Modélisation d'un outil graphique d'aide à la compréhension de résolution de programmes logiques

Mémoire

Mohamed Bouanane

Maîtrise en Informatique

Maître ès sciences (M.Sc.)

Québec, Canada

I. Résumé

Ce projet de recherche traite du domaine de l'intelligence artificielle, plus précisément de la programmation logique, un type de programmation utilisée pour concevoir des systèmes dits intelligents. Ce type de programmation est toutefois assez complexe à assimiler et il n'existe, à notre connaissance, aucun outil interactif qui permette de montrer efficacement le processus d'exécution d'un programme logique. L'objectif de cette recherche consistait donc à proposer un modèle d'outil facilitant la compréhension de la résolution d'un programme logique. Le modèle proposé permet de représenter graphiquement et dynamiquement la trace de résolution d'un programme logique. Un prototype a été construit pour valider ce modèle avec des programmes écrits en langage Prolog. L'utilisateur peut ainsi suivre les étapes d'exécution à travers l'affichage dynamique d'un arbre de dérivation. Actuellement, le modèle ne permet pas de prendre en compte des programmes écrits avec différents langages de programmation, une telle généralisation serait une bonne amélioration.

II. Table des matières

III. Table des Figures

À ma chère mère Sana, et

Tous mes amis proches,

Qui ont tout le temps été là pour moi.

Je vous dédie ce mémoire.

IV. Remerciements

Je tiens à remercier en premier Mme Laurence Capus qui m'a accordé sa confiance et m'a offert l'opportunité d'intégrer son équipe de recherche en ingénierie des connaissances, ERICAE, au département d'informatique et de génie logiciel de l'Université Laval, afin d'y effectuer mon projet de maîtrise.

Je remercie de la même façon mes deux encadreurs Mme Capus et Mr BenSta, pour leurs encadrements, leurs précieux conseils et leurs disponibilités pendant toute la durée de mon projet de maitrise. Je remercie tous les membres de mon jury de maîtrise pour leurs précieux conseils en vue d'améliorer mon travail.

Je tiens à exprimer mes sincères remerciements à tous les professeurs qui m'ont enseigné et qui par leurs encouragements m'ont soutenu dans la poursuite de mes études. Enfin, je remercie tous ceux qui, de près ou de loin, ont contribué à la réalisation de ce travail.

V. Avant-propos

Le présent document a été élaboré dans le cadre d'un travail de maîtrise effectué au sein de l'Équipe de Recherche en Ingénierie des ConnAissancEs (ERICAE) au département d'informatique et de génie logiciel de l'Université Laval. [1]

À l'entame de mon cycle de maîtrise, j'ai eu le choix entre de multiples domaines informatiques très intéressants, mais mon attention à tout de suite été portée vers celui de l'intelligence artificielle et l'ingénierie des connaissances. Un domaine où j'ai eu quelques expériences grâce à des cours dans des cycles inférieurs et qui avait toujours attisé ma curiosité grâce aux grandes avancées technologiques qu'il provoquait de plus en plus dans le monde de l'informatique.

Mon projet de maîtrise aura duré deux ans durant lesquelles je suis passé par deux grandes parties. La première a été consacrée à la recherche, l'exploration et la maîtrise en surface de la notion d'intelligence artificielle, et en profondeur de la notion de programmation logique qui permet en partie de manipuler cette intelligence artificielle. Mais cette première partie a surtout constituée une base très solide sur laquelle je me suis appuyé pour la deuxième grande partie de mon projet de maîtrise : **la conception et réalisation d'un outil graphique pour la programmation logique.**

Au bout de mon cursus, je constate que ces deux années m'ont permis d'acquérir une énorme quantité d'informations sur un domaine qui ne cesse de croitre et d'apporter des innovations technologiques considérables dans notre quotidien, des informations sûrement très utiles pour la suite de mon parcours étudiant et professionnel.

Le lecteur trouvera donc dans ce mémoire le résultat de deux années de recherche et de réalisation dans un cursus de maîtrise que je qualifie à la fois de formateur et de très enrichissant aussi bien sur le plan personnel qu'éducatif.

Introduction

Le présent projet de recherche concerne un domaine jeune et en pleine croissance dans le monde de l'informatique, soit l'intelligence artificielle. C'est un domaine qui a de plus en plus de retombées dans notre vie quotidienne grâce aux nombreux apports qu'il offre. Cette branche de l'informatique est maintenant en développement continu et ses applications se retrouvent à peu près partout dans nos sociétés comme par exemple dans le domaine bancaire avec des systèmes experts d'évaluation de risque lié à l'octroi d'un crédit ou encore en médecine ou d'autres systèmes experts d'aide au diagnostic épaule des médecins chaque jour.

Et afin de concevoir de tels programmes en intelligence artificielle, différents outils sont utilisés, mais l'un des principaux outils reste la programmation logique qui permet de se libérer de certaines contraintes de programmation. En effet, les instructions d'exécution ne sont pas à décrire puisqu'elles sont déjà disponibles avec le langage sous forme d'un démonstrateur de théorèmes ou moteur d'inférences. Cette forme de programmation est donc particulièrement adaptée aux besoins de l'intelligence artificielle.

Au cours de nos recherches, nous avons remarqué que toutes les applications, qui utilisent la programmation logique, se contentent d'afficher le résultat de l'algorithme exécuté. À la demande de l'utilisateur, une trace de résolution du programme peut être fournie, mais ces traces sont rédigées différemment dépendamment du langage et ne sont pas toujours évidentes à déchiffrer pour un débutant dans le domaine de la programmation logique. Ceci ne permet pas à un utilisateur de comprendre le raisonnement effectué par le résolveur pour arriver à ce résultat et donc de comprendre la logique de cette programmation. Ceci est d'autant plus important en programmation logique, puisqu'on ne définit pas la méthode de résolution. À partir de ces constations, nous avons posé notre problématique à savoir : comment améliorer la présentation de l'exécution d'un programme logique pour aider les utilisateurs à mieux comprendre comment le résultat a été trouvé ?

Nous avons étudié les projets existants, semblables à ce contexte d'aide à la compréhension de la programmation logique, et nous en avons identifié les forces et les faiblesses. Ces projets proposent généralement des représentations graphiques de la trace d'exécution sous la forme d'arbres de dérivation, plus visuelles qu'une liste d'instructions textuelles. Nous avons retenu la clarté et la simplicité des interfaces graphiques pour en faciliter l'utilisation ainsi que la portabilité des outils par l'emploi d'un langage ouvert. Nous avons noté le manque d'interactions disponibles entre l'utilisateur et les outils proposés ainsi qu'une absence d'explications. Dans ce contexte, notre objectif était donc de proposer un modèle d'outil d'aide à la compréhension de résolution de programmes logiques par une représentation graphique et dynamique des traces d'exécution. Cette représentation présente un arbre de dérivation, la forme la plus représentative de l'exécution d'un programme logique, qui peut être découverte en fonction des besoins de l'utilisateur.

Nous avons construit un prototype qui prend en entrée une trace de l'exécution d'un programme écrit en langage Prolog, un des principaux langages de programmation logique. Cette trace, sous forme textuelle, est ensuite

affichée sous la forme d'un arbre manipulable. L'utilisateur peut découvrir à son rythme les différentes étapes de la résolution de la racine jusqu'au résultat. Les différents éléments relatifs à chaque étape sont affichés en fonction de cette découverte. Ce prototype nous a permis de valider notre modèle d'un point de vue faisabilité. Il resterait à mener une expérimentation avec des utilisateurs pour vérifier si la compréhension de la programmation logique est améliorée, mais ceci dépasse le cadre de ce projet de recherche. Le modèle proposé pourrait également être amélioré s'il était possible d'utiliser automatiquement des traces d'exécution de n'importe quel langage logique.

Ce mémoire est structuré de la façon suivante. Le premier chapitre permet de présenter l'intelligence artificielle ainsi qu'une analyse des différents langages utilisés permettant la programmation logique. Le deuxième chapitre est dédié à l'étude des projets ayant trait à la représentation graphique des traces d'exécution. La problématique et les objectifs de recherche sont également décrits dans ce chapitre. Le troisième chapitre porte sur le modèle proposé pour représenter graphiquement et dynamiquement les traces d'exécution. Dans le quatrième chapitre, nous présenté le prototype construit pour valider le modèle. Nous discutons également les résultats obtenus et proposons des améliorations. Nous terminons ce mémoire par une conclusion.

Chapitre 1 : La programmation logique et ses différents langages

Notre domaine de recherche se situant dans le domaine de l'intelligence artificielle, nous avons consacré ce chapitre à la présentation de ce domaine. Nous avons ensuite présenté une des formes de programmation utilisée en intelligence artificielle qui est la programmation logique. Nous avons fait une étude des différents langages utilisés en spécifiant les avantages et les inconvénients de chacun d'entre eux.

1.1 L'intelligence artificielle

Nous avons essayé d'acquérir le plus d'information possible sur un domaine qui ne cesse de se développer et de s'enrichir de jour en jour. Nous allons présenter dans cette section les trois états temporels de l'intelligence artificielle : le passé, à travers les origines de cette technologie et certains noms connus ayant participé à son apparition, le présent, avec tous les domaines touchés et les apports que nous offre l'intelligence artificielle, et le futur, avec ce que nous attendons comme avancée technologique grâce au développement croissant de cette dernière.

1.1.1 Origine de l'intelligence artificielle

Il faut remonter jusqu'à la fin du 17ème siècle pour avoir le premier aperçu de la notion d'une machine « pensante ». En 1642, Blaise Pascal parvient à construire une machine capable de résoudre des additions et des soustractions au moyen d'engrenages. Un certain Gottfried Leibnitz parvient quelques années plus tard à y ajouter les opérations de multiplications et de divisions, avant de s'attaquer à son plus grand projet connu sous le nom de *Characteristica Universalis*, un système de règles formelles destinées à résoudre tous les problèmes auxquels étaient confrontés les humains.

Un demi-siècle plus tard, La Mettrie écrit *L'Homme-machine*, aboutissement logique, voire inéluctable, de la pensée newtonienne : le comportement physique et mental de l'homme pourrait être entièrement décrit à l'aide de règles déterministes et, par conséquent, l'homme ne serait rien d'autre qu'une machine, d'une complexité extraordinaire, certes, mais une machine tout de même [2].

Quant à la notion d'intelligence artificielle comme nous la connaissons actuellement, il faudra remonter aux années 1940-1950. Une poignée de scientifiques d'une large gamme de domaines (mathématiques, psychologie, ingénierie, économie et science politique) ont commencé à discuter de la possibilité de créer un cerveau artificiel

suite à l'invention de l'ordinateur programmable en 1940, ordinateur basé sur les descriptions d'Alan Turing en 1936.

Une grande frénésie sur la recherche en intelligence artificielle fut provoquée par une conférence tenue sur le campus de Dartmouth Collège pendant l'été 1956. Suite à cette vague de recherche et jusqu'aux années 1980, ce domaine a connu un cycle en dents de scie, où, durant des périodes de gel et dégel, des fonds de recherche s'alternaient pour cause d'absence de résultats ou de nouvelles découvertes.

Au départ, deux approches se confrontent : l'approche logiciste ou symbolique, qui vise à recréer les « lois universelles » de la pensée et s'inspirent du concept de machine de Turing, et l'approche neuronale, incarnée par Frank Rosenblatt, qui essaie d'imiter les processus biologiques cérébraux. Si l'approche logiciste, inspirée des travaux de Russell, Frege, du cercle de Vienne, de logique mathématique, etc., l'emporte à la DARPA, principal organisme finançant les recherches en intelligence artificielle, l'approche neuronale refait surface dans les années 1980, inspirant les travaux sur le connexionnisme. Wikipédia (1).

1.1.2 Définition de l'intelligence artificielle

Lorsqu'on cherche à définir l'intelligence artificielle, nous remarquons tout de suite une grande difficulté dans le monde de l'informatique à unifier une définition à ce domaine. Cependant, après quelques recherches, nous pouvons tomber sur certains chercheurs dans le domaine de la technologie qui ont proposé leurs propres définitions. Cette difficulté vient du fait que l'intelligence elle-même n'est pas assez clairement définie comme concept. Or, l'intelligence artificielle revient dans un sens à définir l'intelligence en soit.

