pas l'histoire), il peut retrouver son erreur et corriger son savoir pour ne pas avoir à répéter l'erreur une autre fois. Les auteurs rapportent que certaines théories poussent à croire qu'un véritable système intelligent est un système qui peut *apprendre à apprendre*.

"AQUA" veut dire: Answer Questions Understand Answers. Ceci veut dire que le système est capable d'énoncer des questions sur les parties de l'histoire qu'il ne comprend pas, et qu'il peut comprendre et intégrer les réponses données. "Meta" tient pour les outils d'introspection. L'introspection est un ajout à la version précédente (AQUA).

Le modèle du système est décrit comme un modèle différent des autres. Ici, il y a des mécanismes permettant aux stratégies de sélection de méthodes d'apprentissage de s'adapter.

Pour comprendre une histoire, Meta-AQUA génère une trace XP (pour eXplanation Pattern) montrant la compréhension qu'il en a. Il génère aussi une trace Meta-XP expliquant le fonctionnement de son raisonnement. Ces traces sont forment un réseau sémantique. Lorsque la compréhension de l'histoire échoue le système commence d'abord à répondre au problème de l'assignation du blâme. "Quelle est donc mon

erreur?" Pour répondre à cela, il faut examiner la trace du raisonnement et déterminer la faille. Une stratégie est sélectionnée ensuite pour apprendre ce qui est nécessaire pour corriger l'erreur, tout en respectant le but de notre apprentissage. Le système est capable de choisir une stratégie pour apprendre une nouvelle connaissance (par analogie sur des cas passés), ou bien tout simplement poser des questions pertinentes à l'usager pour corriger son erreur.

Nous approchons ici deux concepts fondamentaux peu ou pas approchés par d'autres systèmes d'apprentissage:

- le problème de l'assignation du blâme. Pour ce faire, le système est conçu pour explorer sa propre trace de raisonnement;
- la représentation explicite du but d'apprentissage. Grâce à elle, nous pouvons faire une décision sur ce qui doit être appris.

Ce système est donc capable d'utiliser différentes stratégies d'apprentissage pour comprendre une histoire. Ceci peut être pratique pour comparer le résultat de différents raisonnements. Mais en plus, il est possible d'apprendre des erreurs de raisonnements en utilisant les mêmes stratégies de raisonnements sur la trace même du raisonnement. La bonne connaissance peut être corrigée pour éviter que l'erreur ne se reproduise dans un raisonnement futur.

Ainsi, il est possible d'apprendre un concept important pour mieux comprendre une histoire, et d'apprendre à modifier son plan d'apprentissage d'une histoire. En effet, comme il est difficile de programmer un système pour qu'il apprenne correctement, pourquoi ne pas lui donner la possibilité d'apprendre à mieux apprendre?

Un tel système se donne beaucoup de flexibilité sur ses domaines d'applications. En effet, d'un domaine à l'autre, les stratégies de compréhension peuvent avoir à varier. Ici, le système est capable d'adapter ses stratégies de compréhension à un nouveau domaine. Cette flexibilité rend le système beaucoup robuste.

Ceci est un bon exemple d'apprentissage cognitif. Il serait intéressant de s'inspirer de ce travail pour la couche cognitive des acteurs (propriété cognitive).

5.3 Le pédagogue dans le processus d'apprentissage, DISCIPLE

DISCIPLE [Tecuci *et* Hieb, 96] est un système d'explicitation pour un agent. Il est utilisé entres autres à des fins militaires (projet Captain). La mémoire de l'agent est constituée d'un réseau sémantique. Les règles ont deux conditions possibles (ou bornes) d'application, s'adressent à un problème particulier, et une liste de solutions possibles si ses conditions sont remplies. Nous allons donc voir le processus d'apprentissage, puis nous allons discuter de la façon dont le système peut déduire de nouvelles règles.

5.3.1 Le processus d'apprentissage

Les deux bornes sont en fait des approximations de la véritable règle hypothétique. L'une est pensée pour être un peu plus spécifique (pouvant rejeter des cas positifs), et l'autre un peu plus générale (pouvant accepter des cas négatifs).

Les deux bornes pour chaque règle sont approximées. Sauf qu'ici, lors de la phase d'explicitation, le système offre une interaction importante avec l'usager. Le système génère des cas par analogie et propose des solutions. S'il y a erreur, l'expert peut corriger le système et ce dernier peut demander une explication si nécessaire. Il doit aussi réviser les bornes de la ou des règles fautives. Si jamais la borne plus générale couvrait un contre-exemple, on le spécialise. S'il ne couvrait pas un bon exemple, on déduit qu'il lui manque un concept et on en demande l'explicitation d'un nouveau concept. Inversement, pour la borne plus spécialisée, si elle couvre un mauvais exemple, on le généralise et si elle ne couvre pas un bon exemple, on demande un nouveau concept. Une telle explicitation guidée par le système est très appréciée des experts.

Après l'explicitation faite, le système pourra résoudre la majorité des cas par déduction.

Il pourra continuer à apprendre de lui-même devant un cas non prévu grâce à un raisonnement plausible (plausible reasoning), qui se fait avec un arbre de justification (voir 5.3.2, p. 85). Si le raisonnement ne fonctionne pas, on garde le cas comme une exception. Au bout d'un certain nombre d'exceptions accumulées, le système peut être révisé par l'expert.

Un tel système peut donc apprendre à partir de peu d'exemples et déjà trouver des concepts à surveiller pour faire son raisonnement. Il peut apprendre des règles partielles et les compléter avec des exemples futurs. L'approximation de la bonne règle par deux autres permet de réduire énormément le temps d'évaluation tout en donnant une approximation assez fiable de la bonne réponse.

De plus, sa possibilité de se perfectionner lors de son utilisation avec l'usager (et non l'expert) lui donne encore plus de souplesse. DISCIPLE est un autre exemple de méthode qui permet de guider le processus d'explicitation. Ce processus qui donne la propriété d'éducation à un agent.

5.3.2 L'arbre de justification

Tecuci [Tecuci, 94] propose un modèle pour traiter l'apprentissage par induction. Son modèle cherche à intégrer différentes méthodes d'inférence (dans son modèle, Tecuci utilise la déduction, l'analogie et l'abduction par exemple) en utilisant un arbre de justification. Pour l'ajout de chaque noeud dans l'arbre, on a le choix entre les différentes méthodes d'inférence.

L'idée est de constamment modifier notre arbre de telle sorte qu'il explique tous les exemples positifs rencontrés, mais sans couvrir les contre-exemples. Pour ce faire, on le généralise et on le spécialise à partir de l'arbre de justification expliquant le dernier cas entré.

Généralement, pour créer les différents noeuds de son arbre, on utilise de préférence la

déduction. Si elle ne peut fonctionner, on utilise alors d'autres types de raisonnements. On associe aussi une fonction de coût qui indique quelle est la confiance en cet arbre. Moins on retrouve de déductions dans la fabrication d'un noeud, moins on a confiance dans notre arbre. Les trois autres types de raisonnements sont l'analogie d'abord, puis la prédiction inductive (forme de généralisation), puis l'abduction. Si jamais un des trois types de raisonnement en contredit un autre, on refuse l'arbre.

On choisit l'arbre généré avec le coût le plus faible, donc le plus plausible et on ajoute les nouvelles règles qui ont servi à la construction du nouvel arbre.

Tecuci explique que ce modèle s'applique au raisonnement humain qui suit plusieurs lignes de raisonnements pour résoudre un problème en estimant la confiance (ou la force) des différentes lignes de raisonnements [Collins *et* Michalski, 89]. On peut donc déduire que les agents construits à travers DISCIPLE ont la propriété cognitive.

L'intérêt d'un tel système est qu'il peut intégrer d'autres types d'inférences et que l'entrée peut être de multiples formes: des faits, des exemples et des résolutions de problème.

Tecuci rapporte cependant des problèmes avec les contre-exemples. Dans le cas de contre-exemple, il est possible de détruire l'arbre de telle sorte qu'il ne couvre plus un certain nombre de cas positifs. De la même façon, ce modèle ne semble pas très résistant au bruit (i.e. n'est pas capable de rejeter facilement un contre-exemple présenté comme un exemple).

On peut souligner ici le fait que Tecuci n'a pas vraiment accordé d'importance à un fait important ici. En effet, dans un modèle se servant beaucoup de l'induction, il est important de bien structurer les connaissances sur lesquelles on va faire des inférences sous peine de générer des paradoxes de Hempel [Kodratoff, 94]. En effet, A implique B est équivalant à non-B implique non-A. Ainsi, nous pourrions confirmer une induction

"tous les corbeaux sont noirs" en trouvant une contraposée "un soulier blanc". Notre entité qui n'est pas un corbeau et qui n'est pas noir, est logiquement un bon exemple pour augmenter la confiance en la couleur des corbeaux. Pour éviter ce paradoxe, il est proposé de construire un arbre topologique des différents concepts utilisés dans les règles. En classant ainsi les différents concepts, on peut vérifier la pertinence des concepts d'un exemple pour modifier la confiance en une autre règle. Dans cete arbre, les concepts noir et blanc doivent être proches (dans la catégorie couleur par exemple). Cependant, les concepts corbeau et soulier ne le sont pas (dans les catégories respectives objet animé et objet inanimé par exemple). Lorsque les concepts décrivant un exemple sont éloignés de ceux constituant la règle à modifier, il s'avère intéressant de l'ignorer pour éviter un paradoxe de Hempel.

5.4 L'intérêt de l'apprentissage multistratégique pour les acteurs

Nous avons vu comment DISCIPLE et Meta-AQUA possèdent les propriétés d'éducation et de cognition. Ces deux exemples utilisent l'apprentissage multistratégique, sont flexibles et peuvent viser plusieurs domaines. Dans SAFARI, nous voulons des acteurs qui puissent être réutilisés d'un STI à l'autre et d'une ressource à l'autre. Ils doivent alors posséder aussi des outils d'apprentissage flexibles pour la couche cognitive. Comme nous voulons obtenir les propriétés d'éducation et de cognition et que les exemples discutés dans ce chapitre possèdent ces propriétés, nous croyons que l'apprentissage multistratégique est nécessaire pour la couche cognitive des acteurs.

