Dessalles, J-L. & Meyers, P. (1993). "Exemple d'une simulation argumentée pour l'apprentissage de Prolog". In M. Baron, R. Gras & J-F. Nicaud (Eds), *Environnements interactifs d'apprentissage avec ordinateur*. Paris : Eyrolles, 147-157.

Exemple d'une simulation argumentée pour l'apprentissage de Prolog

Jean-Louis Dessalles
TELECOM-Paris - 46 rue Barrault - 75634 Paris Cedex 13
Tel.: (1) 45 81 75 29 - Fax: (1) 45 81 31 19 - E-✉: dessalles@enst.fr

Pascal Meyers
Université Catholique de Leuven (KUL)
Département d'Informatique

RESUME : L'étudiant qui cherche à acquérir un savoir-faire, ici la maîtrise de Prolog, a aussi besoin de connaissances conceptuelles. Pour répondre à ce type de besoin, nous avons développé un système qui permet à l'étudiant de simuler l'exécution de son programme Prolog, mais qui lui offre aussi la possibilité de soumettre ce programme au regard critique de SAVANT3. Ce dernier système a été conçu pour soutenir une argumentation avec l'étudiant. Il est utilisé ici pour critiquer la justesse et l'efficacité du programme écrit par l'étudiant, ce qui permet à celui-ci de corriger d'éventuelles fautes conceptuelles. L'étudiant peut ainsi faire tourner son programme et observer son exécution, pour ensuite "discuter" de ce qu'il a écrit avec SAVANT3. Nous abordons la question de savoir s'il est possible et souhaitable d'étendre ce qui n'est pour l'instant qu'une maquette à des situations réelles (par ex. programme Prolog complexe) et à des sujets quelconques (économie, architecture de réseau, etc.).

1. Le rôle de la critique dans l'apprentissage de savoir-faire

Contrairement à la plupart des produits d'assistance à l'enseignement, qui visent à faciliter l'acquisition de savoir-faire, le développement de SAVANT3 a été motivé au départ par le souci d'enseigner des concepts, (cf. [dessalles 1990]). Dans certaines situations, en effet, l'étudiant est supposé acquérir des connaissances de nature conceptuelle (par exemple en biologie moléculaire ou en histoire). En observant les connaissances échangées dans les conversations naturelles [dessalles 1992b], nous avons pensé que l'argumentation était un bon moyen d'enseigner des notions nouvelles. SAVANT3 est ainsi conçu pour argumenter avec l'étudiant en utilisant une stratégie simple : le système essaie de déceler une contradiction logique dans le discours de l'étudiant, et ce-dernier doit donc faire des déclarations qui soient cohérentes du point de vue du système, au besoin en révisant certaines d'entre elles [dessalles 1992a].

SAVANT3 a été appliqué à des contenus comme la théorie des systèmes, les communications analogiques et numériques. Dans chaque cas, il s'agissait d'enseigner des connaissances de nature conceptuelle : notion de filtre, transformées de Fourier et de Laplace, interférence entre symboles, etc. Bien que la majeure partie de l'enseignement à TELECOM-Paris soit consacrée à l'apprentissage de notions, nous nous sommes posé la question de savoir si SAVANT3 pouvait être utile lors de l'acquisition d'un savoir-faire comme la programmation. Certes, nous faisons une distinction qualitative très nette entre *concepts* et *savoir-faire* [Dessalles 1990], [Inhelder & Piaget 1979], [Ohlsson 1991], [Sleeman 1989]. Mais il apparaît clairement qu'ils sont liés lors de l'accomplissement de tâches inhabituelles.