Voici quelques définitions proposées par différentes personnes dans le domaine informatique :

- « Le domaine de la science et de l'ingénierie qui traite de la compréhension à l'aide de l'ordinateur, de ce qui est appelé couramment le comportement intelligent, et de la création de systèmes artificiels qui reproduisent ces comportements. » [3].
- « Une branche de l'informatique qui concerne l'automatisation du comportement intelligent. » [4].
- « la recherche de moyens susceptibles de permettre à une machine d'exécuter des fonctions normalement associées à l'intelligence humaine : compréhension, raisonnement, dialogue, adaptation, apprentissage… » [5].
- « recherche de moyens susceptibles de doter les systèmes informatiques de capacités intellectuelles comparables à celles des êtres humains » Wikipédia (2).

Après avoir complété notre recherche, nous formulons notre propre définition qui nous a guidé tout au long de notre projet de recherche. L'intelligence artificielle peut être perçue comme l'automatisation du comportement humain à travers la programmation logique, une tentative de doter les machines d'une capacité d'apprentissage et d'une faculté de raisonnement logique et non préétablie, autrement dit la possibilité de réagir intelligemment ou par expérience. Nous avons choisi cette définition car nous ne considérons pas l'intelligence comme une faculté qu'on possède ou non à la naissance, mais comme une faculté qui se construit grâce à notre expérience et notre vécu. Apprendre à réagir et à se débrouiller dans les différentes situations (compliquées ou quotidiennes) vient de nos erreurs passées ou du vécu de nos proches.

1.1.3 Les différents domaines de l'intelligence artificielle

L'utilisation de l'intelligence artificielle ne cesse d'augmenter de jour en jour dans notre quotidien. Notre sécurité en dépend, nos moyens de transport y font appel, et nos gadgets électroniques ne jurent que par cette technologie. Nous vous présentons ici quelques exemples des différents domaines touchés par l'intelligence artificielle :

- **Reconnaissance et système de parole :**

La reconnaissance vocale prend de plus en plus d'ampleur dans notre quotidien. Elle offre de multiples services allant de la simple distraction, comme les technologies utilisées dans les jeux vidéo où les personnages obéissent aux ordres vocaux du joueur, à un outil indispensable comme les supports numériques offerts aux handicapés moteurs, qui à travers leurs voix peuvent commander les différentes fonctionnalités de leurs fauteuils roulants.

Cet apport de l'intelligence artificielle se développe de plus en plus, de telles sortes qu'avec les nouveaux téléphones intelligents, il est possible d'effectuer n'importe quelle tâche juste par commande vocale. La technologie « Siri » utilisée par Apple pour ces téléphones intelligents en est un très bon exemple. En effet, en utilisant la commande vocale pour communiquer avec « Siri », nous pouvons demander oralement de trouver le meilleur chemin pour une destination particulière, d'envoyer un message texte à une personne ou même de lire le bulletin d'information de la journée.

- **Reconnaissance et système d'images :**

Un exemple simple d'utilisation de l'IA dans ce domaine se trouve dans les appareils photo. De nos jours, tout appareil photo moderne est capable de détecter les différentes formes se trouvant dans son champ de vision. Un exemple très commun est le détecteur de visage, qui grâce à la détection des yeux et d'une forme arrondie du visage, peut en conclure que c'est un visage humain. C'est le même principe pour le détecteur de sourire.

Cette technologie peut être plus poussée, pour offrir des services plus importants encore, comme le système de reconnaissance de personnes développé par Acagi Inc en partenariat avec l'université du Maryland pour l'armée américaine. Grâce à une caméra placée dans le casque ou le fusil du soldat, le système conçu permet de détecter et identifier tous les visages qui passent devant l'œil de la caméra, ce qui permet de savoir si l'individu en vue représente une menace ou non. [6]

La reconnaissance faciale ou digitale est aussi devenue une référence dans le domaine de la sécurité. Elle remplace les clés des maisons et les codes numériques dans les téléphones portables, coffres forts et ordinateurs.

- **Reconnaissance de l'écriture :**

La reconnaissance de texte est moins en vue que celui de la reconnaissance vocale ou d'imagerie. Cependant, elle est en train de devenir le moyen le plus important dans la numérisation des œuvres manuscrites. Il s'agit ici d'un processus d'intelligence artificielle plus complexe, car interagissant avec des éléments réels, c'est-à-dire que ce n'est pas une simple image ou une séquence vocale qui est traitée, mais du texte avec des lettres distinctes et des mots différents. L'aspect sémantique est à prendre en compte.

Par exemple, les nouveaux téléphones intelligents sont dotés de fonctionnalités permettant de reconnaitre, traduire et sauvegarder du texte rien qu'en pointant la caméra de l'appareil sur ce texte.

- **Aide à la décision (les jeux)**

Le domaine d'aide à la décision est celui qui est le plus utilisé dans le monde professionnel et le monde de la recherche et optimisation, parmi toutes les autres filières de l'intelligence artificielle. Il couvre de multiples domaines de recherche comme notamment :

- les agents adaptatifs et les agents décisionnels ;
- l'apprentissage et la fouille de données ;
- l'optimisation et la résolution de problèmes ;
- les bases de données intelligentes ;
- le traitement d'informations multimédia ;
- les interactions personne/machine.

Ce concept se retrouve beaucoup aussi dans les jeux à « tendance intelligente » comme les jeux d'échecs.

- **Apprentissage et adaptation (les jeux)**

Ce domaine est apparu grâce à l'intelligence artificielle, et est aussi appelé apprentissage automatique. Il est défini comme la capacité d'un système à améliorer ses performances via des interactions avec son environnement.

Remarque : La plupart des types de reconnaissances (d'images, de textes, de voix) se basent sur un des domaines les plus connus de l'intelligence artificielle, soit la planification.

- **Systèmes à base de connaissances**

Un système à base de connaissances ou système-expert, selon la définition proposée par Pomerol [7] est un outil informatique d'intelligence artificielle, conçu pour simuler le savoir-faire d'un spécialiste, dans un domaine précis et bien délimité, grâce à l'exploitation d'un certain nombre de connaissances fournies explicitement par des experts du domaine.

Une base de connaissances représente à la fois le savoir-faire et l'expertise nécessaires pour résoudre un problème. Les unités de raisonnement s'écrivent généralement sous la forme de règles libellées de la façon suivante : si "situation" alors "action", la situation correspondant à l'hypothèse de la règle et l'action à la conclusion. [8]

En conclusion l'intelligence artificielle grâce aux nombreux et importants domaines qu'elle touche, possède un grand impacte dans notre quotidien, mais les découvertes et avancées technologiques ne cessent de se manifester de jour en jour. Jusqu'au nous mènera ces avancées, à quel point l'intelligence artificielle impactera notre vie et ou que peut-on espérer comme invention futur grâce à cette technologie révolutionnaire?

1.1.4 Le futur de l'intelligence artificielle

L'intelligence artificielle est un domaine assez jeune par rapport à aux autres branches de l'informatique comme l'algorithmique ou la cryptologie. C'est l'une des raisons pour lesquelles elle est considérée comme l'un des domaines de l'informatique avec le plus de recherches. Par conséquent, les découvertes et améliorations de ce domaine n'arrêtent pas de progresser. L'intelligence artificielle prend de plus en plus de place dans notre quotidien, et cela grâce à son impact dans différentes sphères comme la communication, l'éducation ou encore la sécurité. Mais jusqu'où pourrons-nous aller avec un rythme de progression aussi rapide ? C'est une question à laquelle nous ne pouvons répondre. L'un des objectifs ultimes des scientifiques dans ce domaine reste celui de pouvoir simuler l'intelligence humaine et le comportement d'un être humain doté de conscience et de sentiments.

Nous pouvons toutefois citer quelques exemples de projets futurs proches présentés par des firmes de technologie ou des chercheurs travaillant en intelligence artificielle :

- le premier androïde : commercialisation prévue dans 10 ans par la communauté scientifique asiatique ;

- l'amélioration de la relation personne-robot : construction de robots qui pourront entrer en empathie avec l'être humain, l'écouter et comprendre ces sentiments, par la société Honda ;

- la voiture intelligente : commercialisation d'un véhicule qui peut se déplacer d'un point à un autre sans aucune intervention humaine et en garantissant une sécurité routière totale par la société Google.

Avec tous ces exemples, nous pouvons dire que l'intelligence artificielle commence à prendre un aspect incontournable dans notre société. Nous allons découvrir dans la section suivante une des composantes de cette technologie innovatrice : la programmation logique.

1.2 La programmation logique

La programmation logique se base sur un ensemble de faits élémentaires et de règles logiques associées à des conséquences. À la réception d'une question, le moteur d'inférence exploite les faits et règles définis précédemment pour répondre à la requête.

Si la programmation logique est considérée comme l'un des principaux outils de l'intelligence artificielle, c'est parce que ce type de programmation répond parfaitement au besoin de cette dernière. Ceci s'explique par le fait que la programmation logique est considérée comme une programmation déclarative plutôt qu'impérative, car elle répond d'avantage au quoi qu'au comment.

1.2.1 Programmation impérative et programmation déclarative

Afin de mieux comprendre ce qu'est la programmation logique et son fonctionnement qui traite du "quoi" plutôt que du "comment", nous allons présenter une vue globale des deux types de programmation existants : la programmation impérative et la programmation déclarative.

- La programmation impérative est la forme de programmation rattachée à trois concepts fondamentaux qui sont le test, l'ordre et l'itération. Elle est appliquée par les langages dits ' procéduraux '. « La caractéristique principale de cette programmation est de nécessiter l'expression, pas le détail, du 'comment '. Elle est la traduction d'une démarche algorithmique, au sens de la recherche d'automatisme et d'économie de pensée, où le passage des données vers les résultats est décrit comme une suite d'actions. ». [8]

- La programmation déclarative est appelée plus communément programmation logique. Elle a comme principale caractéristique la délégation de la charge du 'comment ' au langage lui-même. Dans ce type de programmation, le problème est décrit à partir des objets concernés, leurs propriétés et bien sûr les relations entre elles.

En programmation logique, deux extensions sont mises en évidence : la programmation logique inductive et la programmation par contraintes.

La programmation logique inductive est une technique d'apprentissage automatique qui consiste à parcourir l'ensemble des représentations possibles d'un concept, afin de découvrir celles qui décrivent le mieux l'ensemble des exemples, instances positives ou négatives de ce concept. La programmation logique inductive est particulièrement utile pour le traitement de la langue naturelle.

La programmation par contraintes est un paradigme de programmation apparu dans les années 1980 permettant de résoudre des problèmes combinatoires de grandes tailles tels que les problèmes de planification et d'ordonnancement.

> « *Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.* »
>
> — E. Freuder 1997 [9].

Le principal avantage ou apport de la programmation logique par rapport aux langages plus classiques, tels que Pascal, est que ces derniers sont de nature impérative. En effet, il faut décrire le problème à résoudre selon un algorithme, alors que les langages logiques sont de nature déclarative. Cela signifie qu'il suffit d'indiquer au système les données du problème à traiter. Ainsi avec cette nouvelle vision, nous pouvons résoudre des problèmes complexes sans avoir besoin de recourir à des techniques algorithmiques qui peuvent s'avérer difficiles voire impossibles à mettre en œuvre ou encore trop coûteuses.

1.2.2 Impact de la programmation logique sur les différents domaines de l'intelligence artificielle

Après avoir parcouru les différents domaines de l'intelligence artificielle dans notre vie de tous les jours, nous devons aussi savoir que l'intelligence artificielle elle-même détient plusieurs différents domaine, nous allons dans cette section mettre le point sur l'impacte de la programmation logique sur ces derniers.

Reconnaissance des images/ symboles :

Un bel exemple a été donné dans l'article « Inductive Logic Programming for Symbol Recognition » présenté lors de la 10éme conférence internationale sur l'analyse et la reconnaissance de document en 2009 [10] Cet exemple décrit un ensemble d'expériences autour de l'utilisation de la programmation logique inductive pour l'apprentissage automatique de descriptions de formes non-triviales, basée sur une description formelle.