Lors de la rédaction de ce mémoire, la couche cognitive n'est pas encore complètement établie chez les acteurs. Cependant, nous espérons montrer ici l'intérêt de l'apprentissage multistratégique pour l'obtenir (i.e. les propriétés d'éducation et de cognition).

Chapitre 6

Discussion et Conclusion

La conception du module pédagogique dans un STI est un problème complexe. Nous avons vu dans ce mémoire que les agents sont de bons outils pour réaliser de tels modules, d'où le choix du projet SAFARI d'introduire un agent appelé acteur. Les acteurs forment le module des acteurs (sous-partie du module pédagogique) et peuvent composer des stratégies pédagogiques, offrant ainsi des façons différentes d'intervenir auprès de l'apprenant et de gérer les ressources adidactiques d'un STI.

Nous avons vu que SAFARI a plusieurs attentes envers les acteurs

- Un expert pédagogique doit pouvoir les modifier à l'aide d'un éditeur spécialisé;
- Un programmeur doit pouvoir les modifier directement;
- Un acteur doit éventuellement pouvoir se modifier lui-même dans une prochaine étape du développement de la couche cognitive;
- Il est indépendant des ressources gérées, du domaine enseigné et des STI pour des fins de réutilisation;
- Il possède plusieurs propriétés d'agents, dont les propriétés d'éducation et de cognition;
- Le module des acteurs est intégré aux autres modules de SAFARI;
- Il doit être écrit en C++ portable;
- Les acteurs doivent pouvoir exécuter plusieurs tâches en même temps.

L'architecture des acteurs présentée dans ce mémoire montre comment obtenir plusieurs propriétés des agents mais ne cerne que partiellement les propriétés d'éducation et de cognition. Les autres attentes sont répondues par le langage ADL, pour Actor Description Language. Ce langage est un langage interprété de haut niveau forçant toute

implémentation à suivre l'architecture des acteurs. De plus, comme ce langage est interpété, il facilite l'édition des acteurs.

L'indépendance des acteurs envers les ressources et les autres STI est assumé par la généralité d'ADL et par l'établissement d'un protocole de communication avec les ressources par envoi de messages. La sémantique même des messages est laissée à l'implémentation ADL des acteurs, laissant une flexibilité aux protocoles.

Les trois dernières attentes mentionées ci-dessus sont répondues par la nature même de l'interprète (moteur ADL). Il est écrit en C++ portable, gère lui-même le parallélisme des tâches et établit l'intégration aux autres modules. Cette intégration est complètement transparente pour les acteurs. Le langage offre des primitives qui permettent d'utiliser les services des différents modules et laisse le soin au moteur de communiquer avec eux.

À ce jour, les partenaires du projet SAFARI utilisent les résultats des recherches en les adaptant à leurs besoins. NOVASYS considère l'ajout les acteurs (i.e. le langage ADL et son interprète, le mopteur ADL) dans une version future de leur propre version du projet SAFARI, TrainingOffice.

6.1 Les résultats d'ADL

Comme nous avons pu constater lors de la création des différents prototypes d'acteurs, ADL a tenu ses promesses. Il facilite l'implémentation des acteurs de par sa nature spécialisée (i.e. il épouse l'architecture des acteurs) et offre des performances d'exécution satisfaisantes (rappelons qu'ADL est précompilé). Nous n'aurions clairement pas pu répondre aux différentes exigences de SAFARI avec un autre langage d'acteurs (i.e. KQML) parce qu'elles étaient précises et nécessitaient un langage fait sur mesure.

Les acteurs ne sont pas restreints aux STI. Ils peuvent être utiles dans des domaines très différents comme tout agent. De même ADL ne restreint en aucune façon les acteurs aux

STI sauf par les primitives qui servent à l'intégration aux autres modules de SAFARI et qui peuvent être parfaitement retirées du langage au besoin. Il est facile de modifier ce langage interprété. ADL peut ainsi facilement s'adapter à de nouvelles fonctionnalités apportées par SAFARI, voire même à un projet complètement différent utilisant les acteurs.

De plus, l'interprète est écrit en C++ entièrement portable, pouvant ainsi être utilisé sur différentes plateformes. Il gère lui-même l'exécution en parallèle des acteurs. Le langage ADL peut alors fonctionner sur toute machine et permet un contôle absolu des acteurs. Ceci peut s'avérer aussi très intéressant lors du développement d'un dévermineur (en mauvais français "déboggueur") qui voudrait interrompre l'exécution de code et examiner l'état des acteurs. Ceci pourrait s'avérer intéressant pour une version future de l'éditeur des acteurs.

6.2 Le potentiel d'ADL pour les propriétés d'éducation et de cognition

Même si l'architecture des acteurs et le langage ADL ne cernent pas le problème de l'établissement des propriétés de cognition et d'éducation, ils apportent quand même certains éléments de solution.

- Nous avons vu l'utilité de la trace des acteurs pour leur permettre d'apprendre de leur comportement passé;
- Nous avons expliqué l'importance du focus des acteurs pour établir l'apprentissage cognitif. Rappelons que pour établir un apprentissage réaliste (ressemblant à celui d'un humain), un acteur ne doit pas être au courant des décisions des autres acteurs: il doit les déduire lui-même;
- Certaines primitives sont prévues pour modifier le code des acteurs pendant leur exécution (voir 3.2.5.5 p. 53). Un acteur peut ainsi examiner et modifier son code. On dit d'un langage qui permet l'examen et la modification de son propre code qu'il est introspectif. Nous croyons que l'introspection d'ADL est nécessaire pour l'établissement de la couche cognitive en général.

- Le fait que le moteur ait un contrôle absolu sur l'exécution des acteurs et qu'un éditeur puisse s'en servir pour dérouler une session et examiner leur état ne peut qu'aider au processus d'éducation.

6.3 Limitations

L'architecture des acteurs amène un couche appelée cognitive, qui s'occupe des différentes activités d'apprentissage machine. Elle offre aux acteurs les propriété d'éducation et de cognition entres autres. Les différents aspects de l'architecture de l'acteur sont bien cernés dans ADL, sauf pour la couche cognitive. Ceci s'explique par le fait que le problème de réaliser l'aspect cognitif de l'acteur est complexe. Ne serait-ce que pour le problème de l'assignation du blâme. Comme nous avons vu, ce problème est encore ouvert. Nous ne connaissons donc pas de solution évidente pour qu'un agent (i.e. un acteur) puisse déterminer quelle est la partie fautive de ses connaissances (i.e. de son code). La recherche d'une telle solution dépasse le cadre de ce mémoire et n'as pas été abordé.

La nature même du langage nous apporte aussi certaines limitations. ADL et son interprète ne s'appliquent qu'aux acteurs. Les utiliser pour modéliser d'autres types d'agents serait une mauvaise idée due à leur nature spécialisée. De plus, malgré le fait que le code ADL passe par une précompilation, son interprétation n'est pas aussi performante que si on avait écrit les acteurs dans un autre langage (i.e. directement en C++). Cependant, le moteur ADL s'est montré suffisamment efficace pour les différents prototpes qui ont été écrits (i.e. le prototype d'apprentissage à deux et celui développé avec NOVASYS).

6.4 Recherches futures

La plus grande difficulté d'un travail futur sur la couche cognitive est d'attaquer le problème d'assignation du blâme, c'est-à-dire d'être capable de dire quel acteur, quelle tâche, bref, quelle partie du code est responsable du problème.

La description des buts est considérée implicite dans le code d'un acteur, mais pour une véritable couche cognitive, il faut penser à une extension du langage permettant de les représenter clairement. L'utilisation d'un arbre de buts serait intéressant à envisager (i.e. un but qui se décompose en sous-buts).

À cause des vastes possibilités de sujets sur lequel il faut apprendre, ainsi que multiples outils d'apprentissage qui risquent de se montrer nécessaires, il est sage de considérer un apprentissage multistratégique. Une autre raison pour laquelle la couche cognitive devrait utiliser un tel apprentissage est que deux projets très proches des acteurs (soit Meta-AQUA pour la propriété cognitive et DISCIPLE pour la propriété d'éducation) font tous deux de même.

La forme d'apprentissage multistratégique de la couche cognitive devrait être de la forme "tool-box", c'est-à-dire que plusieurs outils d'apprentissage se trouvent à être disponibles en même temps et qu'une stratégie implantée dans la couche cognitive devrait faire un choix judicieux sur ces outils pour analyser le comportement de l'acteur.

Cependant, un simple algorithme d'apprentissage n'est pas suffisant pour répondre aux besoins de la couche cognitive. Pour faire un apprentissage cognitif (réaliste), nous devons faire l'apprentissage sur des informations pertinentes extraites des différents exemples (voir 4.3, p. 67). La sélection des informations pertinentes (ou sémantique) est critique pour le bon fonctionnement de l'algorithme d'apprentissage et pour

l'interprétation des résultats. Cette sélection est faite par un expert humain dans l'exemple proposé. La sélection de telles informations par les acteurs dépasse le cadre de cette thèse.

Il reste encore quelques étapes à franchir pour bien établir la couche cognitive. Pour ce faire, il faut aborder des problèmes encore ouverts en apprentissage machine (i.e. l'assignation du blâme) et proposer les différents outils d'apprentissage, définir des sémantiques, ainsi que des stratégies de sélection d'outils d'apprentissage.

Bibliographie

[Aïmeur *et al.*, 95] Aïmeur, E., Frasson, C., Sthiaru-Alexe, C. (1995). *Towards New Learning Strategies In Intelligent Tutoring Systems*. Brazilian Conference of Artificial Intelligence SBIA'95.

[Bradshaw, 97] Bradshaw, J. M. (1997). *Software Agents*. AAAI Press, Menlo Park, CA.