Nous sommes donc partis de l'idée maintenant bien établie que de nombreuses erreurs lors de l'accomplissement d'une tâche non maîtrisée sont la conséquence de fautes conceptuelles (*misconceptions*) [Stevens et al. 1979], [Sleeman 1982], [Self 1992]. Prenons l'exemple qui a servi de base au développement de notre maquette de simulation argumentée : placer des "cuts'" aux bons endroits d'un programme Prolog constitue indéniablement un savoir-faire. La connaissance conceptuelle associée (un cut inhibe toutes les alternatives rencontrées depuis l'appel de la clause dans l'arbre de résolution) est insuffisante dans la pratique pour que le débutant place des "cuts" là où ils sont utiles. Mais elle est nécessaire, car il est difficile d'imaginer que le programmeur débutant trouve des indices, par exemples syntaxiques, pour placer correctement des cuts dont il ne comprendrait pas l'effet. De plus, des versions erronées de cette connaissance conduisent à des erreurs de placement : par ex. celui qui croit que le cut rend déterministes les buts (goals) qui le suivent placera des cuts trop tôt au sein d'une clause.

Il existe plusieurs systèmes dont l'objectif est d'apporter un diagnostic ou une critique de type conceptuel sur un programme informatique qu'un étudiant vient d'écrire. Par exemple PROUST [Johnson & Soloway 1987], pour l'apprentissage du Pascal, peut introduire le concept de "test sentinelle" dans son analyse du programme de l'élève. Nowé et Jonkers [1991] disent utiliser des "concepts abstraits de programmation" pour formuler des diagnostics de haut niveau (en Lisp ou Pascal). Dans Lisp-Tutor [Anderson et al. 1989] les diagnostics utilisent des concepts comme "variable de comptage", "communication d'une valeur d'initialisation à une fonction". Intelligent-Prolog-Tutor [Lee 1991] signale à l'apprenant des erreurs comme des définitions circulaires. Lisp-Critic [Fischer et al. 1991] est capable de suggérer des transformations d'un programme Lisp en donnant des explications relatives à l'efficacité ou à la lisibilité.

Dans tous ces systèmes qui ont pour objectif d'aider l'étudiant à acquérir un savoir-faire en programmation, on constate que les diagnostics sont généralement conceptuels puisqu'ils font intervenir des notions de programmation (par ex. PATAT [Nowé & Jonkers 1991] peut écrire : "variable *result* must be initialised to *0* because *result* is the running total variable and the unity element for the sum operator is *0*", par opposition à un diagnostic non conceptuel qui pourrait être : "cette clause est erronée, voici la forme correcte"). Autrement dit l'effet pédagogique de tels diagnostics nécessite un examen par l'étudiant de ses propres connaissances.

Notre approche consiste elle aussi à apporter une critique de type conceptuel à l'étudiant, mais elle diffère sur un point qui nous semble important. Dans son principe, la simulation argumentée que nous présentons ici ne consiste pas à fournir

un simple diagnostic. Il s'agit, à partir de ce que SAVANT3 considère comme une anomalie de structure dans le programme de l'étudiant, de susciter un *échange argumentatif* avec ce-dernier. On rejoint ainsi l'idée de base de SAVANT3 qui utilise une "conversation" pour tester la cohérence des connaissances de l'étudiant, et non une simple succession d'échanges question-feedback. Cette approche semble particulièrement indiquée dans le cas de l'apprentissage d'un langage de programmation, et ce pour plusieurs raisons :

- La possibilité de simuler le programme de l'étudiant permet à celui-ci de constater des exécutions anormales, ce qui est censé le motiver pour solliciter la critique de SAVANT3. On peut imaginer que SAVANT3 intervienne automatiquement en cas d'anomalie (critique active), mais nous n'avons pas implanté cette possibilité.
- Les critiques que l'on peut formuler sur un programme donné sont rarement définitives. L'étudiant peut avoir une motivation, non connue du programme, pour écrire par exemple une clause correcte mais très inefficace. L'échange d'arguments lui permet d'imposer son point de vue.
- Le diagnostic immédiat permet rarement de mettre la faute conceptuelle en évidence. C'est souvent lorsque l'élève se défend en donnant ses raisons que le dialogue remonte jusqu'aux présupposés erronés.