Cette approche consiste en une description des symboles graphiques par un ensemble de primitives de complexité quelconque, et qui ne sont pas nécessairement des primitives habituellement rencontrées comme des lignes et des points. À cet ensemble, il est ajouté des relations de positionnement relatif. L'extraction des primitives ainsi que l'établissement des relations se basent sur des algorithmes classiques de l'état de l'art. Cette représentation est ensuite fournie comme entrée à un résolveur en programmation logique inductive qui en déduit des caractéristiques non triviales et qui pourront, ensuite, conduire à des processus de reconnaissance orientés sémantique.

Reconnaissance de problèmes/maladies en médecine :

Un exemple de l'utilité de la programmation logique dans des domaines très importants comme la médecine montre que la programmation logique inductive aide à détecter une arythmie cardiaque, considérée comme un danger pour le cœur humain [11]. La programmation logique inductive permet de constituer automatiquement une base de symptômes chroniques sous la forme d'une formule logique de premier ordre, qui permet de repérer une dizaine d'arythmies parmi les plus connues.

Traitement du langage naturel :

Plusieurs pistes ont étaient explorées dans le traitement du langage naturel. En fin de compte, tout le monde s'est mis d'accord sur le fait que la meilleure combinaison pour ce domaine est la programmation logique associée aux systèmes à base de connaissances. La programmation logique a pu dans ce domaine résoudre des problèmes non résolus avec d'autres modes de programmation utilisés avant elle. Une description intéressante est présentée dans un article du Laboratoire d'étude et de recherche en informatique d'Anger, où il est expliqué qu'à travers un cycle de traitements composé de trois phases, il est possible d'avoir des résultats très concluants. La première phase du cycle prenait en entrée les données (textuelles, numériques…) et, à travers une analyse sémantique de textes, de la fouille de données ou encore d'induction d'interface graphique de constitution de base de connaissances, nous avions en sortie une base de connaissances logique relationnelle. Cette même base est envoyée en entrée à la deuxième phase qui grâce aux systèmes inférentiels déductifs et interrogatifs génère des décisions. Des décisions qui se transforment en résultats finaux ou services grâce à la troisième phase qui se charge de l'enrichissement, de la génération textuelle et de la présentation graphique [12].

Après avoir fait le tour des principaux impacts de la programmation logique sur l'intelligence artificielle et démontré la diversité de cette dernière, nous allons parcourir les différents langages de la programmation logique et les décortiquer afin d'en apprendre plus sur ce type de programmation.

1.2.3 Les différents langages de programmation logique

Vu la large panoplie de possibilités que la programmation logique offre, de multiples langages de programmation logiques sont apparus au fil des années. Certains semblent similaires mais la plupart offre un mélange distinct de fonctionnalités, facilitant ainsi le choix du langage et la programmation nécessaire à la réalisation d'un projet. Nous vous présentons en suivant une liste de ces différents langages, leurs caractéristiques et pour finir une comparaison de ces langages.

PROLOG [13] : Le langage de programmation Prolog, PROgrammation LOGique, est né d'un projet, dont le but n'était pas de faire un langage de programmation mais de traiter les langages naturels, en l'occurrence le français. Ce projet a donné naissance à une version du langage préliminaire à la fin 1971 et une version plus définitive à la fin de l'année 1972.

LOGTALK [14] : est un langage de programmation logique orienté-objets apparu en 1998. Certains diraient qu'il est issu de Prolog, car il utilise la plupart des modules d'implémentation de ce dernier pour la compilation. Étant un langage multi-paradigme, il offre un support autant pour les classes que pour les prototypes. Il inclut aussi la programmation multitâches et les interfaces.

LISP [15] : ce langage, dont le nom a été forgé à partir de l'anglais « *list processing* » (traitement de listes), est considéré comme étant l'un des plus anciens langages de programmation (en excluant le langage assembleur). Apparu en 1958, il est basé sur le lambda-calcul. Il était le choix par excellence pour la recherche en intelligence artificielle des années 70 et 80. De nos jours, il est utilisé dans plusieurs domaines tels que la programmation web et la finance.

CLIPS [16] : (C Language Integrated Production System) est un environnement et un langage de programmation créé en 1985 par la section d'intelligence artificielle de la NASA afin de pallier le manque de performance du langage LISP et son incompatibilité avec leurs contraintes. CLIPS fait partie du paradigme des langages déclaratifs et logiques. Le langage CLIPS repose essentiellement sur un moteur d'inférences d'ordre 1 fonctionnant en chaînage avant, associé à un formalisme de représentation des connaissances, ainsi qu'à un langage de programmation (inspiré de Lisp).Il s'agit avant tout d'un outil de construction de systèmes à base de connaissances, systèmes qui raisonnent sur des connaissances d'un domaine particulier, ces connaissances étant représentées dans le cas de CLIPS par des règles expertes et/ou des objets [17].

OZ [18] : est un langage de programmation apparu la première fois en 1991 dans le laboratoire de programmation de l'université catholique de Louvain. Le projet a ensuite évolué à travers différentes équipes de chercheurs. Le langage OZ est actuellement pris en charge par le groupe Mozart. Sa première implémentation a vu le jour en 2008 avec le système de programmation « Mozart ». Le langage OZ permet d'employer et de combiner différents paradigmes de programmation. Il fournit par défaut des variables logiques. De même, l'évaluation est stricte par défaut, mais l'évaluation paresseuse est possible. La différence entre les deux types

d'évaluations est que la première se fait automatiquement dès qu'elle peut être liée à une variable, alors que, pour la deuxième, les expressions sont évaluées lorsqu'on en fait explicitement la demande (notamment lorsque l'on cherche à afficher un résultat).

PYTHON [19]: Créé en 1990 par Guido van Rossum pour la société Python Software Foundation, Python est un langage de programmation multi-paradigmes. Il favorise la programmation impérative structurée et orientée objet. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions. Il n'est pas développé pour la programmation logique en particulier, mais, grâce à une extension de ce logiciel, il devient ouvert à ce type de programmation.

MERCURY [20] : est un langage de programmation logique orienté-objet, qui a vu le jour en 1995 à l'université de Melbourne, développé pour la première fois par Fergus Henderson et Thomas Conway supervisé par le professeur Zoltan Somogyi. Ce langage est grandement influencé par les langages Prolog, Hope[21] et Haskell [22]. Il se distingue par sa combinaison de clarté et sa grande expressivité de la programmation déclarative grâce à une analyse statique avancée et sa détection d'erreurs dans les fonctions.

SCHEME [23] : est un langage de programmation apparu en 1975, développé dans un des laboratoires du MIT (Massachusetts Institute of Technology) par Guy L. Steele et Gerald Jay Sussman. Scheme est l'un des deux principaux dialectes du langage de programmation LISP (Common Lisp étant l'autre dialecte). C'est un langage qui a séduit différents types d'utilisateurs : programmeurs, designers, éducateurs, etc.

RACKET [24] : anciennement appelé « PLT Scheme », ce langage de programmation multi-paradigme fait partie de la famille Lisp/Scheme. C'est un langage assez autonome vu que tout son environnement est écrit en Racket même. Ce langage est utilisé pour différents types de programmation à savoir la programmation web, la programmation logique, la génération de analyseurs lexicaux et syntaxiques et bien d'autres.

O'CAML [25] : O'Caml est la principale implémentation du langage Caml (Categorical Abstract Machine Language), qui est un langage de programmation se distinguant en matière de sûreté et de fiabilité des programmes. Ce langage a été inventé en 1985 par l'institut national de la recherche en informatique et en automatique (INRIA) de même que O'Caml en 1996. Afin d'apporter à Caml un très bon constructeur orienté-objet et la programmation modulaire comme objectifs principaux, la gamme d'outils O'Caml inclut aussi un interpréteur de première qualité ainsi qu'un compilateur de bytecode.

Nous pouvons conclure que la popularité de la programmation logique ne cesse d'augmenter avec comme preuve la variété et le nombre de langages développés pour ce mode. Cependant, il est relativement complexe pour un programmeur non encore expérimenté de choisir le langage le plus adéquat pour l'application à implémenter. Nous proposons donc dans les sections suivantes une comparaison de ces différents langages afin d'aider à mieux comprendre leurs différences.

1.2.4 Comparaison de différents langages de l'intelligence artificielle

La comparaison qui suit a pour but de faciliter le choix à tout débutant en programmation logique de faire correspondre ses attentes et ses besoins aux fonctionnalités, que peut offrir un des différents langages étudiés. Pour y arriver, nous avons commencé par choisir les critères sur lesquels nous avons comparé ces langages. Nous avons ensuite indiqué dans un tableau les correspondances entre ces critères et les langages. Pour finir, nous avons comparé chacune des fonctionnalités qu'offre chacun de ces langages.

Critères de comparaison :

- Multi-paradigme : Le langage peut supporter plusieurs paradigmes de programmation (programmation procédurale, programmation orientée-objet, etc.).
- Interaction avec d'autres langages : Le langage de programmation peut être fusionné avec un autre langage dans un même programme.
- Simplicité de la syntaxe et de la sémantique : le niveau de difficulté avec lequel un programmeur apprend à maîtriser un nouveau langage.
- Possibilité de programmation fonctionnelle : Ce langage peut tout aussi bien être utilisé pour la programmation fonctionnelle qui est un paradigme très utilisé dans le monde de la programmation.

Le tableau suivant met en correspondance ces critères avec les différents langages présentés dans la section précédente.

Langages/Critères	Multi-paradigme	Interaction avec les autres langages	Simplicité en syntaxe et sémantique	Supporte la programmation fonctionnelle
Prolog	-		+	-
LISP	+	-	-	+

CLIPS	**+**	**+**		**-**
LOGTALK	**+**		**+**	**-**
OZ	**+**	**+**	**+**	**+**
PYTHON	**+**	**+**		**+**
Mercury	**-**		**+**	**+**
Scheme	**+**	**-**	**-**	**+**
Racket	**+**	**-**	**-**	**+**
O'caml	**+**	**+**	**+**	**+**

Tableau1 Comparaison des langages de programmation logique

Afin d'indiquer si un langage atteint un critère, nous avons indiqué le symbole "+", et "-" pour indiquer qu'il ne l'atteint.

1.2.5 Analyse de la comparaison

Il existe différents langages de programmation, et s'il en existe autant c'est parce que chacun d'entre eux a ces caractéristiques propres qui le distinguent des autres. Selon les besoins du programmeur, ce dernier peut en favoriser un plutôt qu'un autre. Prenons comme exemple le langage Prolog, vu que le but original de ce langage était de procurer un outil pour les linguistes ignorant l'informatique, sa syntaxe et sa sémantique sont considérées comme très simples et claires. C'est donc un parfait outil pour quelqu'un qui n'est pas encore expert en programmation ou qui veut simplement implémenter un programme sans beaucoup de difficulté. Nous pouvons en dire autant pour le langage Mercury qui utilise la même syntaxe que Prolog, ainsi que Lisp qui a la réputation de se distinguer par une syntaxe simple.

Pour en revenir au langage Prolog, sa réversibilité lui donne un sérieux avantage dans la gestion des requêtes sous-contraintes/sur-contraintes. L'absence de statut pour les paramètres d'un prédicat (cf. réversibilité) et le modèle d'exécution employé permet, d'un côté, l'emploi de requêtes sous-contraintes exposant l'ensemble des possibles, et de l'autre côté, l'emploi de requêtes sur-contraintes permettant la vérification de propriétés particulières sur les solutions exhibées ou le filtrage de ces solutions. Cette même absence de statut pour les paramètres est considérée par certains comme un important inconvénient dans la programmation. C'est pourquoi cette catégorie de personnes est plus portée vers d'autres langages comme Logtalk, qui offre la possibilité de paramétrer les prédicats (il supporte les objets statiques et dynamiques) avec « private », « protected » ou « public ». Pour les spécialistes du domaine, Logtalk offre une plus grande facilité et efficacité pour ce qui est de la séparation entre l'interface et l'implémentation et est meilleur en termes de portabilité que les modules de Prolog. Cependant, il est toujours possible de combiner les deux langages.