[Burns *et* Capps, 88] Burns, H. L., Capps, C. G. (1988). Foundations of intelligent tutoring systems. Dans M.C. Polson && J.J. Richardson (dir.), Foundations of intelligent tutoring systems, pp. 21-53. Lawrence Erlbaum Associates, Hillsdale, NJ.

[Carbonnell, 70] Carbonnell, R., (1970). *AI in CAI: an artificial intelligence approach to computer aided instruction*. IEEE transactions on man-machine systems, vol 11, pp. 190-202.

[Castelfranchi, 95] Castelfranchi, C. (1995). *Garanties for autonomy in cognitive agent architecture*. Dans Wooldridge, M. et Jennings, N.R., editors, *Intelligent Agents: Theories, Archiectures and Languages* (LNAI vol 890), Springer Verlag, pp. 56-70.

[Chan *et* Baskin, 90] Chan, T. W., Baskin, A.B. (1990). *Learning Companion Systems*. Dans C. Frasson *et* G. Gauthier (Eds.), *Intelligent Tutoring Systems: At the Crossroads of Artificial Intelligence and Education*, ch 1, New Jersey, Ablex Publishing Corporation.

[Cheikes, 95] Cheikes, B. (1995). *GIA: An Agent-Based Architecture for Intelligent Tutoring Systems*. Proceedings of the Forth International Conference on Information and Knowledge Management.

[Collins *et* Michalski, 89] Collins, A. et Michalski, R.S. (1989). *The Logic of Plausible*

Reasoning: A Core Theory. Cognitive Science, Vol. 13, pp. 1-49.

[Cox, 96a] Cox, Michael. (1996). *Introduction*. Thèse de doctorat présenté au Georgia Institute of Technology, ch 1.

[Cox,96b] Cox, Michael. (1996). *A Process Theory Of Understanding And Learning*. Thèse de doctorat présenté au Georgia Institute of Technology, ch 5.

[Danylik, 94] Danylik, A.P. (1994). *GEMINI: An Integration of Analytical and Empirical Learning*. Machine Learning a Multistrategy Approach V.4, Morgan Kaufman Publishers, San Francisco, CA, pp. 107-138.

[Doyle *et* Hayes-Roth, 96] Doyle, P., Hayes-Roth, B.(1996). *Computer-Aided Exploration of Virtual Environments*. In Working Notes of the AAAI-96 Worshop on AlLife. AAAI Press, Menlo Park,CA.

[Finin *et al.*, 94] Finin, T., Fritzson, R., McKay, D., McEntire, R. (1994). *KQML as an Agent Communication Language*. Proceedings of the Third International Conference on Information and Knowledge Management. ACM Press.

[Franklin *et* Graesser, 96] Franklin, S., et Graesser, A. (1996). *Is it an Agent or Just a Program?: A Taxonomy for Autonomus Agents*. Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Springer-Verlag.

[Frasson *et* Aïmeur, 96] Frasson, C., Aïmeur, E. (1996). *SAFARI: a University-Industry Cooperative Project*. International Conference on Success and Pitfall of Knowledge-Based in Real-World Application, Bankok, Thailand, D. Batanov & P. Brezillon, pp. 225-253.

[Frasson *et al.*, 96] Frasson, C., Mengelle, T., Aïmeur, E., Gouardères, G. (1996). *An*

Actor based Architecture for Intelligent Tutoring Systems. ITS'96 Conference, LNCS, No 1086, Springer Verlag, Montréal, pp. 57-65.

[Gilbert *et al.*, 95] Gilbert, D., Aparicio, M., Atkinson, B. (1996). *Intelligent Agent Strategy*. Rapport technique, Reasearch Triangle Park, IBM Corporation.

[Kodratoff, 94] Kodratoff, Yves. (1994). *Induction and the Organization Knowledge*. Machine Learning a Multistrategy Approach V. 4. Morgan Kaufman Publishers, San Francisco, CA, pp. 85-106.

[Lê *et al.*, 98] Lê, T., Gauthier, G.,Frasson, C. (1998) *The Process of Planning for an Intelligent Tutoring System*. The Fourth World Congress on Expert Systems. Mexico. pp. 707-714.

[Mengelle *et al.*, 98] Mengelle, T., De Léan, C., Frasson, C. (1998). *Teaching and Learning with Inteligent Agents : Actors*. ITS-98 Conference, The Fourth International Conference on Intelligent Tutoring Systems. San Antonio, Texas. pp. 284-293.

[Mengelle *et al.*,97] Mengelle, T., Leibu, D.,Frasson, C., Aïmeur,E. (1997). *Processus d'instruction d'acteurs pédagogiques*, Colloque Cinquièmes journées EIAO de Cachan, Hermés, France, pp. 251-262.

[Mengelle *et* Frasson, 96] Mengelle, T., Frasson, C. (1996). *A Multi-Agent Architecture for an ITS with multiple strategies.* 3rd International Conference on Computer Aided Learning and Instruction in Science Engineering CALISCE'96, San Sebastian, Spain, pp. 29-31.

[Michalski, 94] Michalski, Ryszard S. (1994). "Inferential Theory Of Learning" *Machine Learning* a *Multistrategy Approach V. 4* Morgan Kaufman Publishers, San Francisco, CA, pp. 3-61.

[Morik, 94] Morik, K. (1994). *Balanced Cooperative Modeling*. Machine Learning a Multistrategy Approach V.4, Morgan Kaufman Publishers, San Francisco, CA , pp. 295-317.

[Nkambou *et al.*, 96] Nkambou, R., Lefebvre, B., Gaulthier, G.(1996). *A Curriculum-Based Student Model for Intelligent Tutoring System*. Fifth International Conference on User Modelling, Kalia-Kona, pp. 91-98.

[Nkambou, 96] Nkambou, R.(1996). *Modélisation des connaissances de la matière dans un système tutoriel intelligent: modèles, outils et applications*, Thèse de Ph.D. au département d'informatique et de recherche opérationnelle de l'Université de Montréal.

[Plaza *et* Arcos, 94a] Plaza, Enric et Arcos, J. L. (1994). *Integration of Learning into a Knowledge Modeling Framework*. Lecture Notes in Artificial Intelligence. Springer, 867, pp. 355-373.

[Plaza *et* Arcos, 94b] Plaza, Enric et Arcos, J. L. (1994). *Flexible Integration of Multiple Learning Methods into a Problem Solving Architecture*. Lecture Notes in Artificial Intelligence. Springer, 784, pp. 403-406.

[Ram *et* Cox, 94] Ram, Ashwin et Cox, Michael. (1994). *Introspective Reasoning Using Meta-Explanations for Multistrategy Learning*. Machine Learning a Multistrategy Approach V. 4. Morgan Kaufman Publishers, San Francisco, CA, pp. 349-377.

[Rossignol *et al.*, 96] Rossignol, J.-Y., Aïmeur E., Frasson, C. (1996). *Les agents, un aperçu*. Publication départemantale du Département d'Informatique et de Recherche Opérationnelle de l'Université de Montréal (DIRO). Publication 1054.

[Segen, 94] Segen, J. (1994*). GEST: A Learner Computer Vision System That Recognizes*

Hand Gestures. Machine Learning a Multistrategy Approach V. 4. Morgan Kaufman Publishers, San Francisco, CA, pp. 621-634.

[Self, 97] Self, J. (1997). *Students models in computer-aided instruction*. Man-machine studies, vol. 6, pp. 295-352.

[Serroud *et al.*, 95] Serroud, A., Aïmeur, E., et Frasson, C. (1995). *Le stratège*. Rapport semestriel d'activités, Projet SAFARI, pp. 235-249, Universite de Montréal.

[Sharp, 95] Sharp, D. R. (1995). *Machine Learning A Multisrategy Approach*. Submitted to the Department of Mathematics and Computer Science in partial fulfillment of the requirements for the degree of Bachelor of Arts with Honors.

[Tecuci, 94] Tecuci, G. (1994). *An Inference-Based Framework for Multistrategy Learning*. Machine Learning a Multistrategy Approach V. 4. Morgan Kaufmann Publishers, San Francisco, CA, pp.107-138.

[Tecuci *et* Hieb, 96] Tecuci, G, et Hieb, Michael R.(1996). *Teaching Intelligent Agents: the Disciple Approach*. International Journal of Human-Computer Interaction.

[Veloso *et* Carbonnell, 94] Veloso, M. et Carbonnell, J. (1994). *Case-Based Reasoning in PRODIGY*. Machine Learning a Multistrategy Approach V. 4. Morgan Kaufman Publishers, San Francisco, CA, pp. 523-548.

[Wisniewski *et* Medin, 94] Wisniewski, E. J. et Medin, D. L. (1994). *The Fiction and Nonfiction of Features*. Machine Learning a Multistrategy Approach V. 4. Morgan Kaufman Publishers, San Francisco, CA, pp. 63-84.

[Wooldridge *et* Jennings, 95] Wooldridge, M. et Jennings, N. (1995). *Intelligent Agents: Theory and Practice*. Knowledge Engineering Review, Vol 10, No2, pp. 115-152.

Annexe A Grammaire du langage de description d'acteur

Voici la grammaire complète du langage d'implantation d'acteur qui est interprété par le moteur.

On peut remarquer que le langage ressemble à un croisement de SmallTalk et de C.

A.1 Structure générale du langage

Commençons par expliquer comment se présente généralement du code

A.1.1 Les commentaires

Dans ce langage, les commentaires se font comme en C++. "//" indique que le reste de la ligne est un commentaire et tout bloc de texte compris entre une "/*" et un "*/" en est un aussi.

A.1.2 Les constantes

Les constantes numériques (entiers) ne sont constituées que par les chiffres de 0 à 9.

Les chaînes de caractères sont délimités par des guillemets (i.e. "Une chaîne").

Il existe deux booléens: true et false.

Il existe une valeur nulle: nil

A.1.3 Les variables

Ce langage d'acteur est un langage atypé. Tout nom d'identificateur doit commencer par une lettre.