2. L'apprentissage d'un savoir-faire en Prolog avec un exercice simple

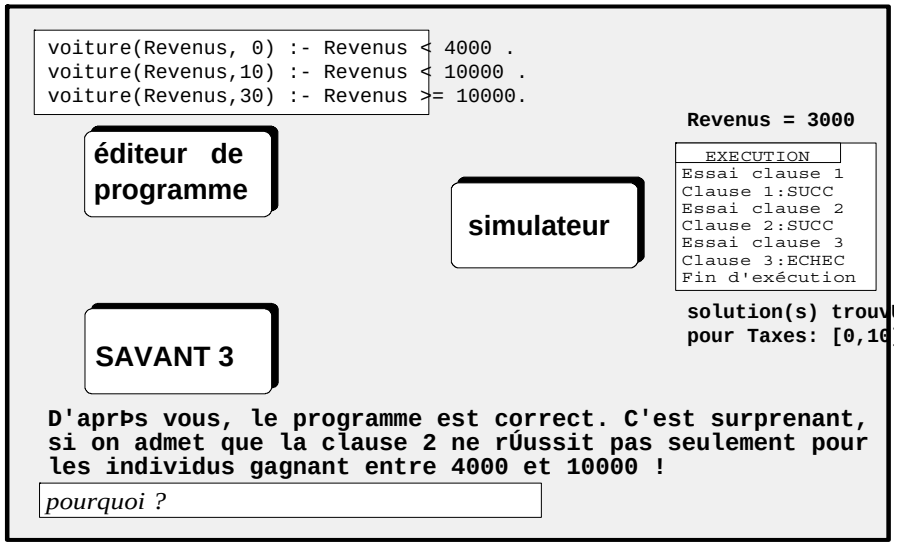
Dans la maquette que nous avons mise au point, on demande à l'étudiant d'écrire un petit programme Prolog qui calcule la relation *voiture(**X**,**Y**)* suivante :

Pour **X** inférieur à 4000, **Y** doit être nul.
Pour **X** entre 4000 et 10000, **Y** doit être égal à 10.
Pour **X** supérieur à 10000, **Y** doit valoir 30.

Ce programme est supposé calculer la taxe (**Y**) qu'un individu doit payer pour sa voiture d'après le montant de ses revenus (**X**). Mais nous ne laissons pas à l'étudiant la possibilité d'écrire un programme quelconque. Nous lui fournissons un éditeur très limité qui impose les contraintes suivantes :

- le programme comprend trois clauses
- la tête de chaque clause est imposée : *voiture(**Revenus**, 0) :-*
  *voiture(**Revenus**, 10) :-*
  *voiture(**Revenus**, 30) :-*
- l'étudiant peut placer des buts pour construire ces clauses en les choisissant dans une liste :
  < 4000
  >= 4000
  < 10000
  >= 10000
  ! (cut)

Le nombre de programmes qu'il pourra écrire avec un tel éditeur est tout de même égal à 2^{15} (32768) (chaque but est présent ou absent dans chacune des trois clauses). A chaque instant l'étudiant peut exécuter son programme : il fournit la valeur de ***Revenus*** dans *voiture(**Revenus**, **Taxes**)*, puis l'interpréteur Prolog cherche toutes les solutions possibles pour ***Taxes*** pendant qu'une fenêtre de trace montre les clauses appelées et leur échec éventuel (voir figure).



Cet exercice est utilisé pour enseigner à l'étudiant certains savoir-faire nécessaires pour l'écriture et la compréhension de programmes Prolog. L'objectif est d'attirer son attention sur la justesse et l'efficacité de son programme. Par exemple, un programme peut être correct, mais très peu efficace :

voiture(Revenus, 0) :- *Revenus < 4000, Revenus < 10000 .*
voiture(Revenus, 10) :- *Revenus >= 4000, Revenus < 10000 .*
voiture(Revenus, 30) :- *Revenus >= 4000, Revenus >= 10000 .*

Ce programme est inefficace pour plusieurs raisons : il comporte des buts redondants dans les clauses 1 et 3, et l'aspect déterministe du programme n'est pas pris en considération (par ex. un cut peut être introduit en clause 1 pour éviter un appel inutile des clauses suivantes).