Pour continuer avec les inconvénients de Prolog, vu que c'est un langage faiblement typé, cela fait de lui un langage assez difficile à débugger. Sans le typage, les problèmes de débogage se multiplient et il y a risque de violation de l'intégrité du domaine des bases de données.

Avant de commencer à développer une application, un programmeur doit avoir une bonne idée sur la conception de cette dernière. En fonction de cela, il aura le choix d'opter pour un langage comme Lisp ou comme CLIPS. Le langage LISP offre un typage dynamique des données ainsi que le support de la programmation fonctionnelle. Il se distingue surtout par sa gestion automatique de la mémoire et sa faculté de manipuler le code source en tant que structure de données. LISP allie une grande précision à une puissance considérable quand il s'agit de développer une application simple, qui est conçue pour une tâche précise. Cependant, son inconvénient majeur est la complexité de sa syntaxe. Par contre, le langage CLIPS, qui en plus d'être rapide, efficace et gratuit, intègre un langage orienté-objet assez complet pour développer des systèmes à base de connaissances grâce à ses différents paradigmes de programmation. Contrairement à Lisp, il est fait pour les grosses applications complexes, la preuve en est que Clips est largement utilisé dans les gouvernements, les industries et les académies.

Dans le même type de langage (ayant la caractéristique d'être plus global que centré sur un type précis de programmation), SCHEME est un sérieux concurrent pour CLIPS, grâce à son élégance et à un design minimaliste. Le langage SCHEME est devenu une excellente solution pour les designers de langages et les éducateurs. Ce langage est d'ailleurs utilisé par de nombreuses écoles d'informatique à travers le monde pour la simplicité de sa syntaxe entre autres. Cette même caractéristique qui a fait de SCHEME un langage populaire est aussi son principal point faible. Son aspect global a comme conséquence une large diversité entre les implémentations.

D'autres caractéristiques importantes sont prises en compte lors du choix d'un langage de programmation logique, l'une d'entre elles est la sécurité, domaine ou le langage OZ excelle. Toutes les entités du langage sont créées et introduites explicitement. Une application ne pourra donc jamais créer ou forger une référence qui ne lui a pas été passée explicitement. Plus encore, le programmeur n'a pas accès à la base des représentations des entités du langage. Ces concepts mélangés à quelques autres sont essentiels pour assurer une très bonne politique de sécurité, un autre des avantages qu'offre le langage OZ, une bonne et transparente distribution. Plusieurs sites OZ peuvent communiquer et se partager des variables, objets ou classes, comme si c'était une seule station, et cela en toute sécurité.

Certains développeurs se basent sur des langages existants pour créer de nouveaux langages. C'est ce que les programmeurs du langage Mercury ont fait. Ils se sont basés sur le langage Prolog et se sont fixés comme objectif de l'améliorer. Ils ont repris la même syntaxe et les concepts de Prolog mais ont préféré séparer la phase de compilation, ce qui permet d'identifier un plus grand nombre d'erreurs avant l'exécution du programme. Les développeurs prétendent que c'est le successeur de Prolog, et, grâce aux informations obtenues lors de la

compilation, les programmes écrits en Mercury gagnent significativement en rapidité par rapport à ceux écrits en Prolog. Les auteurs du langage Mercury prétendent même que c'est le langage de programmation le plus rapide au monde et de loin.

Un autre type de langage pourrait intéresser les programmeurs ayant à représenter des types algébriques, autrement dit hiérarchisés et possiblement récursifs, c'est le langage O'Caml. En plus de son typage statique, fort et inféré qui lui permet de manipuler aisément des structures de données assez complexes, c'est un langage réputé pour sa sûreté, un atout de plus surtout quand on manipule des données complexes.

En conclusion, on ne peut dire que l'un de ces langages est meilleur que les autres, plus populaire peut être mais pas meilleur. L'efficacité et la performance vont dépendre du contexte d'utilisation.

Conclusion

Suite à cette étude de la programmation logique, nous constatons que ce type de programmation a eu et a toujours un énorme impact sur l'intelligence artificielle dans tous ses domaines. Ce type de programmation a permis non seulement d'améliorer considérablement les pratiques existantes, mais aussi d'apporter des innovations dans les différents domaines. Il est à noter que la programmation logique est même enseignée en dehors des sections informatiques, car elle aide à structurer la pensée humaine, ce qui élargit encore plus l'impact de ce type de programmation dans notre vie de tous les jours.

En programmation logique, ce n'est pas le programmeur qui donne les étapes de la résolution de problèmes puisqu'on se focalise sur le quoi. Quels sont les outils à la disposition du programmeur pour comprendre comment se passe cette résolution de problèmes ? Cette question est importante car il est difficilement concevable de dissocier la bonne programmation de la compréhension de la résolution. Dans le prochain chapitre, nous présentons les résultats de notre étude des outils permettant d'aider la compréhension de la résolution de problèmes en programmation logique. De cette étude, nous avons mis en évidence une problématique que nous avons souhaité résoudre en fixant des objectifs de recherche.

Chapitre 2 : Problématique et objectifs de recherche

Après avoir détaillé la programmation logique, nous nous sommes attaqué dans ce chapitre au principe de la compréhension de cette programmation et de ses principes. Nous allons voir qu'elle est la meilleure façon de faciliter cette compréhension et ce qui a été déjà proposé dans la littérature comme outils pour répondre à ce besoin. En nous basant sur cette étude, nous avons mis en évidence un certain nombre de lacunes. Ceci nous a permis d'identifier ce que nous pouvons apporter de plus et qui pourrait aider un programmeur débutant en programmation logique à assimiler son fonctionnement et son exécution plus facilement.

2.1 Les moyens pour faciliter la compréhension de la résolution en programmation logique

En programmation, il n'est jamais facile de décoder ou comprendre un algorithme ou un programme seulement en le parcourant. Quelquefois même à travers des jeux d'essais, il existe des fonctionnalités ou des choix qui peuvent nous échapper. Plusieurs études ont été faites sur ce sujet. L'une d'entre elles a été réalisée par un groupe de recherche de l'université Duke à Durham, NC aux débuts des années 90 [26]. Cette étude a permis de démontrer qu'un étudiant apprenait plus facilement et comprenait mieux un algorithme après avoir visualisé son effet ou impact en temps réel, autrement dit en le regardant s'exécuter sous forme graphique, c'est ce qu'on appelle avoir une **trace de résolution**.

Une trace de résolution peut être décrite comme étant la présentation des différentes étapes de résolution d'un problème, commençant par les éléments du problème jusqu'à arriver à la solution trouvée. Autrement dit, la trace de résolution nous montre comment la solution a été trouvée. Une trace de résolution peut être facilement représentée sous forme d'un texte ou d'un tableau. Un texte correspond à une description textuelle du parcours exact des éléments de la solution choisie dans programme étape par étape. On peut également utiliser un tableau à deux dimensions représentant d'un côté les différentes étapes jusqu'à la solution choisie et de l'autre l'évolution des différentes variables de la solution tout au long du programme. Pour ce qui est de la forme graphique, on fait généralement appel aux arborescences.

Il existe de multiples types d'arborescences dans le monde de l'informatique. Chaque type est utilisé selon ses caractéristiques dans un même domaine ou dans différents domaines. Nous avons limité notre intérêt aux arbres de dérivation, car c'est le seul type d'arborescence approprié pour notre projet de recherche. Les arbres de dérivation ou arbres syntaxiques sont surtout utilisés dans le domaine du langage automate et celui de la compilation. Un arbre de dérivation peut être défini comme étant une représentation graphique d'une phrase ou

d'un ensemble de mots d'une grammaire donnée, c'est-à-dire qu'il décrit graphiquement, et sous la forme d'un arbre (inversé) comme son nom l'indique les différentes possibilités d'obtenir cette phrase ou cet ensemble de mots donné à partir d'une grammaire définie.

Tant que le résultat, la solution ou l'approche à décrire est décomposable en phases ou étapes, nous pouvons les représenter à travers une arborescence. La figure 1 ci-dessous représente la similarité entre un arbre et une arborescence informatique, avec le sommet de l'arbre comme état initial et ses feuilles comme étant les différents chemins ou phases par lesquelles nous pouvons atteindre un but ou un état donné. Dans sa forme la plus simple, cette représentation consiste à avoir un triangle où le sommet de ce dernier représente un axiome, autrement dit le symbole de départ de la grammaire représentée, et les extrémités du triangle représentent les deux choix possibles à partir de l'axiome en termes de symboles terminaux, soit les mots donnés. Pour des grammaires plus complexes, l'arborescence correspond à la fusion de plusieurs arbres (triangles). Les extrémités (feuilles) ou symboles non terminaux sont remplacés dans l'arbre correspondant et ainsi de suite jusqu'aux symboles terminaux, soit les mots de la phrase. Dans l'exemple de la figure 1, le sommet du triangle identifie par "S" la représentation syntaxique d'une phrase qui est composée d'un groupe nominal "N" et d'un groupe verbal "V" qui sont présentés par les deux extrémités du triangle, Ensuite, le groupe Verbal "V" devient à son tour un sommet avec comme extrémités un verbe "V" qui est "dort" et un adverbe "ADV" qui est "énormément". Le groupe nominal N n'a pas encore été dérivé jusqu'aux symboles terminaux.

Figure 1 : Arborescence

La principale raison de l'utilisation des arborescences graphiques est d'expliquer l'acheminement entre les données de bases d'un problème et la ou les solutions possibles de ce dernier. Cette méthode d'explication s'est avérée la plus efficace grâce à son côté graphique, car comme le disait William Playfair, inventeur du concept de l'histogramme dans son livre Political Atlas : *"En fait de calculs et de proportion, le plus sûr moyen de frapper l'esprit, est de parler aux yeux."*. L'être humain a généralement plus de facilité à assimiler les concepts reliés entre eux, quand ils sont représentées graphiquement plutôt que décrits par des mots.

Nous pouvons prendre un exemple simple afin visualiser cette transformation :

Soit une grammaire ‘G’ avec un axiome ‘S’, qui a pour règles de production S⟶aSbS et S⟶ε (Le symbole ε représente l’état nul, c'est-à-dire que la branche le l’arbre s’arrête la). En combinant toutes les possibilités réalisables avec ces deux règles de production, nous obtenons la liste des mots que peut générer la grammaire G. Par exemple, pour obtenir le mot ‘aabbab’, nous devons utiliser comme point de départ l’axiome S, puis dérivons cet axiome à l’aide des deux règles production données. La figure 2 présente le résultat de cette dérivation. Au premier niveau, nous utilisons la règle S⟶aSbS. Au deuxième niveau, nous dérivons les deux symboles S obtenus précédemment. Ensuite, chaque nouveau symbole S obtenu est dérivé en utilisant la règle de production S⟶ε. Nous obtenons seulement des symboles terminaux et donc la dérivation est terminée. Le mot ‘aabbab’ a bien été dérivé à partir de la grammaire donnée.

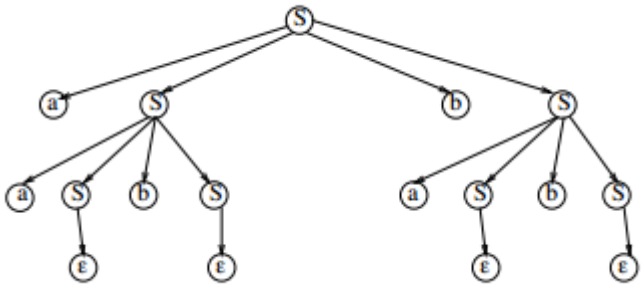


Figure 2 : Arbre de dérivation de 'aabbab'

L’utilisation des arbres a beaucoup apporté dans le domaine des automates, très répandu et utilisé en mathématiques et en informatique pour sa représentation adéquate de plusieurs équations mathématiques ou solutions informatiques. C’est le cas du raisonnement de la programmation logique, dont la structure se représente parfaitement sous la forme d’une arborescence. Les nœuds représentent les prédicats et leurs enfants de possibles résultats de chaque prédicat. Cette méthode de représentation a donc un grand impact dans le monde de la recherche. Cependant, la génération des arbres de dérivation a toujours été réalisée manuellement et très peu de chercheurs ou de développeurs ont cherché à l’informatiser ou à l’automatiser. Il n’existe aucun logiciel ou application développée et mise sur le marché par les grandes entreprises de développement capable de réaliser ce type de traitement. Cependant et après des recherches approfondies, nous avons trouvé des applications développées en interne ou à des fins personnelles traitant les arbres de dérivation.