A.1.4 Les structures

Nous pouvons aussi faire une liste d'expressions. Pour ce faire, nous plaçons une série de

constantes, variables, structures ou appel de primitives entre parenthèses séparées par des virgules, le tout à l'intérieur d'accolades (i.e. {1,2,3, "Go", maVariable}).

De plus, il est possible de manipuler des enregistrements. Nous pouvons les créer avec la fonction **make** et accéder aux différents champs avec la fonction d'extraction "." (le point).

A.1.5 Le passage en paramètre

Tout paramètre est passé par valeur. Pour l'instant, il n'y a pas de passage par référence et pas de pointeur non plus. Chaque paramètre est une *expression*. Une expression consiste en une variable ou une constante. Une expression peut aussi être un appel à une primitive. Un tel appel doit être mis entre parenthèses. La seule exception est une expression avec des primitives binaires avec paramètres des deux côtés (opérations arithmétiques). Ainsi écrire " 1 + 2 * b" ou "1 + (2 * b)" revient au même.

A.1.6 Les blocs de code

Un bloc de code est un ensemble d'instructions. Le bloc est délimité par les accolades { et }. Chaque instruction est séparée par un point-virgule ;. Une instruction peut être une structure de contrôle ou un appel à une primitive.

Un bloc de code peut contenir des variables locales. Pour les déclarer, il suffit de mettre en première ligne le mot clé **VAR**, suivi des noms de variables séparés par des espaces ou des virgules. On finit la déclaration des variables par un point-virgule.

Un bloc de code ressemble à ce qui suit.

{
<phrase1>;
<phrase2>;
...

<phrase>;

}

où **<phrase>** ressemble à

- une primitive;
- une structure de contrôle;
- une assignation de variable;
- un retour de valeur (i.e. **return <expression>**).

A.1.7 Les structures de contrôle

Nous avons pour l'instant deux structures de contrôle: le while et le if.

Nous les écrivons comme suit:

while (<expression>) do <action>

if (<expression>) then <action> else <action>

où **<action>** est: <bloc de code>|<instruction>

A.2 Les primitives

Ce langage vient avec une série de primitives. Elles peuvent prendre un nombre variable de paramètres et chacune est identifiée comme en SmallTalk. Il existe quelques exceptions.

Exemple: maPrimitive a monDeuxièmeParamètre { b , "Chaîne" } monTroisièmeParamètre (a< b)

Voici la liste présente des primitives. Commençons d'abord par les exceptions.

Les exceptions sont les différents opérateurs arithmétiques et logiques.
Les primitives +, -, * et /. Prennent deux expressions numériques et en donnent un. La concaténation de chaînes de caractères se fait avec +. En plus si + a une expression numérique et une chaîne de caractères, l'expression numérique sera transformée en chaîne de caractères.
Les primitives ==,<,>,<= et >=. Prennent deux expressions numériques (ou deux chaînes de caractères) et donnent une expression booléenne. La primitive = prend une variable à gauche et une expression à droite.
Ces fonctions s'écrivent comme en C.

Trois autres fonctions ayant rapport avec les structures sont aussi à annoncer.

- **make** <type>

<type> est le nom d'une structure déclarée dans le code. Ce nom n'est pas délimité par des guillemets. La fonction retournera une structure du bon type ou *nil* s'il n'existe pas;

- l'extraction d'un champ de structure. L'extraction d'un champ se fait avec le point.

À gauche du point, on doit avoir une variable contenant la structure. À droite, le nom du champ. Ce nom n'est pas délimité par des guillemets. Exemple: MaVariable.UnChamp. Une telle expression est considérée comme une variable;

- <expression> **istype** <type>

<type> est un type de structure défini dans le code, ou une des constantes suivantes: **integer**, **boolean**, **string** et **list**. Les dernières constantes tiennent pour un entier, un booléen, une chaîne de caractères et une liste. Cette fonction permet de vérifier si une valeur est d'un certain type;

Maintenant, voici un répertoire des différentes primitives du langage. On y rappelle leur effet et on donne leur grammaire. Lorsqu'un paramètre est souligné (i.e. <param>) cela indique que le paramètre doit contenir une expression donnant chaîne de caractères. Sinon, à moins d'indication contraire, les paramètres sont des expressions ordinaires.

- **ask** <question>

demande une question à l'apprenant de la part de l'acteur. La valeur de retour sera un booléen si l'apprenant répond (oui ou non) et *nil* sinon;

- **calltask** <alias de tâche> [with {<param1>, <param2> ... <param>}][passive][ifno <que faire si échec>]

fait appel à une sous-tâche. L'option passive permet à la fonction d'être passive au lieu d'être active. Si l'option est utilisée, alors la primitive retournera un identificateur au lieu du résultat de l'appel. Il faut alors utiliser **existreply** et **getreply** pour récupérer le résultat. **<que faire si échec>** peut être deux choses:

  - ignore: continue d'exécuter la tâche (comportement par défaut);
  - abort: arrête la tâche et génère un échec à son tour.;

- **existbb** [<titre>] [from <nom d'acteur>][category <catégorie>]

renvoie un booléen indiquant si une information correspondant aux critères se trouve dans le tableau noir (voir **getbb** pour en savoir plus);

- **existreply** <identificateur> [from <nom d'acteur>]

indique s'il y a une réponse à une requête. Elle n'est pas bloquante: elle retourne false s'il n'y a pas de réponse (voir **getreply** pour en savoir plus);

- **getbb** [<titre>] [from <nom d'acteur>][category <catégorie>]

renvoie la première information correspondant aux critères de recherche. **<titre>** et **<catégorie>** sont les informations placés par un **putbb**. Par défaut, ces valeurs sont "NoName" et "AllActors" ce qui veut dire que le nom est sans importance et que tous les acteurs sont cherchés. **<nom d'acteur>** est le nom de l'acteur qui l'a mis dans le tableau noir. **<catégorie>** a pour valeur par défaut "NoCategory". Si on spécifie "AllCategory", toutes les catégories seront fouillées. Cette information est retirée du tableau noir. Voir aussi **lookbb**;

- **getelement** <entier> of <liste>

<entier> est un entier et **<liste>** est une liste. Cette primitive donne un élément indexé de la liste (le premier élément a un indice de 1);

- **getevent**

récupère l'événement détecté avant la dernière évaluation des situations types. Si aucun événement n'a été détecté, on obtient *nil*. Si utilisée dans une situation type, cette fonction détecte les événements produits après la dernière évaluation des situations types. Les tâches visibles par l'acteur (à travers son focus global) seront aussi vus dans les événements dans des structures de type *task* ayant pour paramètres *name* pour le nom de la tâche, *actor* pour le nom de l'acteur, et *id* pour l'identificateuir de tâche (peut servir dans un appel de **lookact** par exemple). Cette primitive ne pourra pas voir les événements eliminés par **seteventfilter**;

- **getglobalvar** <nom>

donne le contenu d'une variable globale (déclarée dans le bloc VAR d'un acteur ou d'une stratégie). **<nom>** est le nom de la variable et ne doit pas être délimité par des guillemets;

- **getmem** <titre> in <nom de variable de mémoire>

Renvoie l'entier placé par un appel de **putmem** avec les mêmes paramètres et retire cette information de la mémoire. Voir aussi **lookmem**;

- **getnameact**

Donne le nom de l'acteur exécutant la tâche;

- **getreply** <identificateur> [from <nom d'acteur>]

<identificateur> est ce retourne **calltask** ou **req** lors d'appels passifs. Cette fonction émet le résultat d'une requête ou appel de fonction. Elle restera bloquante tant qu'il n'y aura pas de réponse;

- **getstudentinfo** attention | alert | anxious | interest | self-confidence | self-esteem | workmemory | generalknowledge | associativelearning | inductivereasoning | quickness | defaultknowledge | metacognition | impulsive | serial | verbal | passive | random | conformist

permet de récupérer de l'information affective, cognitive ou conative chez l'apprenant. Les six premiers mots-clés (*attention* à *self-esteem*) donnent une information affective. Les sept suivants (*workmemory* à *metacognition*), une information cognitive et les six derniers (*impulsive* à *conformist*) , une information conative Les valeurs retournées sont des entiers de 0 à 10 indiquant de degré d'importance de l'information demandée sur l'apprenant;

- **getstudentknowledge** objective | resourcepassed | resourcescore | capacity | maxlevelcap <identificateur>

permet de récupérer de l'information sur les connaissances de l'apprenant envers le cours. Le premier paramètre indique le type d'information recherché dans l'information désigné par <identificateur>. Si le premier paramètre est *objective*, la primitive retournera si l'objectif en question a été atteint. Il en va de même avec *resourcepassed* pour une ressource. Avec *resourcescore*, on obtiendra un score de 0 à 10 indiquant la réussite de l'apprenant avec la ressource, et *maxlevelcap* indiquera le niveau maximum qu'une capacité peut atteindre;

- **gettracetaskname** <élément de trace>

le paramètre de cette fonction est ce que retourne un appel de **lookact**. Cette fonction indique le nom de la tâche de l'élément de trace;

- **gettracetaskparam** <élément de trace>

le paramètre de cette fonction est ce que retourne un appel de **lookact**. Cette fonction donne les paramètres ayant servi à l'appel de la tâche sous forme de liste;

- **gettracetime** <élément de trace>

le paramètre de cette fonction est ce que retourne un appel de **lookact** Cette fonction indique le temps de l'élément de trace. Ce temps peut servir à un appel futur de **lookact** au paramètre jumpto;

- **ifno**

retourne un booléen indiquant si on a refusé de répondre à la dernière requête ou appel de sous-tâche active;

- **lookact** [{<nom d'acteur>, <nom d'acteur>, ...}] [for <filtre de tâches>] [jumpto <temps>][notended]

<**filtre de tâche**> peut être deux choses

  - une combinaison de quatre mots clés séparés par des "|". Ces mots sont **Operating**, **Abstract**, **Control** et **Cognitive**. L'ensemble ne doit pas comporte d'espaces;
  - une liste de noms de tâches. ex: { "tache1" , "tache2" }.