L'interprète Prolog ne peut pas mettre clairement en évidence le manque d'efficacité du programme. Il peut simplement montrer que le programme a un comportement anormal, mais il ne peut pas relier ceci à une faute conceptuelle éventuelle sur le fait, par exemple, qu'il y a backtracking et que le programme peut donner plusieurs solutions, ou encore sur la signification des cuts, etc. C'est en cela que l'intervention de SAVANT3 peut être utile.

3. Le regard critique de SAVANT3

Pour rendre SAVANT3 capable d'argumenter sur le programme écrit par l'étudiant, nous avons dû lui fournir une connaissance du problème. Celle-ci consiste en une centaine de règles comme les deux suivantes :

> *règle 27 (1)* :
>
> la clause 2 ne réussit pas pour les individus gagnant moins de 4000
> la clause 2 est appelée pour les individus gagnant moins de 4000
> un argument inférieur à 4000 satisfait tous les buts de la clause 2

> *règle 90 (0)* :
>
> Revenus >= 10000 est utilisé en clause 3
> le programme est le plus efficace possible
> la clause 3 n'est pas appelée pour les individus gagnant moins de 4000
> la clause 3 n'est pas appelée pour les individus gagnant entre 4000 et 10000

Ces règles sont présentées (et stockées) sous forme symétrique, comme des ensembles de propositions incompatibles (clauses paradoxales). On peut lire la règle 27 par exemple ainsi : si la clause 2 est appelée pour les individus gagnant moins de 4000, et qu'un argument inférieur à 4000 satisfait tous ses buts, alors il serait paradoxal qu'elle ne réussît pas pour les individus gagnant moins de 4000. De telles règles sont utilisées par SAVANT3 soit pour démontrer des faits (lorsque toutes les propositions d'une règle sont vraies sauf une qui est indéterminée, cette dernière est nécessairement fausse), soit pour s'étonner (lorsqu'une règle est saturée, c.à.d. que toutes ses propositions sont vraies). Le nombre entre parenthèses distingue les règles de bon sens (1) des règles *a priori* moins évidentes pour l'étudiant (0).

A tout moment l'étudiant peut soumettre son programme à SAVANT3. Il doit d'abord préciser s'il considère son programme comme correct et efficace, puis SAVANT3 s'efforce de saturer une de ses règles. Dans l'extrait 1, une règle se trouve effectivement saturée, ce que le système exprime sous forme d'étonnement :

extrait 1

```
voiture(Revenus, 0) :-   Revenus < 4000 .
voiture(Revenus, 10) :-  Revenus < 10000 .
voiture(Revenus, 30) .
```

S3- *D'après vous, le programme est correct. C'est surprenant si l'on admet que la clause 2 ne réussit pas que pour les individus gagnant entre 4000 et 10000 !*
Et.- [la clause 2 ne réussit que pour les individus gagnant entre 4000 et 10000]
S3- *D'après vous, la clause 2 ne réussit que pour les individus gagnant entre 4000 et 10000. C'est surprenant si l'on admet que la clause 2 réussit pour les individus gagnant moins de 4000 !*
Et.- [la clause 2 ne réussit pas pour les individus gagnant moins de 4000]
S3- *D'après vous, la clause 2 ne réussit pas pour les individus gagnant moins de 4000. C'est surprenant si l'on admet que la clause 2 est appelée pour les individus gagnant moins de 4000 !*

Et.- je mets un cut dans la clause 1
S3- Ok...