Nous décrivons dans la prochaine section les applications trouvées dans la littérature qui ont été développées pour générer des arbres de dérivation et les afficher.

2.2 Description des générateurs d'arbres de dérivation

Comme mentionné précédemment, en faisant quelques recherches approfondies, nous avons trouvé quelques applications qui génèrent des arbres de dérivation pour différentes raisons. Nous avons pu faire le tour de ces applications, de leurs conceptions, et explorer leurs avantages et inconvénients. Nous avons conclu qu'en conservant les aspects positifs de chacune d'elles et en les fusionnant, nous pouvons proposer une application intéressante pour notre contexte.

Une première application permettant la génération automatique d'arbres de dérivation a vu le jour à l'an 2000 avec le développement d'une plateforme générale de 'LTAG' (Lexicalized Tree Adjoinning Grammar)[27]. C'est un ensemble d'outils de manipulation graphique de grammaires d'arbres adjoints développé par Patrice Lopez, un chercheur du centre allemand de recherche en intelligence artificielle « German Research Center for Artificial Intelligence » DFKI*.

Cet outil est totalement codé en JAVA, ce qui lui offre une assez grande portabilité et une intéressante réutilisabilité de ses propriétés. L'encodage des ressources et des résultats se font avec le formalisme le plus portable : XML. Pour ce qui est du côté graphique, Lopez a choisi de se baser sur un éditeur général d'arbres développé par la société Française « Thomson-CSF », maintenant connue sous le nom de « Thales ».

Lopez a utilisé dans sa plateforme certains principes intéressants appuyant l'apprentissage par visualisation, à savoir l'exploitation des principales ressources virtuelles afin de simplifier la manipulation et la gestion de toute grammaire. Cet outil offre aussi la possibilité d'encoder la redondance des sous-structures afin de pouvoir afficher les sous-arbres, les caractéristiques de leurs structures et celles des équations.

Comme nous pouvons le voir dans la figure 3, l'application est composée de trois sections :

- ✓ un entête qui traite le découpage des données, donc les unifications, la stratégie de découpage et les résultats à afficher;
- ✓ une deuxième section (à gauche) qui donne la main à l'utilisateur pour choisir le fichier à traiter;
- ✓ une fenêtre principale dans laquelle se fait l'affichage des arborescences finales.

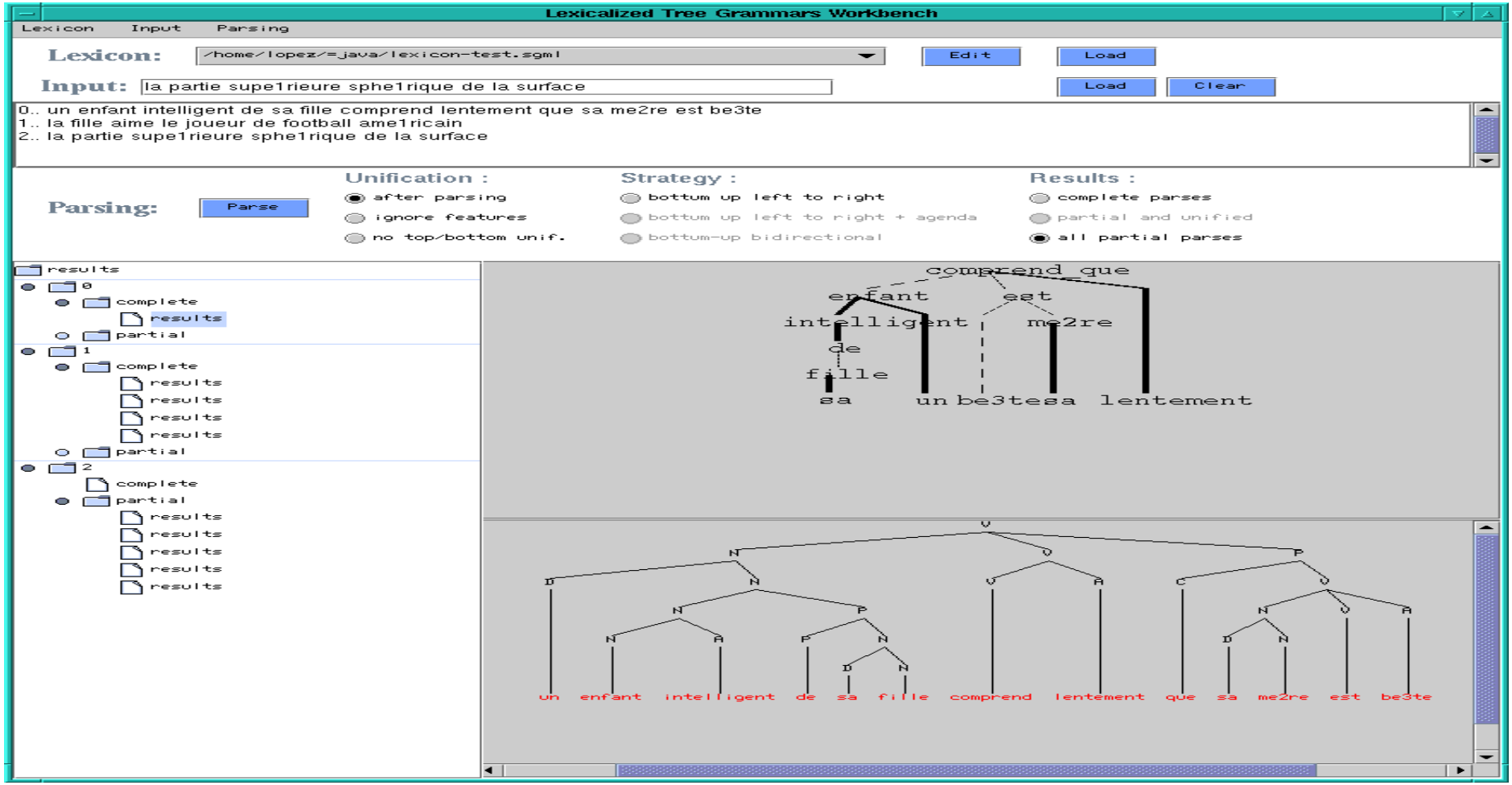


Figure 3 : Interface de la plateforme LTAG (tirée de [27])

Plusieurs développeurs ou chercheurs se sont inspirés de la plateforme de Lopez, présentée précédemment, afin de développer leurs propres applications. Par exemple, Roussanaly a développé l'application 'LLP2*' (Loria LTAG Parser 2)[28], un analyseur LTAG basé sur les travaux de Lopez.

Nous pouvons citer un autre exemple, celui d'un groupe de chercheurs français spécialisés dans le domaine des grammaires d'arbres adjoints. En 2001, ce groupe a repris et enrichi l'interface proposée par Lopez. Ils lui ont ajoutée comme principale innovation la possibilité d'avancer et de reculer dans les étapes de dérivation, en observant à chaque pas l'arbre partiel de dérivation et l'arbre partiel d'analyse [29].

La capture d'écran de l'application, présentée à la figure 4 ci-après, nous confirme que l'interface est bien inspirée de celle proposée par Lopez. Si nous enlevons la partie qui gère le découpage dans l'interface de l'application LTAG, nous obtenons la même interface proposée par [29].

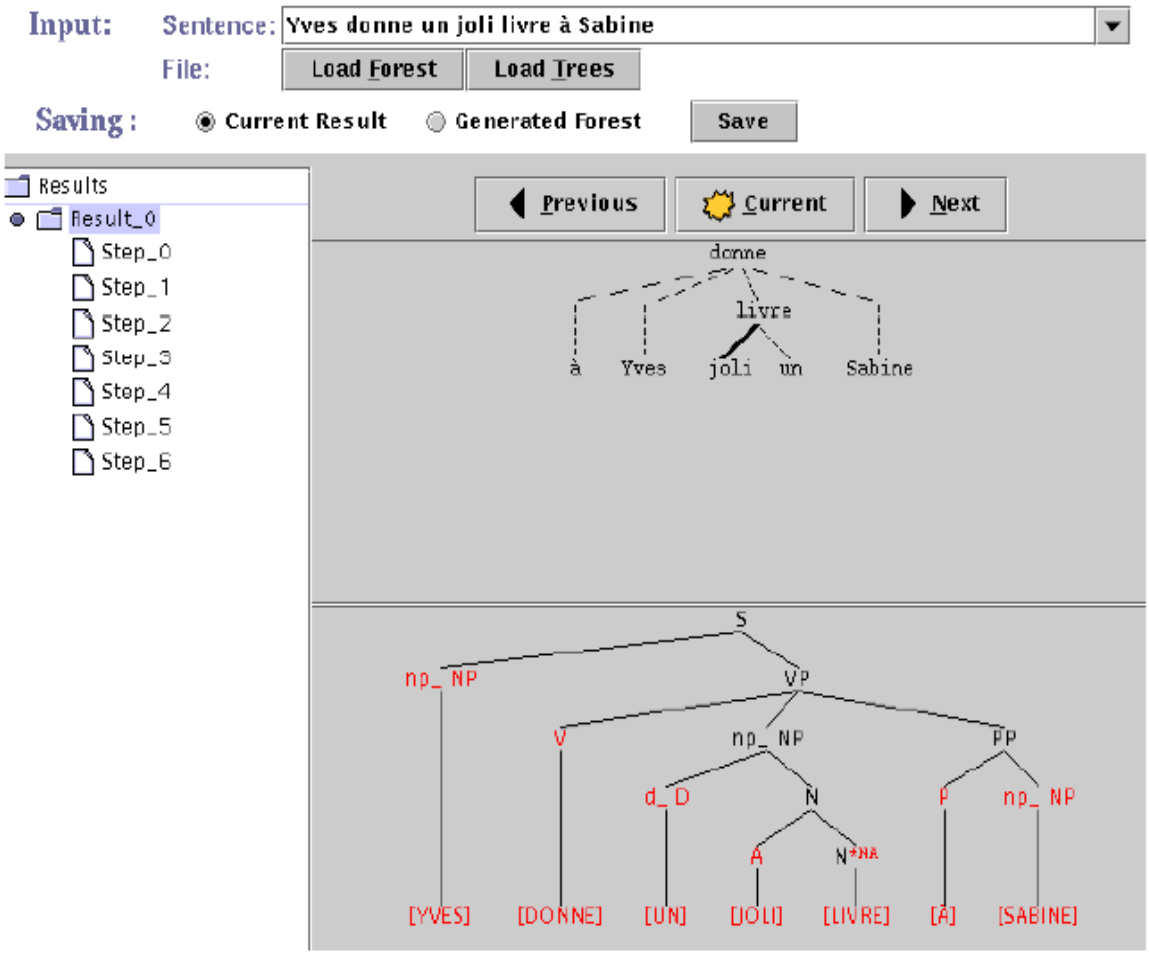


Figure 4 : Interface de visualisation (tirée de [29])

D'autres chercheurs ont aussi développé leur propre application. Par exemple, au début des années 90, des chercheurs de l'université Duke à Durham, NC [30], ont décidé de développer une collection d'outils qui vise à expliquer graphiquement et en temps réel les modifications et les mises à jour des graphes quand on travaille sur la théorie des automates et le langage formel.

Cet ensemble d'outils a été développé principalement en Java. Seuls les analyseurs ont été implémentés en C++. Parmi les nombreux outils offerts, on trouve un générateur d'arbres de dérivation très intéressant nommé 'PumpLemma'. L'utilisateur entre la grammaire, puis le mot à analyser, et voit apparaitre un arbre de dérivation de cette grammaire avec les différentes possibilités de retrouver le mot en coloriant les nœuds concernés.

Une capture d'écran de l'interface de ce générateur est présentée dans la figure 5. Un champ texte est situé tout en haut de l'interface, dans lequel l'utilisateur peut entrer le mot à analyser. Ensuite, il lui suffit de cliquer sur le bouton "Parse" pour réaliser le découpage. Finalement, le bouton "Derivation : Graphical Output" juste en dessous permet l'affichage de l'arbre dans la fenêtre principale de l'interface.