<**temps**> est un temps à partir duquel on veut faire la recherche. Ce temps doit être récupéré d'une recherche subséquente avec **gettracetime**. **notended** indique qu'on est aussi intéressé pas les tâches non terminées. Cette fonction renvoie un élément de trace correspondant aux critères de recherche. Voir les fonctions **gettracetaskname**, **gettracetaskparam** et **gettracetime**;

- **lookbb** [<titre>] [from <nom d'acteur>][category <catégorie>]

fait la même chose que **getbb** sauf qu'on ne retire pas l'information du tableau noir;

- **lookmem**

fait la même chose que **getmem** sauf qu'on ne retire pas l'information de la mémoire.

- **matchevent** [<var> =] <type> , [<var> =] <type> ... [<var> =] <type>

permet de vérifier si les derniers évènements sont des structures ayant les types <type> en

primitive retourne la réponse, sinon, on retourne *nil*;

- **seteventfilter** {<filtre1>,<filtre2> ... <filtre>}

détermine quelles sont les événements que l'acteur ne veut pas pouvoir détecter avec la fonction **getevent**. Par défaut, tous les événements sont détectés. Les événements dont les noms sont dans le filtre d'événement seront ignorés par l'acteur;

- **setexpression** <expression>

change l'expression de l'acteur. Les différentes expressions possibles ne sont pas définies dans ce langage. Elles le sont dans le module de l'interface graphique des acteurs;

- **setglobalvar** <nom> to <expr>

détermine le contenu d'une variable. <nom> ne doit pas être délimité par des guillemets. Voir **getglobalvar** pour plus d'information;

- **updatecapacity** <idcapacite> to <niveau>

met à jour une capacité <idcapacite> au niveau <niveau>;

- **updateobjective** [<idobjective> to] <niveau>

met à jour un objectif <idobjective> au niveau <niveau>. Par défaut, <idobjective> est l'objectif qui est à atteindre par les acteurs;

- **updateressource** [<idressource> to] <niveau> [context <contexte>]

met à jour une capacité <idcapacite> au niveau <niveau>. Par défaut, <idressource> est la ressource courante gérée par les acteurs. <contexte> peut contenir des informations sur l'état de la ressource (sauvegarde pour une session ultérieure). Cette primitive termine la ressource courante et cause la prochaine ressource à être chargée. Elle retourne un booléen indiquant s'il y a bien une nouvelle ressource qui est chargée.

respectant l'ordre. Elle retournera un booléen annonçant si c'est le cas ou non. Si oui, pour tout <type> précédé d'un <var> =, la variable <var> se trouvera assigné la structure <type>. Si <type> est "*", tout type conviendra. Les événements vus ne seront que les tâches pouvant passer par le focus de l'acteur et par le filtre d'événements (voir **getevent**);

- **putbb** <information> as <titre> [category <catégorie>]

<titre> est un nom à la discrétion du programmeur **<catégorie>** est optionnel. Sa valeur par défaut est "NoCategory". Ajoute l'information dans le tableau noir;

- **putmem** <entier> as <titre> in <nom de variable de mémoire>

ajoute un entier dans la mémoire associative de l'acteur. Le nom de la variable doit se trouver dans le bloc **MEMORY** de l'entité exécutant la primitive (i.e. un acteur ou la stratégie);

- **req** <titre> [with {<param1>,< param2> ... <param>}] [to {<nom d'acteur>, <nom d'acteur>, ...}] [passive] [ifno <que faire si refus>]

émet une requête. Et renvoie un identificateur. **<titre>** en est le nom. **<nom d'acteur>** est le nom d'acteur à qui on veut faire la requête. Par défaut, on l'envoie à tout le monde. **passive** permet de rendre la requête passive (par défaut, elle est active). **<que faire si refus>** peut être deux choses: **abort** ou **ignore**. Par défaut, c'est **ignore**. Pour en savoir plus, voir la primitive **calltask**;

- **say** <message>

annonce que l'acteur veut donner un message a l'apprenant. Le texte contenu dans <message> lui sera transmis;

- **send** <structure>

envoie la structure <structure> comme message à la ressource. Le type et le contenu de la structure contiennent le message et sa signification. Ce message sera considéré comme un événement par la ressource. Si la ressource définit une réponse au message envoyé, cette

A.3 Les structures

Une structure se déclare de façon très proche à qu'une structure en C. Elle se fait comme suit.

Type <nom de structure>

{<champ1>, <champ2> ... <champ>}

où **<champ>** est le nom d'un champ. Une telle déclaration crée un nouveau type de valeur. Voir les fonctions **make**, **istype** et la fonction d'extraction de champs (.).

Il y deux structures prédéfinies dans ce langage.

- **conflict** avec les champs **actor** et **ts**

Cette structure ne servira que dans du code mis directement dans la stratégie (pour la résolution de conflit). **actor** contient le nom de l'acteur et **ts** celui de sa situation type en conflit;

- **task** avec les champs **name**, **actor** et **id**.

Cette structure est un type *d'événement* contenant une tâche. **name** contient son nom, actor l'acteur qui l'exécute et **id** son identificateur de tâche (peut servir dans **lookact** pour récupérer plus d'informations).

A.4 Les tâches

La déclaration d'une tâche ressemble beaucoup à celle d'une procédure en C.
Elle se fait comme suit.

<type de tâche> <nom de tâche> (<param 1>,<param 2>...<param>)

<bloc de code>

où **<type de tâche>** est l'un des mots clés suivants: **Operating**, **Abstract**, **Control**, **Cognitive**. Une tâche peut ne pas avoir de paramètres. Néanmoins, il lui faudra quand même des parenthèses. Finalement, suit un bloc de code comme expliqué plus haut. Les

noms des paramètres n'ont pas à se répéter. Elles sont déjà considérées comme des variables locales.

A.5 Les stratégies

La déclaration d'une stratégie se fait comme suit:

Strategy <Nom de la Stratégie>
{
SOCIETY { <nom d'acteur> [begin <nom de tâche>] , <nom d'acteur> ... }
[CONFLICT
[VAR <nom de var> , <nom de var> ...]
[MEMORY { <nom de var> , <nom de var> ... }]
<bloc de code>
]
}

Le code de conflit va être exécuté de façon atomique. Il y est donc interdit d'appeler une tâche ou de faire une requête. De plus, ce code n'est pas exécuté par un acteur, et n'est pas la description d'une tâche. Toutes les primitives prenant cela en considération sont aussi interdites.
Le code possède un paramètre nommé conflict, contenant une structure de type conflict (voir A.3, p. A-11). Ce code devra retourner le nom de l'acteur pouvant déclencher sa situation type.

La partie **VAR** est la partie des variables globales atteignables que par le code de conflit et la partie **MEMOIRE** est la partie de variables de mémoire.

A.6 Les situations types

Une situation type se déclare comme suit:

TypicalSituation <nom de situation type>

ALIAS <nom de tâche>

[<un focus>]

EVALUATOR <bloc de code>

Le bloc de code doit renvoyer un booléen et va être exécuté de façon atomique. Il y est donc interdit d'appeler une tâche ou de faire une requête.

A.6.1 Les Focus

Un focus s'écrit comme suit:

FOCUS

<restriction de temps> { <niveau de perception> , <niveau de perception> ... }

où **<restriction de temps>** peut être:

- begin
- <nom d'acteur> did <positionnement> [<description de tâche>]

<positionnement> peut être

- first
- last
- <un entier> times

<description de tâche> peut être

- un alias de tâche (nom de tâche)
- <filtre de tâche> task, une combinaison de quatre mots clés séparés par des "|". Ces mots sont Operating, Abstract, Control et Cognitive. L'ensemble ne doit pas comporter d'espaces et est suivi du mot clé task.

<niveau de perception> s'écrit: <nom d'acteur> [at <filtre de tâche>][request]

<filtre de tâche> est décrit un peu plus haut.

A.7 Les acteurs

Un acteur se déclare comme suit

Actor <Nom d'acteur>

{

[VAR <nom de var> , <nom de var> ...]

[MEMORY { <nom de var> , <nom de var> ... }]

[ALIAS { {<nom de tâche> , <alias>} , { <nom de tâche> , <alias> } ... }]

[REQUEST { {<nom de requête> [, <nom de tâche>]} , {<nom de requête> [, <nom de tâche>] } ... }]

[<description de focus>]

[<description de situation type> <description de situation type> ...]

}

La partie **VAR** est la partie des variables globales atteignables que par le code de conflit et la partie **MEMOIRE** est la partie de variables de mémoire. La partie **ALIAS** est l'ensembles des alias de tâche (i.e. si on appelle une tâche avec un nom d'alias, on déclenche la tâche auquel elle se réfère) et la partie **REQUEST** est le dictionnaire de requêtes de l'acteur (i.e. l'ensemble des requêtes auquel l'Acteur peut répondre et la tâche qu'il déclenchera pour y répondre). Si dans la partie **REQUEST** le nom de la tâche n'est pas spécifié, les deux noms sont considérés identiques.

A.8 L'organisation d'un fichier source

Un fichier source contient une série de déclarations. Pour l'instant, il n'y a pas d'ordre exigé entre ces déclarations. Ces déclarations sont soit des stratégies, des tâches, des

acteurs ou des structures. Les autres types de déclarations y seront imbriqués. Le programme est conçu pour pouvoir avoir du code source partagé en plusieurs fichiers. On peut ainsi faire plusieurs compilations successives par le modèle. Ces compilations seront cumulatives.

A.9 La grammaire des structures envoyées aux ressources

La traduction se fait dans les deux sens. On passe d'une chaîne de caractères à une structure (de type **Structure**) par la méthode **ReadExpression(Valeur Val, char *str)**, et dans le sens contraire par **WriteExpression(Valeur Val, char *str)**. La première méthode est aidée par les méthodes **ListeValeur ReadListe(char *str)** et **Structure ReadStruct(char *str)** qui peuvent respectivement lire une liste de valeurs et un structure.