La conversation s'arrête dès que l'étudiant apporte une modification à son programme. Cet extrait montre que SAVANT3 tire des conclusions de longues chaînes d'inférence (parfois après 10 déductions). Lorsque SAVANT3 est incapable de déceler une incohérence, il suggère qu'une proposition inconnue de lui est peut-être vraie, comme dans l'extrait 2 où l'étudiant n'a pas précisé que son programme était efficace.

extrait 2

```
voiture(Revenus,0) :-Revenus < 4000, !.
voiture(Revenus,10):-Revenus >= 4000,
                     Revenus < 10000, !.
voiture(Revenus,30).
```

S3- D'après ce que j'ai compris, la clause 2 contient [Revenus >= 4000]. Mais peut-être pensez-vous que le programme est le plus efficace possible ?
Et.- Pourquoi ?
S3- Il y a tout de même un problème... sauf si vous considérez que la clause 2 est appelée pour les individus gagnant moins de 4000
Et.- oui
S3- D'après vous, la clause 2 est appelée pour les individus gagnant moins de 4000. C'est surprenant si l'on admet que la clause 1 réussit pour les individus gagnant moins de 4000 !
Et.- pourquoi est-ce surprenant ?
S3- Il y a tout de même un problème... D'après ce que j'ai compris :
- *la clause 2 est appelée pour les individus gagnant moins de 4000*
- *la clause 1 réussit pour les individus gagnant moins de 4000*
- *il y a un cut dans la clause 1*

Que proposez-vous comme solution ?
Et.- [la clause 2 n'est pas appelée pour les individus gagnant moins de 4000]
S3- *Mais attendez... D'après ce que vous avez dit :*
- *la clause 2 contient [Revenus >= 4000]*
- *la clause 2 n'est pas appelée pour les individus gagnant moins de 4000*

Je m'avoue surpris !
Et.- j'enlève [Revenus >= 4000] de la clause 2
S3- Ah bon!... Ok...

Le fonctionnement de SAVANT3 est simple dans son principe (cf. [dessalles 1992a]). Le système cherche à saturer l'une des règles de sa base de connaissances. S'il y parvient d'emblée, comme dans l'extrait 1 où la règle 27 donnée plus haut se trouve saturée, c'est à l'étudiant de sortir de la situation incohérente décelée par SAVANT3. Lorsque aucune règle n'est saturée, le programme amène l'étudiant à spécifier davantage la situation, comme au début de l'extrait 2, dans le but de saturer la règle dont la plus grande proportion de propositions est vérifiée.

Les messages donnés par SAVANT3 mêlent des expressions fixes et des libellés de propositions. Les répliques de l'étudiant sont reconnues par détection de mots clés et d'une négation éventuelle, sauf lorsque l'étudiant préfère choisir dans un menu fermé (répliques entre crochets dans les extraits).

4. Les difficultés techniques rencontrées

L'un des intérêts techniques de SAVANT3 est d'être transposable. Pour lui permettre d'argumenter sur un sujet quelconque, l'auteur se contente de donner au système un ensemble statique de règles (typiquement de 15 à 20 règles pour une conversation d'une dizaine de répliques) en utilisant un système auteur spécifique. L'auteur n'a pas à se préoccuper de la gestion du dialogue et il n'a pas à modifier SAVANT3 [dessalles 1992a]. La situation est malheureusement moins claire, pour l'instant , dans le cas de la simulation argumentée.

Un premier problème vient de la réalisation de la base de règles. Ces règles sont issues de la structure du programme que l'étudiant peut écrire avec l'éditeur limité qui lui est offert, ainsi que de la sémantique du problème posé. Pour le problème de la voiture taxée, nous avons dû écrire une centaine de règles "à la main", de manière à rendre compte sous forme logique du fonctionnement du programme Prolog. Une telle approche n'est pas envisageable pour un programme Prolog quelconque.

Nous avons aussi été confrontés à un problème de temps de réponse. SAVANT3 résout à maintes reprises un problème de satisfiabilité d'un ensemble de règles ; or il s'agit d'un problème exponentiel. Alors que le temps de réponse est instantané pour moins de 20 règles, il peut atteindre plus de 10 secondes avec 100 règles, même après que nous avons opéré un certain nombre d'optimisations dans le moteur d'inférences.