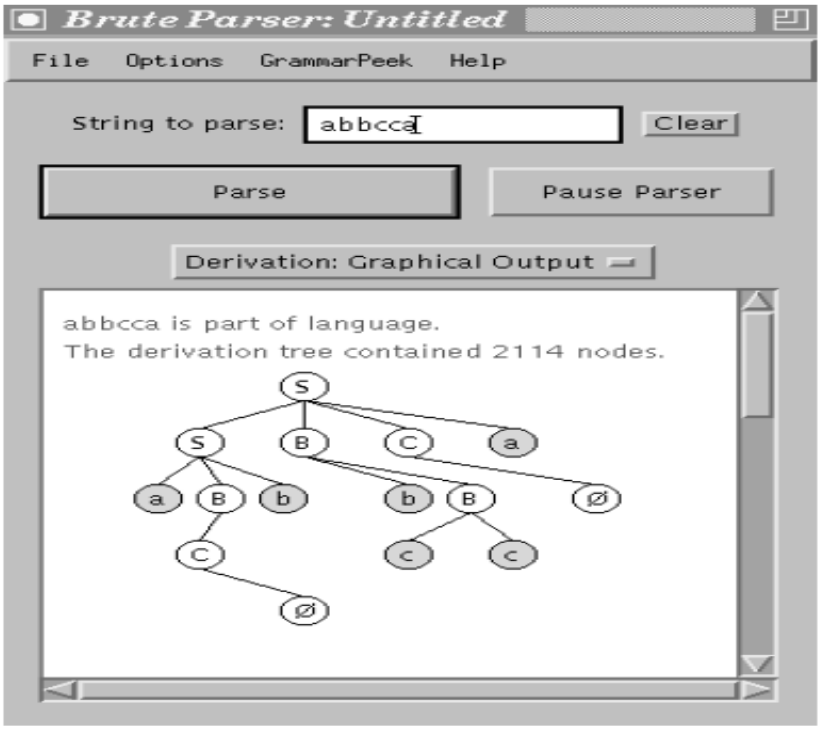


Figure 5 : Interface de visualisation (tirée de [30])

Nous pouvons conclure que les travaux et projets existants dans le domaine sont peu nombreux, et assez similaires. Par conséquent, ils ont les même avantages et inconvénients.

2.3 Analyse des générateurs d'arbres de dérivation existants

En analysant les trois applications décrites précédemment, nous remarquons que d'une manière générale leur conception est réalisée dans un but avant tout professionnel plus que formatif, ce qui justifie le peu d'interactions entre l'utilisateur et l'arborescence générée. Les liens entre les nœuds, et les nœuds eux-mêmes, sont des objets statiques que l'on ne peut explorer.

Dans le cas de la plateforme LTAG, le chemin de la résolution choisi est tracé par des lignes en gras. Cela nous indique bien les nœuds choisis par la résolution, mais cela ne nous permet pas de voir le parcours en temps réel de l'exécution de la fonction. Autrement dit si par exemple, il y a eu un retour arrière dans l'exécution, l'utilisateur ne pourra le remarquer, élément qui serait intéressant à voir pour la compréhension du déroulement de la résolution en temps réel.

Les trois applications analysées offrent une arborescence claire et facile à lire, et une simplicité d'affichage qui se base sur de simples droites comme liens entre le nom des nœuds parent et enfant aux deux extrémités de ces droites.

Grace à l'utilisation de « Java » comme langage de programmation principal, les applications existantes ce dotent d'une grande portabilité, une capacité à fonctionner facilement dans différents environnements, ce qui augmente le nombre d'utilisateurs et offre une grande facilité d'adaptation.

À la clarté de lecture des arborescences, nous pouvons ajouter la simplicité d'utilisation aux points forts de ces applications. En effet, grâce à une interface claire et bien décomposée, nous pouvons utiliser l'application sans avoir recours à un manuel d'utilisation. Des boutons bien placés permettent une manipulation aisée de l'interface et de l'arbre généré.

Comme toute application, celles-ci ont des points forts mais aussi des points faibles. La plupart des applications existantes offrent peu d'interactions avec leurs utilisateurs. Une fois l'arborescence affichée, il n'est plus possible d'interagir avec cette arborescence. L'utilisateur ne peut pas cliquer sur un nœud pour l'analyser ou en arrêter l'affichage afin de comprendre une étape d'exécution. L'une des raisons, pour laquelle cette interaction n'est pas possible, est que l'affichage des arbres se fait sous forme statique. C'est donc une simple image qui est projetée dans l'interface de l'application et ne permet pas d'interagir avec elle.

Ce manque d'interactions avec les arborescences est la source d'un autre point faible dans la plupart de ces applications. Les nœuds sont représentés simplement par leurs noms et il n'y a aucune possibilité d'avoir plus d'information sur le nœud ou la fonction qu'il représente.

En résumé, ce qui est proposé actuellement consiste en des implémentations efficaces de générateur automatique d'arborescences. Le résultat est une façon rapide et simple d'obtenir un arbre descripteur du langage, grammaire ou mot que nous voulons découvrir. Mais les affichages proposés n'aident pas l'utilisateur à comprendre parfaitement et en temps réel le déroulement de l'exécution. Par exemple, l'utilisateur ne peut pas obtenir des réponses aux questions suivantes : pourquoi avoir choisi ce sous-arbre et pas un autre ? Pourquoi avoir évité ou choisi tel nœud ou tel autre ? Etc. Ainsi, nous souhaitons répondre à la question à savoir : comment afficher un arbre de dérivation de la trace de résolution de programmes logiques pour aider un utilisateur à mieux la comprendre ? Pour répondre à cette question, nous nous sommes fixé un objectif qui porte notamment sur la réutilisation des fonctionnalités pertinentes des générateurs existants comme la portabilité obtenue grâce à l'utilisation du langage Java, la simplicité d'utilisation de l'interface et surtout la clarté des arbres de dérivations générés. Nous ajoutons à cela la nécessité d'intégrer une bonne interaction entre l'utilisateur et le générateur..

2.4 Objectif général et méthodologie

Nous avons exploré la programmation logique, ses apports et les différents langages qui la supportent. Puis, nous avons étudié les arbres de dérivation et l'automatisation de la génération de ces arbres. Nous avons constaté que les outils existants pour la génération automatique de ces arbres offrent différentes fonctionnalités, mais qu'aucun d'entre eux n'est vraiment conçu pour faciliter la compréhension du déroulement des traces de résolution. La majorité se contente simplement d'afficher l'arbre de résolution résultant.

Nous avons donc proposé de reprendre les bonnes fonctionnalités existantes que nous avons identifiées dans notre étude, puis de les enrichir avec d'autres modules afin de concevoir un nouveau modèle d'affichage graphique amélioré. Le but principal est de présenter graphiquement une trace de résolution d'un programme logique plus adapté à notre contexte, soit pour améliorer la compréhension de l'utilisateur.

En d'autres termes, le prototype construit à partir de ce modèle pourra présenter en sortie un arbre de dérivation expliquant les étapes d'exécution. Ainsi, quand un utilisateur lance un programme, il obtient un schéma graphique à travers lequel il peut suivre les étapes de l'exécution permettant d'atteindre la solution. L'objectif de notre modèle est d'offrir une interaction avec l'utilisateur. Chaque nœud de l'arbre contient une étape dans l'accomplissement de l'exécution avec laquelle l'utilisateur peut interagir pour mieux comprendre le parcours d'exécution.

Afin d'atteindre notre objectif, nous avons commencé par modéliser l'application envisagée en définissant une structure, des fonctionnalités principales et les différentes classes qui la composent. Nous avons utilisé les différents diagrammes UML pour cette description. Ensuite, nous avons construit un prototype afin de montrer la faisabilité de notre modèle. Nous avons eu recours à une étude de la compatibilité des interfaces graphiques

avec le langage Prolog, le langage de programmation logique retenu pour la réalisation de notre prototype. Le prototype a été implanté en langage Java, choisi en grande partie pour sa grande portabilité. Enfin, nous avons testé notre prototype afin de nous assurer de sa fonctionnalité et d'ajuster notre modèle au besoin. Afin de vérifier l'efficacité du modèle proposé, il serait intéressant de le mettre à disposition d'un groupe d'étudiants afin de recueillir leur avis. Une telle expérimentation dépasse le cadre de ce travail de maitrise et ne sera donc pas traitée.

Nous présentons dans le prochain chapitre le modèle permettant de générer automatiquement des arbres de dérivation en vue d'aider les utilisateurs à mieux comprendre les traces de résolution en programmation logique.

Chapitre 3 : Modèle proposé

Ce chapitre a pour but de décrire en détail le modèle de générateur automatique d'arbres de dérivation, que nous proposons afin d'afficher dynamiquement une trace d'exécution de programmes logiques. L'idée est que les futurs générateurs construits à partir de ce modèle puissent contribuer à améliorer la compréhension de la résolution de programmes logiques par les utilisateurs.

Pour commencer, nous allons procéder à la description des spécifications de l'application envisagée, nous expliquons les fonctionnalités qu'offrira cette application, ses modules ainsi que les éléments en entrées/sorties des modules. Par la suite, à travers les différents diagrammes UML, nous présentons plus en détail les différentes facettes de la phase de conception. Pour finir, une description globale de l'utilisation du modèle proposé sera présentée.

3.1 Spécification de l'application envisagée

La figure 6 montre un schéma des fonctionnalités de l'application envisagée. L'utilisateur interagit avec l'application par l'intermédiaire d'une interface d'entrée/sortie, il rentre le nom de la trace qu'il souhaite traiter, demande à l'application d'élaborer l'arbre de dérivation à travers le clique sur un bouton. Cette dernière lance le traitement qui va lire la trace en entrée, la convertir et en générer un arbre de dérivation qu'elle affichera ensuite dans le centre de l'interface. Par la suite l'application redonne la main à l'utilisateur pour interagir avec l'arbre généré. Cinq modules sont proposés pour réaliser les différentes tâches : traduction, traitement, exécution, affichage et interaction. Nous allons maintenant présenter les fonctionnalités de chaque module ainsi que les relations entre chacun d'eux.