Une expression s'écrit comme suit: **<tête> <corps>**

où **<tête>** est **event** | **req <identificateur>**

<corps> est **<entier>** | **true** | **false** | "Une chaîne entre guillemets" | **<liste>** | **<structure>**

<liste> s'écrit **(<corps>, <corps> ... <corps>)**

<structure> s'écrit **<nom de structure> { <corps>, <corps> ... <corps> }**

Annexe B Le code des acteurs utilisé dans la mammographie

On peut remarquer l'absence de protocole d'échange de structures entre la ressource et les agents. Ceci vient du fait que ce prototype est l'un des tous premiers. Cependant, nous avons déjà un protocole de communication permettant une séparation de l'expertise des acteurs et de celle du domaine de la ressource.

```
// ************************************************************************
// La strategie
// ************************************************************************

Strategy Perturbateur
{
        SOCIETY
        {
                SelectRessource begin Depart,
                Tuteur ,
                Perturbateur,
                Apprenant,
                Ressource
        }
}

// ************************************************************************
// Les acteurs
// ************************************************************************

// ------------------------------------------------------------------

Actor Tuteur
{
        MEMORY { CORRIGE }
        REQUEST { { InitAttitude, TuteurInitAttitude } }

        // Situation declenche des qu'un probleme a ete pose
        TypicalSituation ProblemePose
        ALIAS PresenterProbleme
        EVALUATOR
        {
                VAR problemePose?;
                problemePose?  = getbb "ProblemePose";

                if ( problemePose? == nil ) then return false;
                else return true;
```

```
        }

        // Situation declenche apres une action de l'apprenant
        TypicalSituation ReactionPerturbateur
        ALIAS TuteurIntervient?
        FOCUS Perturbateur did last PerturbateurIntervient? { Tuteur at Operating task, Perturbateur at
        Operating task }
        EVALUATOR
        {
                VAR diagnosticAnnonce?,aide?, tuteurIntervenu?;

                diagnosticAnnonce? = lookact for { "PerturbateurIntervient?" };
                if (diagnosticAnnonce? == nil ) then return false;
                else
                {
                        aide? = lookbb "TUTEUR_INTERVIENT";
                        if ( aide? == 0 ) then return false;
                        else
                        {
                                putbb 0 as "TUTEUR_INTERVIENT";
                                return true;
                        }
                }
        }
}
// ---------------------------------------------------------------------

Actor Perturbateur
{
        REQUEST { { InitAttitude, PerturbateurInitAttitude } ,{PerturbateurBeginProbleme} }

        // Situation declenche apres une action de l'apprenant
        TypicalSituation ActionApprenant
        ALIAS PerturbateurIntervient?
        FOCUS Ressource did last AnnonceDiagnostic { Perturbateur at Operating task ,
                                                     Ressource at Operating task }
        EVALUATOR
        {
                VAR diagnosticAnnonce?, perturbateurIntervenu?;

                diagnosticAnnonce? = lookact for { "AnnonceDiagnostic" };
                if (diagnosticAnnonce? == nil ) then return false;

                perturbateurIntervenu? = lookact for { "PerturbateurIntervient?" };
                if ( perturbateurIntervenu?  == nil ) then return true;
                else return false;
        }
}

// ---------------------------------------------------------------------

Actor Apprenant
{
        MEMORY { MODELE_APPRENANT }
```

```
        REQUEST {  { MiseAJourModele }, { ConsulterModele } }
}

// ---------------------------------------------------------------

Actor Ressource
{
        MEMORY { RESSOURCE_MEMORY}
        REQUEST { { Send } , { SendAndWait } ,  { PartirResourceManager } , { PoserProbleme } }
}

// ***************************************************************************
// Les taches
// ***************************************************************************

// ---------------------------------------------------------------
// Utilitaires
// ---------------------------------------------------------------

Operating SendMessage ( message )
{
        req "Send" with { message } to { "Ressource" }  ;
}

Operating SendMessageAndWait ( message )
{
        return req "SendAndWait" with { message } to { "Ressource" }  ;
}

// ---------------------------------------------------------------
// Taches de SelectRessouce
// ---------------------------------------------------------------

Operating Depart ( )
{
        req "PartirResourceManager" to { "Ressource" } ;
        calltask "InitStrategy";
        calltask "ChoisirRessource";
}

Operating ChoisirRessource ( )
{
        req "PoserProbleme" to { "Ressource" };
}

Operating InitStrategy ( )
{
        calltask "SendMessage" with { "strategy perturbateur" } ;
        req "InitAttitude" to { "Tuteur" } ;
        req "InitAttitude" to { "Perturbateur" } ;
}

// ---------------------------------------------------------------
// Taches de Ressource
```

```
// ---------------------------------------------------------------

Operating PartirResourceManager ( )
{
        putmem ( calltask "InitResourceManager" ) as "socket" in  "RESSOURCE_MEMORY" ;
}

Operating PoserProbleme ( )
{
        calltask "SendMessage" with { "probleme Exercice_1" };
        putbb 1 as "ProblemePose";
        calltask "WatchResource" passive;
}

// ---------------------------------------------------------------

Operating Send ( cmd )
{
        VAR socket;
        socket = lookmem "socket" in "RESSOURCE_MEMORY" ;
        calltask "SendSocket" with { socket, cmd };
}

Operating SendAndWait ( cmd )
{
        VAR socket;
        socket = lookmem "socket" in "RESSOURCE_MEMORY" ;
        return (calltask "SendSocketAndWait" with { socket, cmd });
}

Operating WatchResource ( )
{
        VAR wait , read , socket;

        socket = ( lookmem  "socket" in  "RESSOURCE_MEMORY" ) ;

        while ( socket != nil ) do
        {
                wait = ( calltask "MessageWaiting?" with { socket } ) ;

                if ( wait != nil ) then
                {
                        read = ( calltask "ReadMessage" with { socket } ) ;
                        if ( read == nil ) then socket = 0;
                else
                        calltask  "AnnonceDiagnostic" with { read };
                }
        }
}

Operating AnnonceDiagnostic ( buffer )
{
        VAR name, poly, value, end;
```

```
        poly =  calltask "IsPoly?" with { buffer };
        name = calltask "GetTagName" with { buffer };
        value = calltask "GetValue" with { buffer };

        if(poly) then putbb poly as "STUDENT_POLY";
        else putbb 0 as "STUDENT_POLY";

        req "MiseAJourModele" with {name,value} to {"Apprenant"};
}

// ---------------------------------------------------------------
// Taches du perturbateur
// ---------------------------------------------------------------

Operating PerturbateurInitAttitude ( )
{
        calltask "SendMessage" with { "addattitude perturbateur normal images/pierre1.gif" };
        calltask "SendMessage" with { "addattitude perturbateur fache images/pierre2.gif" };
        calltask "SendMessage" with { "addattitude perturbateur heureux images/pierre3.gif" };
        calltask "SendMessage" with { "addattitude perturbateur surpris images/pierre4.gif" };
}

Operating PerturbateurBeginProbleme ( )
{
        calltask "SendMessage" with { "message perturbateur Hello! I'm happy to meet you" };
}

Operating PerturbateurIntervient? ( )
{
        VAR val,which, cmp;

        which = lookbb "RECENT_DIAGNOSIS";

        val = (req "ConsulterModele" with { which } to {"Apprenant"});
        if (val >= 3) then
                calltask "PerturbateurPerturbation" with { which };
        else
                calltask "PerturbateurSuggestion" with { which };
}

Operating PerturbateurPerturbation ( which )
{
        VAR buffer,pert,answer,next_just,done,studPoly;

        //Send the perturbation (stored in pert)
        buffer = "exercise perturbe " + which;
        pert = calltask "SendMessageAndWait" with { buffer };

        if(pert == "-1") then putbb 1 as "TUTEUR_INTERVIENT";
        else
        {
                // Change attitude
                calltask "SendMessage" with {"attitude perturbateur heureux"};
                //Is the student in agreement?
```

```
        answer = calltask "SendMessageAndWait" with { "accord?" };
        if ((calltask "StringCompare" with {answer,"no"})==0) then
        {
                //No, so repeat justifying until he is or until there are
                //no more justifications
                done = false;
                next_just = "0";
                while (done == false) do
                {
                        buffer = "exercise justifie_perturbation " + which + " " + pert + " " +
                        next_just;

                        next_just = (calltask "SendMessageAndWait" with { buffer });
                        if(next_just == "-1") then
                        {
                                done = true;
                                putbb 1 as "TUTEUR_INTERVIENT";
                        }
                        else
                        {
                                // Give up yet?
                                answer = calltask "SendMessageAndWait" with { "accord?" }
                                if (answer == "no") then  done = false;
                                else
                                {
                                        // God yes!
                                        done = true;
                                        putbb 0 as "TUTEUR_INTERVIENT";

                                        buffer = "exercise undo " + which + " ";

                                        studPoly = lookbb "STUDENT_POLY";
                                        if (studPoly != 0) then
                                                buffer = buffer + studPoly
                                else buffer = buffer + "none" ;

                                        calltask "SendMessage" with {buffer};
                                }
                        }
                }
        }
        else
        {
                putbb 0 as "TUTEUR_INTERVIENT";
                buffer = "exercise undo " + which + " ";
                studPoly = lookbb "STUDENT_POLY";
                if (studPoly != 0) then
                        buffer = buffer + studPoly;
                else buffer = buffer + "none" ;
        }
        calltask "SendMessage" with {buffer};
        putbb as 0 "TUTEUR_INTERVIENT";
}
calltask "SendMessage" with {"attitude perturbateur normal"};
```

```
}

Operating PerturbateurSuggestion ( which )
{
        VAR buffer,pert,answer,next_just,done,studPoly;

        //Send the perturbation (stored in pert)
        buffer = "exercise suggere " + which;
        pert = calltask "SendMessageAndWait" with { buffer };

        if(pert == "-1") then putbb 1 as "TUTEUR_INTERVIENT";
        else
        {
                // Change attitude
                calltask "SendMessage" with {"attitude perturbateur heureux"};
                //Is the student in agreement?
                answer = calltask "SendMessageAndWait" with { "accord?" };
                if (answer == "no") then
                {
                        //No, so repeat justifying until he is or until there are
                        //no more justifications
                        done = false;
                        next_just = "0";
                        while (done == false) do
                        {
                                buffer = "exercise justifie_suggestion " + which + " " + pert + " "
                                next_just;
                                next_just = (calltask "SendMessageAndWait" with { buffer });
                                if(next_just == "-1")==0) then
                                {
                                        done = true;
                                        putbb 1 as "TUTEUR_INTERVIENT";
                                }
                                else
                                {
                                        // Give up yet?
                                        answer = calltask "SendMessageAndWait" with { "accord?" };
                                        if (answer == "no")==0) then
                                                done = false; // Nope!
                                        else
                                        {
                                                // God yes!
                                                done = true;

                                                putbb 0 as "TUTEUR_INTERVIENT";

                                                buffer = "exercise undo " + which + " ";
                                                studPoly = lookbb "STUDENT_POLY";
                                                if (studPoly != 0) then
                                                        buffer = buffer + studPoly;
                                                else
                                                        buffer = buffer + "none" ;
                                        }
                                        calltask "SendMessage" with {buffer};
```

```
                                        }
                                }
                        }
                }
                else
                {
                        putbb 0 as "TUTEUR_INTERVIENT";
                        buffer = "exercise undo " + which + " ";
                        studPoly = lookbb "STUDENT_POLY";
                        if (studPoly != 0) then
                                buffer = buffer + studPoly;
                        else
                                buffer = buffer + "none" ;
                        calltask "SendMessage" with {buffer};
                        putbb as 0 "TUTEUR_INTERVIENT";
                }

                calltask "SendMessage" with {"attitude perturbateur normal"};
}

// ---------------------------------------------------------------
// Taches du tuteur
// ---------------------------------------------------------------

Operating TuteurInitAttitude ( )
{
        calltask "SendMessage" with { "addattitude tuteur normal images/anne1.gif" };
        calltask "SendMessage" with { "addattitude tuteur fache images/anne2.gif" };
        calltask "SendMessage" with { "addattitude tuteur heureux images/anne3.gif" };
        calltask "SendMessage" with { "addattitude tuteur surpris images/anne4.gif" };

        calltask "SendMessage" with { "attitude tuteur normal" };
}

Operating PresenterProbleme ( )
{
        calltask "SendMessage" with { "exercise motive-probleme" };
        req "PerturbateurBeginProbleme" to {"Perturbateur"};
}

Operating TuteurIntervient? ( )
{
        VAR val,which;

        which = lookbb "RECENT_DIAGNOSIS";
        val = (req "ConsulterModele" with { which } to {"Apprenant"});

        if (val == 1) then
                calltask "TuteurCorrigeFort" with { which };
        else
        {
                if (val <= 3) then calltask "TuteurCorrige?" with { which };
                if (val == 5) then calltask "TuteurFelicite" with { which };
        }
```

```
}

Operating TuteurFelicite ( which )
{
        VAR buffer;

        calltask "SendMessage" with { "attitude tuteur heureux" };
        calltask "SendMessage" with { "message tuteur Well done!" };

        buffer = "exercise felicite " + which ;
        calltask "SendMessage" with { buffer };
        calltask "SendMessage" with { "attitude tuteur normal" };
}

Operating TuteurCorrige? ( which )
{
        VAR secondTime;

        secondTime = lookmem which in "CORRIGE" ;

        if (secondTime) then calltask "TuteurCorrigeFort" with { which };
        else calltask "TuteurCorrigeFaible" with { which };
}

Operating TuteurCorrigeFaible ( which )
{
        VAR buffer,studPoly;

        calltask "SendMessage" with {"attitude tuteur surpris"};
        putmem 1 as which in "CORRIGE";
        buffer = "exercise corrige " +  which + " 1";

        studPoly = lookbb "STUDENT_POLY";
        if (studPoly != 0) then
                buffer = buffer + studPoly;


        calltask "SendMessage" with { buffer };
        calltask "SendMessage" with {"attitude tuteur normal"};
}

Operating TuteurCorrigeFort ( which )
{
        VAR buffer,studPoly;

        studPoly = lookbb "STUDENT_POLY" ;

        calltask "SendMessage" with {"attitude tuteur fache"};
        putmem 1 as which in "CORRIGE";
        buffer = "exercise corrige " + which + " 2";
        if (studPoly != 0) then
                buffer = calltask "StringConcatenate" with { buffer,studPoly};

        calltask "SendMessage" with { buffer };
```

```
        calltask "SendMessage" with {"attitude tuteur normal"};
}

// ---------------------------------------------------------------------
// Taches de l'apprenant
// ---------------------------------------------------------------------

Operating MiseAJourModele ( name, val )
{
        putmem val as name in "MODELE_APPRENANT";
        putbb name as "RECENT_DIAGNOSIS";
}

Operating ConsulterModele ( name )
{
        return (lookmem name in "MODELE_APPRENANT");
}
```