Enfin, l'interfaçage de SAVANT3 avec l'éditeur de programme et avec l'interpréteur Prolog ne va pas de soi. Afin de respecter le principe d'un SAVANT3 indépendant du sujet traité, nous n'avons pas poussé cet interfaçage aussi loin que souhaitable :

situation	éditeur → SAVANT3	SAVANT3 → éditeur	interpréteur → SAVANT3	SAVANT3 → interpréteur
actuelle	Ajout de faits à la base de connaissances pour décrire le programme de l'étudiant. Déclenchement de SAVANT3 par l'étudiant.	Détection d'une modification du programme avec sortie de SAVANT3. Mise à jour automatique du programme d'après les déclarations de l'étudiant.	néant	néant
souhaitable	intervention automatique de SAVANT3 (critique active).	mise à jour simultanée du programme sans sortie systématique de SAVANT3.	détection d'anomalie de fonctionnement.	fabrication d'exemples d'exécutions anormales.

Un certain nombre de solutions sont envisageables. L'idée principale pour résoudre le problème de la constitution de la base de connaissances et du temps de réponse consiste à imaginer une synthèse d'un nombre limité de règles pertinentes en temps réel (c.à.d. en fonction de ce que l'étudiant écrit) à partir de patrons généraux de règles, plutôt que de prévoir d'emblée un ensemble statique complet de règles spécifiques. C'est l'approche utilisée dans les autres systèmes d'aide à la rédaction de programmes. Elle reste toutefois dépendante du langage informatique enseigné et de l'exercice posé.

En ce qui concerne l'interfaçage de SAVANT3, l'écueil à éviter est l'écriture d'une interface *ad hoc*. Il nous reste à définir le principe d'une interface générale entre SAVANT3 et un programme de simulation quelconque.

5. Discussion : intérêt et difficultés de la simulation argumentée

Deux causes d'erreurs sont généralement invoquées pour expliquer les erreurs commises par les débutants lors d'un apprentissage : les bogues procédurales (*bugs & mal-rules*) et les fautes conceptuelles (*misconceptions*). Les premières permettent parfois d'expliquer et de prédire des erreurs lors de l'apprentissage de procédures. La procédure à acquérir peut être modélisée par un graphe d'actions élémentaires [VanLehn 1981, 1988] ou par un ensemble de règles de production [Sleeman 1982], [Payne 1990]. Les erreurs sont expliquées par des règles erronées (*mal-rules*) ou des règles correctes improprement appliquées (*bugs*). En revanche, les fautes conceptuelles ont surtout été mentionnées dans des contextes d'apprentissage conceptuel [Stevens et al. 1979].

Après un processus d'apprentissage purement procédural, l'élève a acquis des automatismes (par ex. algorithme de la division), mais il est incapable de justifier ses actions (pourquoi abaisser le chiffre suivant, pourquoi soustraire...) en expliquant pourquoi l'action choisie est correcte et pourquoi une action alternative serait erronée. Il semble que dans un tel contexte, selon [Sleeman 1989][1], donner des explications à propos d'une erreur soit inefficace par rapport à une simple correction procédurale ("ceci est faux, voici l'action correcte"). Un savoir-faire procédural semble déconnecté de toute construction conceptuelle !

Toutefois, dès qu'il s'agit d'acquérir des procédures complexes mettant en jeu une combinatoire importante, si bien qu'il ne suffit pas d'appliquer un nombre restreint de règles *condition* $\rightarrow$ *action*, il semble que le débutant doive comprendre ses erreurs pour progresser. C'est le cas dans l'apprentissage de la programmation. Or comprendre une erreur, c'est reconnaître une faute conceptuelle.