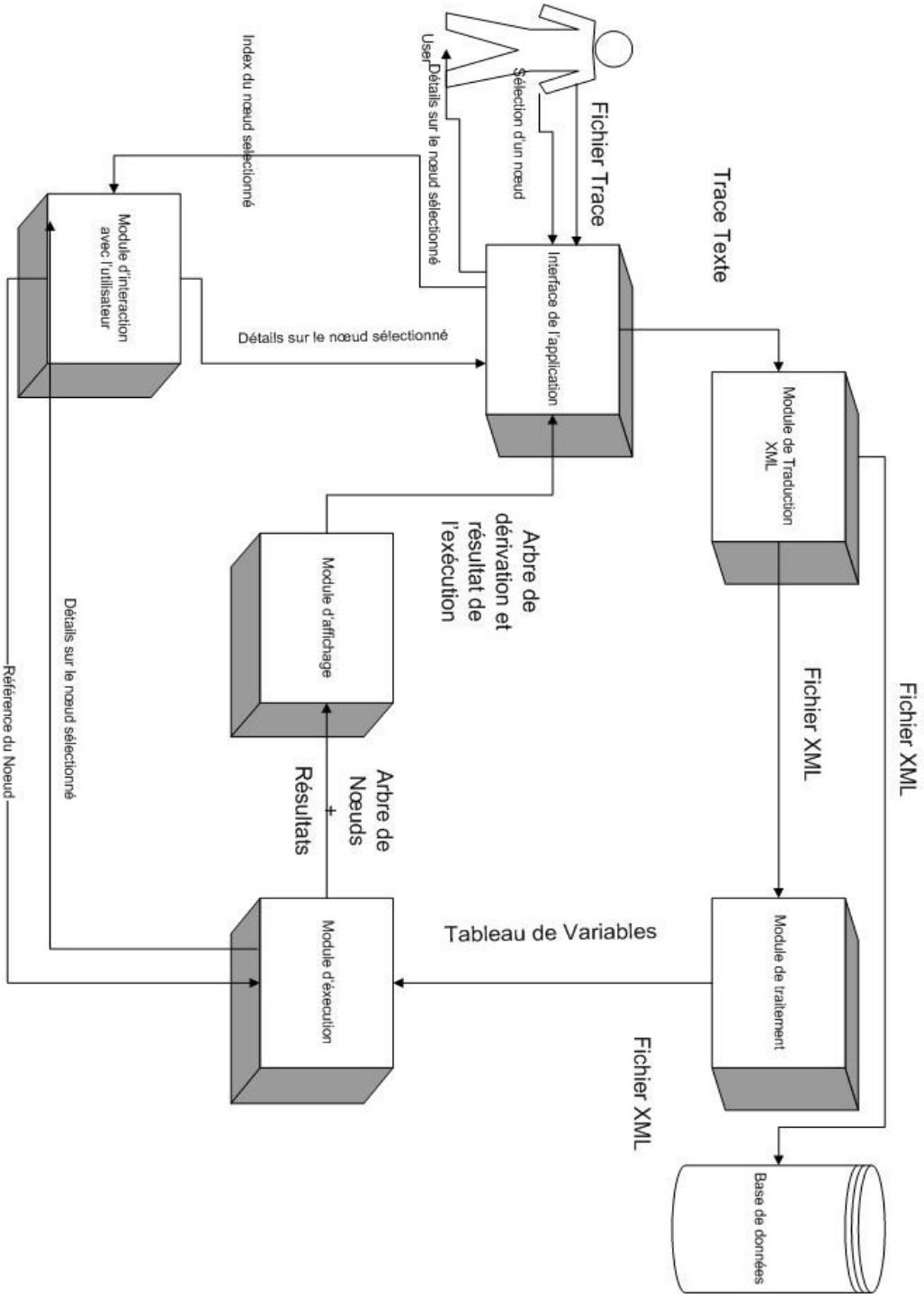


Figure 6 : Description globale de l'application envisagée

Pour enclencher le mécanisme d'exécution, l'utilisateur fournit à l'interface une trace d'exécution générée un langage de programmation logique, celle-ci ressemble à une liste d'itérations exécutées partant du but posé au programme et finissant par le résultat, donc la réponse au but. Nous pouvons voir un exemple de cette trace en utilisant l'appel « trace. » lors de l'exécution d'un programme en langage Prolog.

Une fois que la trace a été fournie à l'application, un module de traduction prend la relève afin de décomposer cette trace textuelle en différentes balises, qui composent en sortie un fichier XML bien structuré. Le choix du langage XML se base sur le fait que c'est un langage standard conçu pour l'échange automatisé de contenus complexes (arbres, textes riches, etc).Le fichier XML sera sauvegardé dans une base de données, pour de futures utilisations.

Un module de traitement prend ensuite la main pour extraire les données structurées du fichier XML et les classer dans deux tableaux distincts. Le premier tableau contient tous les faits et règles de la trace, soit les éléments du programme impliqués dans cette trace. Le deuxième tableau représente le complément du premier tableau en fournissant les liens entre chaque ligne, soit pour chaque entrée du tableau l'entrée appelante et les autres entrées appelées. Le traitement de ce module peut être considéré comme une étape de préparation des données pour le module d'exécution.

Le module d'exécution peut être considéré comme le module principal de l'application, ce dernier va parcourir les deux tableaux, que le module de traitement lui fournit en entrée, pour transformer chaque entrée du premier tableau en un nœud et construire les liens entre ces nœuds en utilisant le deuxième tableau. C'est enfin en réunissant ces nœuds que le module génère son objectif final : l'arbre de dérivation. En sortie, le module d'exécution fournit l'arbre de dérivation accompagné d'un tableau contenant l'ordre dans lequel les nœuds de ce dernier ont été appelés.

Une fois les données de sortie du module d'exécution reçues, le module d'affichage procède à la mise en place de l'affichage dynamique de l'arbre. Un par un et suivant l'ordre indiqué dans le tableau reçu en entrée, les éléments de l'arbre apparaissent laissant derrière eux une trace qui indique l'ordre d'apparition afin que l'on puisse le retracer même après l'affichage complet de l'arbre. Une fois l'affichage complété, l'utilisateur peut relancer l'animation autant de fois qu'il le souhaite.

Lorsque les résultats sont affichés et les arbres générés, l'application donne la main à l'utilisateur pour interagir avec les nœuds de l'arbre final. Si ce dernier souhaite consulter les détails concernant un des nœuds, il lui suffit de cliquer dessus. Le module d'interaction avec l'utilisateur prend en paramètre ce nœud, et consulte le module d'exécution qui, à travers le tableau fourni par le module de traitement, peut retourner la description du nœud en question. Une fois la description obtenue, les détails sont publiés à l'utilisateur dans l'interface de l'application à travers une bulle d'information.

En accédant à l'application envisagée, l'utilisateur peut aussi consulter d'anciens arbres de dérivation déjà générés auparavant. En effet, les fichiers XML, sauvegardés après l'étape de traitement à chaque fois que l'utilisateur entre une nouvelle trace, sont stockés dans une base de données. Cette base représente une bibliothèque accessible à tout moment. Il suffit de cliquer sur le nom du fichier XML dans la liste proposée par le générateur, pour que le traitement se lance automatiquement et génère l'arbre. L'application offre ensuite la main à l'utilisateur pour effectuer les mêmes opérations qu'il ferait avec un arbre nouvellement généré.

À partir des besoins de notre projet, détaillés dans les chapitres précédents, nous avons présenté les fonctionnalités de l'application envisagée. Nous expliquons maintenant la conception de cette application afin de bien spécifier notre modèle. Pour développer une application de qualité qui répond aux attentes et aux exigences, il est nécessaire d'adopter une méthode de conception. Pour cela, nous avons utilisé des diagrammes fournis par le langage de modélisation unifié UML afin de décrire les spécifications. Notre choix s'est porté vers le langage UML, car c'est un langage couramment utilisé dans les projets logiciels, pour spécifier, visualiser, modifier et construire les documents nécessaires au bon développement d'un logiciel orienté objet . UML est un langage qui grâce à ses différents types de diagrammes une grande souplesse permettant la modélisation de différents aspects de l'application [31].

Dans la section suivante, nous exposons les quatre diagrammes UML, que nous avons proposés pour décrire l'application envisagée.

3.2 Les diagrammes utilisés

Nous avons eu recours à différents diagrammes afin de représenter notre modèle sous tous ses aspects. Chaque type de diagramme fait le point sur une ou plusieurs caractéristiques de l'application envisagée. Ensemble, ces diagrammes représentent une vue complète de ce que nous désirons réaliser comme application. Dans cette section, nous présentons les diagrammes appliqués à notre application future, chacun accompagné d'une description.

3.2.1 Diagramme de cas d'utilisation

Les diagrammes de cas d'utilisation sont utilisés pour donner une vision globale du comportement fonctionnel d'un système logiciel (Wikipedia(6)). La figure 7 montre le diagramme de cas d'utilisation de l'application envisagée.

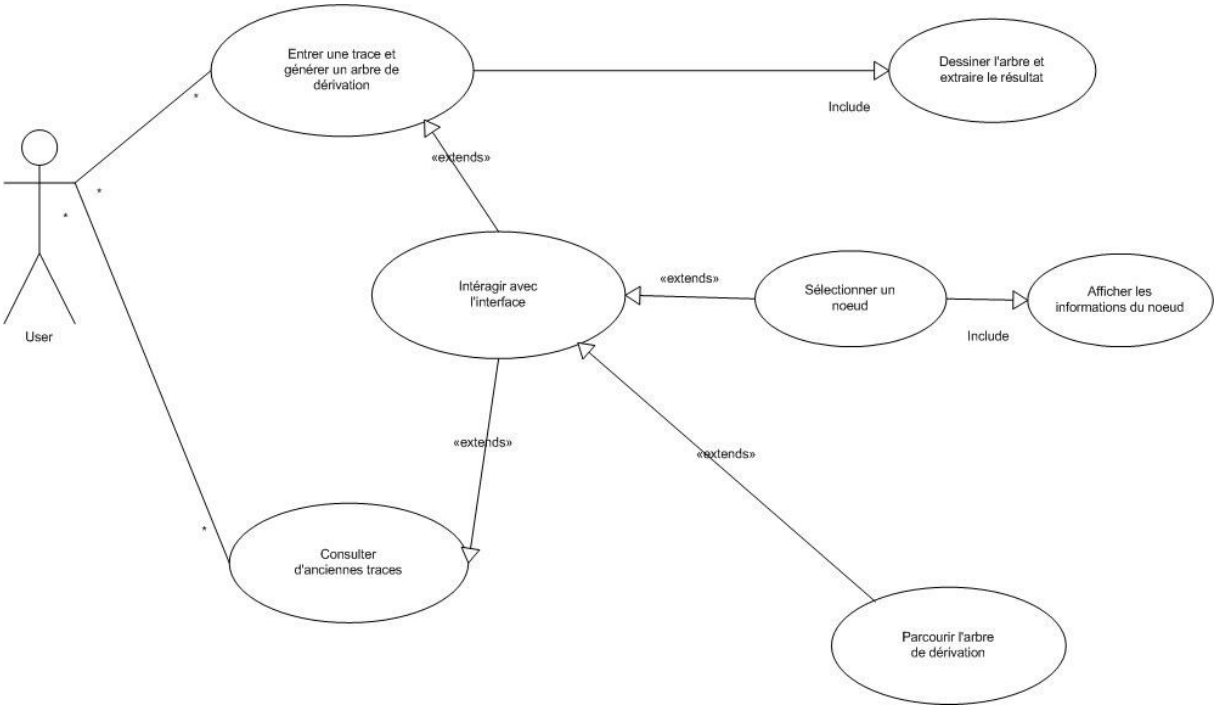


Figure 7 : Diagramme des cas d'utilisation du prototype

L'utilisateur à deux options, soit entrer une nouvelle trace dans l'application, soit consulter d'anciennes traces. Ces dernières, générées précédemment, ont été stockées dans une base de données et accessibles par les noms de fichier source. Lorsque la première option est choisie, l'arbre de dérivation est généré automatiquement et rendue accessible à l'affichage. Une fois que l'affichage d'un arbre de dérivation est disponible que ce soit par l'entrée d'une nouvelle trace ou la sélection d'une ancienne trace, l'utilisateur a la possibilité d'interagir avec cette arborescence en parcourant les différentes étapes de la génération de l'arbre ou en sélectionnant un des nœuds. Bien sûr, la sélection d'un nœud en particulier implique l'affichage des informations concernant ce nœud. Nous donnons en suivant la description des sept cas d'utilisation identifiés, soit en spécifiant les acteurs, les événements, les préconditions et les post-conditions de chacun des cas.

Cas d'utilisation 1 : Entrer une trace pour générer un arbre de dérivation

Acteur	**Utilisateur**
Événement déclencheur	**Bouton "Traiter" appuyé (ou cliqué)**
Pré-conditions	• **Nom du fichier contenant la trace à analyser** • **Nom du fichier pour sauvegarder la structure arborescente**
Post-conditions	**Trace transformée en structure arborescente**

Cas d'utilisation 2: Dessiner l'arbre / extraire le résultat

Acteur	**Système**
Événement déclencheur	**Bouton "Affichage de l'arbre" appuyé**
Pré-conditions	**Trace traitée**
Post-conditions	**Arbre dessiné au centre de l'interface**

Cas d'utilisation 3: Consulter d'anciennes traces

Acteur	**Utilisateur**
Événement déclencheur	**Ancienne trace de la base de données sélectionnée**
Pré-conditions	**Aucune**
Post-conditions	**Affichage des résultats et de l'arbre de dérivation de la trace sélectionnée**

Cas d'utilisation 4: Interagir avec l'interface

Acteur	**Utilisateur**
Événement déclencheur	**Aucun**
Pré-conditions	**Génération d'un nouvel arbre de dérivation ou la sélection d'un arbre appartenant à une ancienne trace**
Post-conditions	**Dépendant du type d'interaction :** **Cas d'utilisation 5 ou 6.**

Cas d'utilisation 5: Parcourir l'arbre de dérivation

Acteur	**Utilisateur**
Événement déclencheur	**Bouton "Suivant" ou bouton "Précédent" appuyé**
Pré-conditions	**Affichage de l'arbre dans l'interface**
Post-conditions	**Affichage de l'état suivant ou de l'état précédent de l'arbre**

Cas d'utilisation 6: Sélectionner un nœud

Acteur	**Utilisateur**
Événement déclencheur	**Un des nœuds de l'arbre sélectionné**
Pré-conditions	**Affichage de l'arbre dans l'interface**
Post-conditions	**Recherche des données spécifiques au nœud sélectionné**

Cas d'utilisation 7: Afficher les informations du nœud

Acteur	**Système**
Événement déclencheur	**Aucun**
Pré-conditions	**Nœud sélectionné**
Post-conditions	**Affichage des données spécifiques au nœud sélectionné**

À travers les différents cas d'utilisation, nous avons pu parcourir les fonctionnalités de l'application future et ses interactions avec l'utilisateur ainsi que les conditions qui rendent ces interactions possibles. Nous allons maintenant passer à une description plus profonde de l'application, celle de l'organisation des classes et packages qui composent cette dernière.