Annexe C Le code des acteurs de l'exemple de l'apprentissage à deux

Cet exemple n'est pas une stratégie pédagogique en tant que telle, mais un premier prototype en ADL illustrant comment l'auteur voit l'apprentissage au sein d'un acteur.

```
// ***********************************************************************
// La strategie
// ***********************************************************************

Strategy ApprentissageADeux
{
 SOCIETY { Traceur, Peintre}
}


// ***********************************************************************
// Les Acteurs
// ***********************************************************************

Actor Traceur
{

        VAR TermineLettre;

        TypicalSituation ChoisirLettre
        ALIAS ChoisirForme
        FOCUS Traceur did last ChoisirForme {Traceur at Abstract|Operating task, Peintre at Operating
task}
        EVALUATOR
        {
                VAR tache;
                tache=lookact for {"Colorie"};
                if ((tache == nil) && (lookact != nil)) then return false;
                if ((tache != nil) && (lookact for {"Dessine"} == nil)) then  return false;

                if (tache != nil) then setglobalvar TermineLettre to true;
                else setglobalvar TermineLettre to false;

                return true;
        }
}

Actor Peintre
```

```
{
        VAR TermineLettre;

        TypicalSituation ChoisirLettre
        ALIAS ChoisirCouleur
        FOCUS Peintre did last ChoisirForme {Peintre at Abstract|Operating task , Traceur at Operating
task}
        EVALUATOR
        {
                VAR tache;
                tache=lookact for {"Dessine"};
                if ((tache == nil) && (lookact != nil)) then return false;
                if ((tache != nil) && (lookact for {"Colorie"} == nil)) then return false;

                if (tache != nil) then {setglobalvar TermineLettre to true;}
                else setglobalvar TermineLettre to false;

                return true;
        }
}

// ************************************************************************
// Taches du Traceur
// ************************************************************************


Abstract ChoisirForme()
{
        VAR forme;

        if (lookbb "NbLettres" >= 20) then calltask "EnregisterSession" with {true};
        else
        {
                forme=calltask "ExpertiseForme";
                if (forme==nil) then calltask "EnregistreSession" with {false};
                else
                {
                        calltask "Dessine" with {forme};
                        putbb (lookbb "NbLettres" + 1) as "NbLettres";
                        if (getglobalvar TermineLettre) then calltask "RecordLetter";
                }
        }
}

Control ExpertiseForme()
{
        VAR tache,couleur,forme,hasard,couleur2,forme2,choixCarre,choixCercle;

        hasard= calltask "Hasard";

        tache = lookact for {"Colorie"};
        if (tache == nil) then  //Forme de la premiere lettre
        {
                if (hasard==1) then  return "Carre";
```

```
            else return "Cercle";
        }

        couleur= ((getelement 1 of (gettracetaskparam tache)));
        forme = ((getelement 1 of (gettracetaskparam (lookact for {"Dessine"}))));

        tache = lookact for {"ChoisirForme"};
        couleur2=nil;
        forme2=nil;
        if (tache != nil) then
        {
            VAR t2;
            t2=lookact for {"Colorie"} jumpto (gettracetime tache);
            if (t2 != nil) then
            {
                couleur2=((getelement 1 of (gettracetaskparam tache)));
                t2=lookact for {"Dessine"} jumpto (gettracetime tache);
                forme2=((getelement 1 of (gettracetaskparam tache)));
            }
        }

        choixCarre=false;
        choixCercle=false;

        if (couleur == "Bleu") then  choixCercle=true;
        if (forme=="Cercle" && forme2=="Cercle") then  choixCarre=true;

        if (choixCercle && choixCarre) then  return nil;
        if (choixCercle) then  return "Cercle";
        if (choixCarre) then  return "Carre";

        // Inserer les heuristiques ici


        // Fin d'insertion d'heuristiques

        if (hasard==1) then  return "Carre"; //pas de heuristique choisi
        else return "Cercle";

}

Operating Dessine(forme)
{
}

// ************************************************************************
// Taches du Peintre
// ************************************************************************

Abstract ChoisirCouleur()
{
        VAR couleur;
        couleur=calltask "ExpertiseCouleur";
        if (couleur==nil) then  calltask "EnregistreSession" with {false};
```

```
        else
        {
                calltask "Colorie" with {couleur};
                if (getglobalvar TermineLettre) then calltask "RecordLetter";
        }
}


Control ExpertiseCouleur()
{
        VAR tache,couleur,forme,hasard,couleur2,forme2,choixBleu,choixRouge;

        hasard= calltask "Hasard";

        tache = lookact for {"Dessine"};
        if (tache == nil) then  //Couleur de la premiere lettre
        {
                if (hasard==1) then  return "Carre";
                else return "Cercle";
        }


        forme= ((getelement 1 of (gettracetaskparam tache)));
        couleur = ((getelement 1 of (gettracetaskparam (lookact for {"Colorie"}))));


        tache = lookact for {"ChoisirCouleur"};
        couleur2=nil;
        forme2=nil;
        if (tache != nil) then
        {
                VAR t2;
                t2=lookact for {"Colorie"} jumpto (gettracetime tache);
                if (t2 != nil) then
                {
                        couleur2=((getelement 1 of (gettracetaskparam tache)));
                        t2=lookact for {"Dessine"} jumpto (gettracetime tache);
                        forme2=((getelement 1 of (gettracetaskparam tache)));
                }
        }

        choixBleu=false;
        choixRouge=false;

        if (forme == "Carre") then  choixRouge=true;
        if (couleur=="Rouge" && couleur2=="Rouge") then  choixBleu=true;

        if (choixRouge && choixBleu) then  return nil;
        if (choixRouge) then  return "Rouge";
        if (choixBleu) then  return "Bleu";

        // Inserer les heuristiques ici
```

```
        // Fin d'insertion d'heuristiques

        if (hasard==1) then  return "Rouge"; //pas d'heuristique choisi
        else return "Bleu";
}

Operating Colorie(couleur)
{
}


// ********************************************************************
// Taches partagees
// ********************************************************************


Cognitive RecordLetter()
{
        VAR time,tskcol1,tskdess1,tskcol2,tskdess2,idbbcol1,col1,col2,dess1,dess2,idbb;

        tskcol1=lookact for {"Colorie"};
        tskdess1=lookact for {"Dessine"};

        time=gettracetime tskcol1;
        if (time > gettracetime tskdess1) then time=gettracetime tskdess1;

        tskcol2=lookact for {"Colorie"} jumpto time;
        if (tskcol2 == nil) then return; //premiere lettre

        tskdess2=lookact for {"Dessine"} jumpto time;

        dess1= getelement 1 of (gettracetaskparam tskdess1);
        col1=  getelement 1 of (gettracetaskparam tskcol1);

        dess2= getelement 1 of (gettracetaskparam tskdess2);
        col2= getelement 1 of (gettracetaskparam tskcol2);

        idbb = dess1+col1+dess2+col2;
        putbb ((lookbb idbb) +1) as idbb;

        idbb = dess1+dess2+col2;
        putbb ((lookbb idbb) +1) as idbb;

        idbb = col1+dess2+col2;
        putbb ((lookbb idbb) +1) as idbb;

        idbb = dess1+dess2;
        putbb ((lookbb idbb) +1) as idbb;

        idbb = col1+dess2;
        putbb ((lookbb idbb) +1) as idbb;

        idbb = dess1+col2;
        putbb ((lookbb idbb) +1) as idbb;
```

```
        idbb = col1+col2;
        putbb ((lookbb idbb) +1) as idbb;
}

Cognitive EnregistreSession(succes)
{
        calltask "AddExample" with {
        getbb "CercleRougeCercleRouge",getbb "CercleRougeCercleBleu",
        getbb "CercleRougeCarreRouge",getbb "CercleRougeCarreBleu",
        getbb "CarreRougeCercleRouge",getbb "CarreRougeCercleBleu",
        getbb "CarreRougeCarreRouge",getbb "CarreRougeCarreBleu",
        getbb "CercleBleuCercleRouge",getbb "CercleBleuCercleBleu",
        getbb "CercleBleuCarreRouge",getbb "CercleBleuCarreBleu",
        getbb "CarreBleuCercleRouge",getbb "CarreBleuCercleBleu",
        getbb "CarreBleuCarreRouge",getbb "CarreBleuCarreBleu",

        getbb "CercleCercleRouge",getbb "CercleCercleBleu",
        getbb "CercleCarreRouge",getbb "CercleCarreBleu",
        getbb "CarreCercleRouge",getbb "CarreCercleBleu",
        getbb "CarreCarreRouge",getbb "CarreCarreBleu",
        getbb "BleuCercleRouge",getbb "BleuCercleBleu",
        getbb "BleuCarreRouge",getbb "BleuCarreBleu",
        getbb "RougeCercleRouge",getbb "RougeCercleBleu",
        getbb "RougeCarreRouge",getbb "RougeCarreBleu",

        getbb "CercleRouge",getbb "CercleBleu",
        getbb "CarreRouge",getbb "CarreBleu",
        getbb "CarreCercle",getbb "CarreCarre",
        getbb "CercleCarre",getbb "CercleCercle",
        getbb "BleuRouge",getbb "BleuBleu",
        getbb "RougeRouge",getbb "RougeBleu",
        getbb "RougeCercle",getbb "RougeCarre",
        getbb "BleuCercle",getbb "BleuCarre",
        succes
        }
}
```