Le diagnostic d'une faute conceptuelle est difficile, voire impossible à partir d'une performance unique [Wenger 1987]. Le pari qui est à la base de notre approche est qu'un tel diagnostic n'est pas nécessaire pour que le système commence à argumenter. Le système détecte une anomalie. Si cette anomalie est la conséquence d'une faute conceptuelle, une argumentation bien menée doit permettre à l'élève de prendre conscience de l'origine de la faute.

Nous avons évoqué plus haut les difficultés techniques de la simulation argumentée. Les difficultés pédagogiques résident essentiellement dans la bonne perception de la pertinence des arguments que la machine oppose à l'étudiant. Nous avons cherché à copier les mécanismes argumentatifs que l'on peut observer dans les conversations naturelles [dessalles 1992b]. Toutefois, il n'est pas certain que l'effet logique souhaité d'un argument soit toujours perçu par l'étudiant. Par exemple, celui-ci doit comprendre les raisons qui poussent la machine à s'étonner (cf. extrait 2). Il s'agit là d'un problème non trivial sur lequel nous travaillons. En revanche,

---

[1] D. Sleeman a mené son étude sur des élèves apprenant l'algèbre élémentaire.

l'amélioration de la forme des arguments (pour éviter la monotonie de l'extrait 1) ne semble pas constituer une difficulté majeure.

6. Conclusions

Le programme présenté ici est une maquette, et il serait prématuré de tenter d'en faire un produit opérationnel. Nous devons encore résoudre les problèmes fondamentaux que nous venons d'évoquer en essayant de coupler SAVANT3 à d'autres simulations ou à des systèmes qualitatifs. Cependant nous restons convaincus de l'efficacité pédagogique potentielle d'un tel système dans le créneau qui est le sien, celui de savoir-faire complexes. Lorsque nous serons parvenus à résoudre toutes les difficultés de principe, nous pourrons envisager de développer un système complet qui pourra intégrer, si besoin, un modèle de l'étudiant. Mais notre système ne saurait être considéré comme un "tuteur".

La simulation argumentée est bâtie sur un principe qui rejoint plutôt celui des systèmes critiques [Fischer et al. 1991]. L'intervention du système est conditionnée par l'existence d'une situation anormale ou problématique. Nous avons pu observer que les interlocuteurs agissent ainsi au cours des conversations naturelles [dessalles 1992b]. Dans un contexte d'enseignement, nous pensons que l'élève sera motivé pour argumenter si le système "l'accuse" d'auto-contradiction [dessalles 1990] [Rätz 1992].

Si l'on conçoit la simulation argumentée comme un système critique et non comme un tuteur, alors on peut envisager les extensions les plus diverses, car il n'est plus nécessaire d'avoir un modèle complet de la tâche à résoudre et de sa solution. Le système émet les critiques qu'il est capable de formuler compte tenu de son champ de compétence, sans prétendre à l'omniscience. Nous avons pu ainsi envisager le couplage de SAVANT3 avec une simulation économique et une simulation de mise en place de réseaux de télécommunications.

L'intérêt spécifique de la simulation argumentée est de permettre l'argumentation des critiques. Cette argumentation peut permettre de remonter à une faute conceptuelle de l'élève, mais elle peut aussi conduire à une gestion d'exceptions : le système accepte un argument de l'utilisateur qui présente la situation présente comme faisant exception à la critique. Le système peut ainsi apprendre, tenir à jour un modèle de l'utilisateur, ou tout simplement reconnaître que l'utilisateur a raison. Nous travaillons à de tels développements.

L'objectif que nous poursuivons est double : l'efficacité pédagogique, et l'acceptabilité. Nous avons tenté de montrer que l'approche qui consiste à s'inspirer des interactions argumentatives naturelles pourrait être prometteuse.