Annexe D Un premier prototype fabriqué conjointement avec NOVASYS

Cet exemple de code est l'un des premiers protoypes fabriqués pour des fins commerciales. On y retrouve un exemple de protocole de communication par structures avec la ressource. Ce prototype permet de naviguer à travers plusieurs questions et donne une évaluation à la fin de l'exercice.

```
// ************************************************************************
// La strategie
// ************************************************************************

Strategy Navigator
{
 SOCIETY { Navigator begin Init}
}

// ************************************************************************
// Les objets
// ************************************************************************

Type PageLoaded
{
        concept,
        type,
        time
}

Type Diagnosis
{
        name,
        correct
}

Type LoadPage
{
        url
}

Type GiveHint
{
        name
}

Type GiveCongratulation
```

```
{
        name
}

Type GiveCorrection
{
        name
}

Type GiveAnswers
{
        name
}

Type PageFinished
{}


Type ResourceFinished
{}

// ************************************************************************
// L'acteur
// ************************************************************************

Actor Navigator
{
        VAR LastReussi, Total, LastRes, LastPageValue, NbQuests, Sommation,
                PageScore;

        TypicalSituation ReactToNewPage
        ALIAS PageLoaded
        EVALUATOR
        {
                return (getevent istype PageLoaded);
        }

        TypicalSituation ReactToDiagnostic
        ALIAS ReadDiagnosis
        EVALUATOR
        {
                return (getevent istype Diagnosis);
        }

        TypicalSituation ReactToPageFinished
        ALIAS LoadNextPage
        EVALUATOR
        {
                return (getevent istype PageFinished);
        }

        TypicalSituation ReactToResourceFinished
        ALIAS ResourceOver
        EVALUATOR
```

```
        {
                return (getevent istype ResourceFinished);
        }
}

// ************************************************************************
// Tâches
// ************************************************************************

Operating LoadNextPage ()
{
        VAR res,val,con;
        res = getevent;
        val=getglobvar LastReussi;
        con=(getglobvar LastConcept);

        while (getbb category "PageConcept") do {}
        while (existbb category "Diags") do {getbb category "Diags";}

        if (! existbb con)  then
                setglobvar Total to (getglobvar Total + getglobvar LastPageValue);

        if ( (val) && (val > lookbb con)) then
                putbb val as con;
}

Operating ResourceOver()
{
        VAR text,temp,Score;
        Score=0;
        temp=getbb;
        while (temp>0) do
        {
        Score = Score+temp;
                temp=getbb;
        }

        text = "You got " +  Score;
        text = text + " on ";
        text = text + (getglobvar Total);
        say "You got " +  Score + " on " + (getglobvar Total);
}

Operating PageLoaded ( )
{
        VAR res;
        res = getevent;
        if (res.type == "Acquisition") then
        {
                setglobvar LastPageValue to 1;
                setglobvar LastReussi to 1;
        }
        else
        {
```

```
                setglobvar PageScore to 0;
                setglobvar LastReussi to 0;
                if (res.type == "Evaluation") then
                {
                        setglobvar LastPageValue to 1; // 50% et +
                        setglobvar NbQuests to res.time;
                        setglobvar Sommation to false;
                }
                else if (res.type == "Sommation") then
                {
                        setglobvar LastPageValue to 2; //75% et + 50% et + =1
                        setglobvar NbQuests to res.time;
                        setglobvar Sommation to true;
                }
        }

        setexpression "paisible";
        say "This page is about "+ res.concept +" please read it carefully";
}

Operating ReadDiagnosis()
{
        VAR d,str,err;
        d= getevent;

        if (getglobvar Sommation) then
        {
                if ((d.correct) && (! existbb d.name category "PageConcept")) then
                {
                        putbb 1 as d.name category "PageConcept";
                        setglobvar PageScore to (getglobvar PageScore + 1);
                }

                if (4*(getglobvar PageScore) >= 3*(getglobvar NbQuests)) then
                        setglobvar LastReussi to 2;
                else if (2*(getglobvar PageScore) >= (getglobvar NbQuests)) then
                        setglobvar LastReussi to 1;
                return;
        }

        if (d.correct) then
        {
                putbb 0 as d.name category "Diags";
                if (! existbb d.name) then
                {
                        putbb 1 as d.name;
                        setglobvar PageScore to (getglobvar PageScore + 1);
                }

                if ((getglobvar PageScore)*2 >= (getglobvar NbQuests)) then
                        setglobvar LastReussi to 1;

                setexpression "excite";
                str = make GiveCongratulation;
```

```
                str.name = d.name;
                send str;
        }
        else
        {
                err=0;
                if (existbb d.name category "Diags") then
                        err= lookbb d.name category "Diags";

                if (err == 0) then
                {
                        str = make GiveHint;
                        str.name = d.name;
                        setexpression "surprise";
                        putbb 1 as d.name category "Diags";
                        send str;
                }
                else
                {
                        if (err == 1) then
                        {
                                str = make GiveCorrection;
                                str.name = d.name;
                                setexpression "frustre";
                                send str;
                                putbb 2 as d.name category "Diags";
                        }

                        if (err == 2) then
                        {
                                str = make GiveAnswers;
                                str.name = d.name;
                                setexpression "paisible";
                                send str;
                        }
                }
        }
}


// ***************************************************************************

Operating Init ( )
{
        VAR message;
        setglobvar Total to 0;
        setglobvar LastReussi to 0;
        setglobvar LastRes to nil;
        message = make LoadPage;
        message.url = getcurrentresource;
        send message;
}
```