Références

Anderson John R., Conrad Frederick G, Corbett A.T. [1989]. Skill Acquisition and the LISP Tutor. *Cognitive Science* 13, 1989, pp. 467-506

Dessalles jean louis [1990]. Computer Assisted Concept Learning. In Norrie D.H., Six H.-W., *Lecture Notes in Computer Science 438 - Computer Assisted Learning*, Springer-Verlag, Berlin 1990, pp. 175-183

Dessalles jean louis [1992a]. SAVANT3 : un système d'EIAO fondé sur l'explication conversationnelle. *Actes des 2èmes journées Explication du PRC-GDR-IA* (juin 1992), INRIA, Sophia-Antipolis 1992

Dessalles jean louis [1992b]. Les contraintes logiques des conversations spontanées. Rapport technique TELECOM-Paris 92-D-011, Paris 1992

Fischer Gerhard, Lemke Andreas C., et al. [1991]. The Role of Critiquing in Cooperative Problem Solving. *ACM Transactions on Information Systems*, Vol.9, n°3, 1991, pp. 123-151

Inhelder Bärbel, Piaget Jean [1979]. Procédures et structures. *Archives de psychologie*, XLVII, 181, 1979, pp. 165-176

Johnson W. Lewis, Soloway Elliot [1987]. PROUST : An Automatic Debugger for Pascal Programs. In Kearsley Greg P., *Artificial Intelligence & Instruction - Applications and Methods*, Addison-Wesley Publishing Company, Menlo Park, USA 1987, pp. 49-67

Lee M.C. [1990]. Designing an Intelligent Prolog Tutor. In Norrie D.H., Six H.-W., *Lecture Notes in Computer Science 438 - Computer Assisted Learning*, Springer-Verlag, Berlin 1990, pp. 420-431

Nowé Ann, Jonckers V. [1991]. The Use of Very High Level Clichés in PATAT : a Program Analysis Tool Using Algorithm Transformation. In Forte Eddy N., *Proceedings of Calisce'91*, Presses Polytechniques et Universitaires Romandes, Lausanne 1991, pp. 209-216

Ohlsson Stellan [1991]. Interview, by J.Sandberg and Y.Barbard. AICOM vol 4, n° 4, 1991, pp. 137-144

Payne Stephen J., Squibb Helen R. [1990]. Algebra Mal-Rules and Cognitive Accounts of Error. *Cognitive Science* 14, 1990, pp. 445-481

Rätz Thomas, Lusti M. [1992]. Explanation Strategies : realization in a tutor for database normalization. In Brezillon Patrick, *Proceedings of the ECAI-92 Workshop on Improving the Use of KBS with explanations*, Rapp. LAFORIA 92/21 Univ. Paris VI, Paris 1992, pp. 47-56

Self John [1992]. Cognitive Diagnosis for Tutoring Systems. In Neuman Bernd, *Proceedings of ECAI92*, John Wiley, Vienne 1992, pp. 699-703

Sleeman Derek [1982]. Assessing aspects of competence in basic algebra. In Sleeman Derek, Brown J.S., *Intelligent Tutoring Systems*, Academic Press, Londres 1982, pp. 185-199

Sleeman Derek, Kelly A.E., Martinak R. [1989]. Studies of Diagnosis and Remediation with High School Algebra Students. R.D. Ward, J.L. Moore - *Cognitive Science* 13, 1989, pp. 551-568

Stevens Albert, Collins Allan, Goldin Sarah E. [1979]. Misconceptions in student's understanding. *Int. J. Man-Machine Studies* 11, 1979, pp. 145-156

VanLehn Kurt [1981]. *On the Representation of Procedures in Repair Theory*. CIS-16 (SSL-81-7), Xerox Palo Alto Research Center, 1981

VanLehn Kurt [1988]. Toward a Theory of Impasse-Driven Learning. In Mandl Heinz, Lesgold Alan, *Learning Issues for Intelligent Tutoring Systems*, Springer Verlag, Berlin 1988, pp. 19-41

Wenger Etienne [1987]. *Artificial Intelligence and Tutoring Systems*. Morgan Kaufmann Publishers, INC., Los Altos, Cal.,USA 1987, pp. 153-184