3.2.2 Diagramme Inter-package

Les packages sont considérés comme une méthode d'organisation efficace qui vise à regrouper différents types de classe ayant un rapport quelconque dans un même paquet. Les packages forment une représentation globale du projet. Wikipédia(4).

La figure 8 représente le diagramme inter-package de l'application envisagée. Elle se compose de trois principaux packages qui sont l'interface, l'interpréteur de trace et l'arborescence. Chacun de ces packages est expliqué en suivant ainsi que les liens qui les relient.

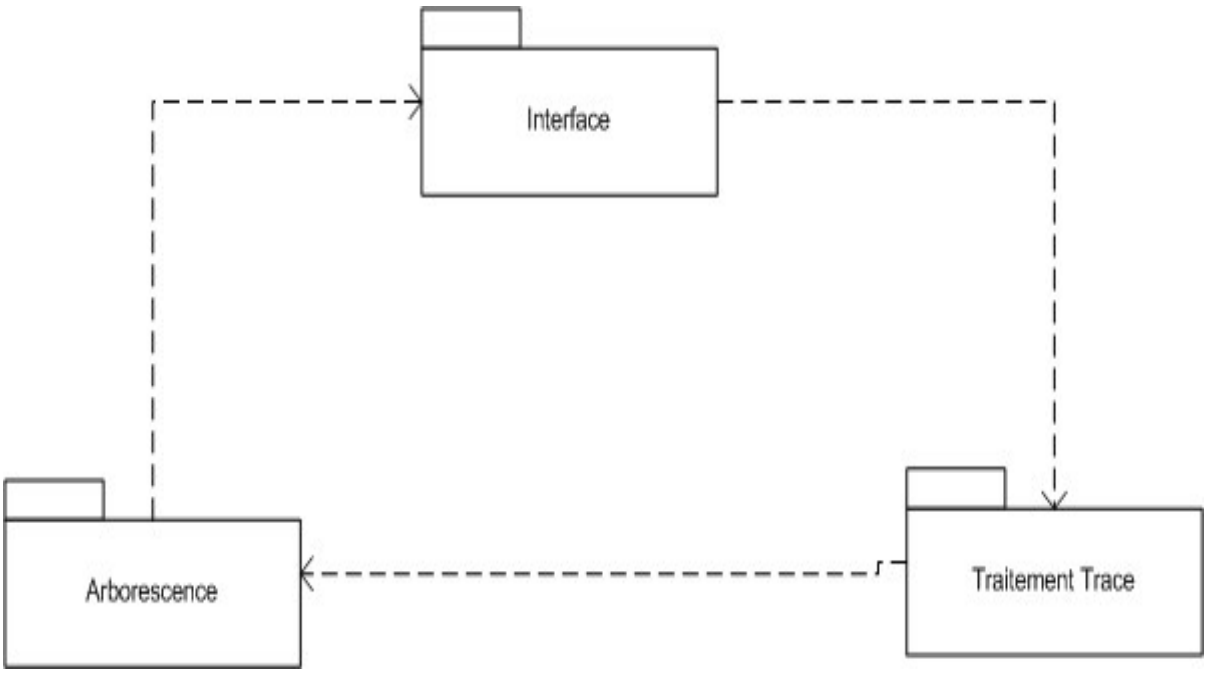


Figure 8 : Diagramme inter-package de l'application

Package 1 : Interpréteur de trace

Ce package peut être décrit comme le cœur de l'application, il permet la conversion de la trace en entrée en un fichier XML. Ensuite, il analyse ce dernier pour en extraire une liste de nœuds qui sera utilisée dans le package suivant pour la génération de l'arbre.

Package 2 : Arborescence

La génération de notre résultat final passe par ce package. il récupère en entrée la liste de nœuds créée par le premier package et utilise les classes Arbre, Nœud, et résultat pour construire nœud par nœud les différentes étapes par lesquelles passe le programme pour résoudre le problème. À la fin du traitement, le package transfère les différents arbres au troisième package afin de les afficher.

Package 3 : Interface

Ce package représente le lien entre l'utilisateur et l'application. À l'ouverture de l'application et par l'intermédiaire de ce package, l'utilisateur entre sa trace, qui sera ensuite dirigée vers le package « interpréteur de trace ».

Le package Interface reçoit en entrée, du package Interpréteur de trace, les différents éléments de l'arbre représentants les étapes d'exécution. Sa tâche principale est de les afficher à l'utilisateur.

Ce package donne aussi la main à l'utilisateur pour sélectionner d'anciennes traces que ce dernier voudrait consulter. Lorsque la trace à consulter a été sélectionnée, l'interface affiche les arbres et les résultats correspondants à cette dernière.

Après avoir fait le tour des différents packages de notre prototype, nous allons dans la section suivante présenter plus en détail le contenue de ces packages : les classes qui les composent et qui représentent les entités principales du projet.

3.2.3 Diagramme de Classes

Le diagramme de classes est un schéma utilisé en génie logiciel pour présenter les classes et les interfaces des systèmes ainsi que les différentes relations entre celles-ci. Ce diagramme fait partie de la partie statique d'UML, car il fait abstraction des aspects temporels et dynamiques. Wikipédia(5).

Comme nous l'avons précisé précédemment, notre projet comporte trois packages où chacun contient plusieurs diagrammes. Cependant pour plus de clarté et une meilleure facilité de description, nous avons choisi de représenter les différentes classes dans le même diagramme à la place de trois diagrammes distincts, Ce schéma est représenté par la figure 9.

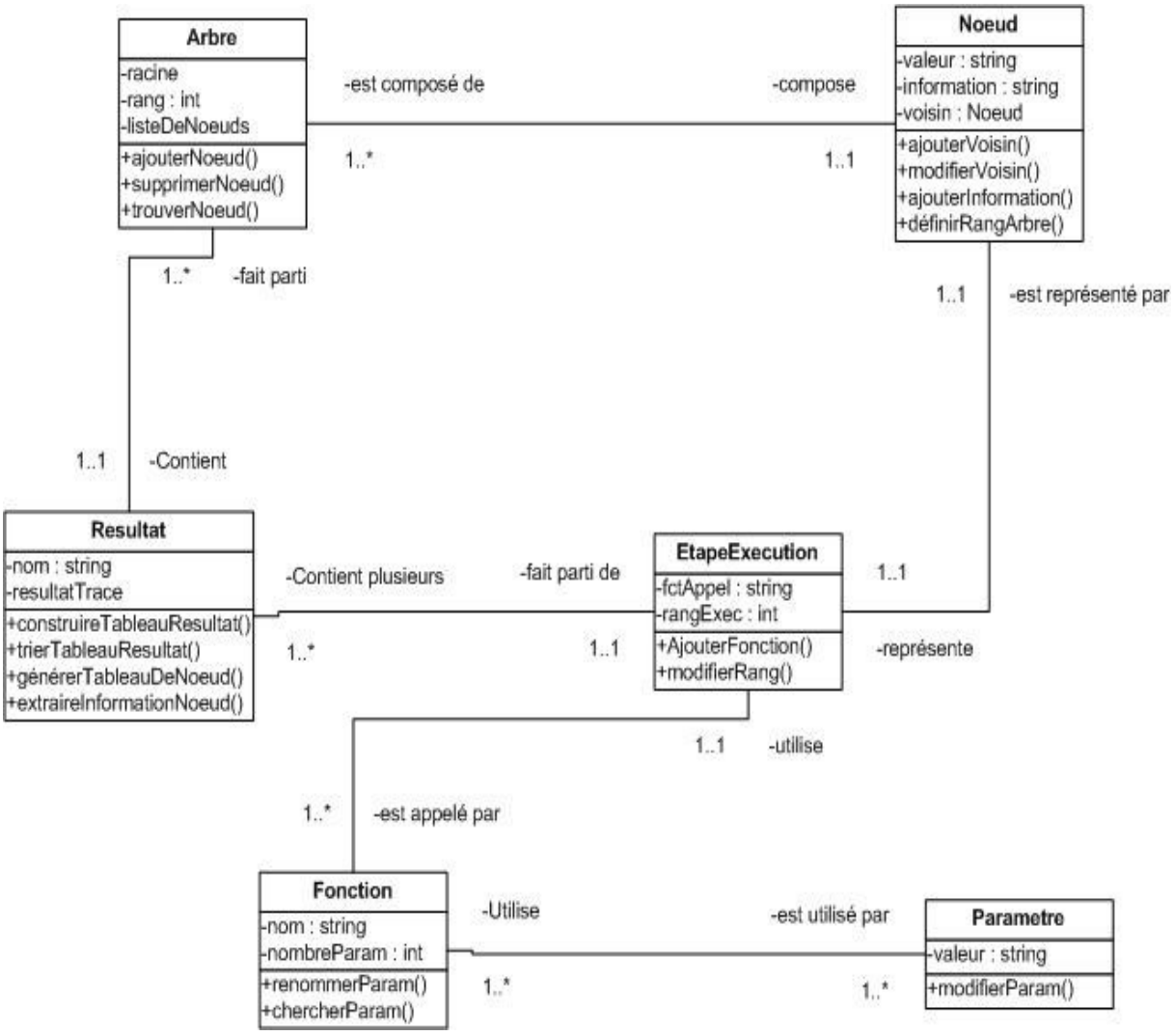


Figure 9 : Diagramme de classe du prototype

Chaque classe est définie distinctement garce à ces attributs et fonctions qui lui sont propres :

Arbre : Cette classe représente un arbre de dérivation représenté par un « rang », une « racine » et une « listeDeNœuds ». Le Rang (entier) permet de connaitre l'étape d'exécution dans l'arbre. Il aura la valeur 0 au début de n'importe quelle exécution, et la valeur maximum quand l'arbre atteint le résultat final. La racine (Qui est un pointeur sur une instance de la classe Nœud) représente quant à elle le point de départ de l'arbre. La liste de nœuds (Liste d'entiers) contient tous les nœuds composants l'arbre courant.

Nœud : La classe Nœud est directement liée à la classe Arbre qui est une composition de nœuds. Les champs représentants cette classe sont « valeur », soit ce que va contenir le nœud dans une chaine de caractères, « Information », des informations textuelles qui décrivent le nœud, et « voisins », une liste d'objets de type de nœud contenant le ou les nœuds avoisinant le nœud en question.

Fonction : Cette classe représente une fonction qui sera exécutée pendant le traitement. Elle est décrite par deux champs : « nom », le nom de la fonction sous forme d'une chaine de caractères, et « nombreParm », le nombre de paramètres qu'elle utilise pour s'exécuter, donc un entier.

Paramètre : C'est la classe qui représente les paramètres utilisés pour l'exécution d'une fonction, elle a pour seul champ « valeur », soit une chaine de caractères.

ÉtapeExécution : EtapeExécution représente la classe principale de notre diagramme, car une trace est composée d'une liste d'étapes exécutées afin d'arriver à la solution. Cette classe est donc en lien avec la plupart des autres classes. Les champs composants cette classe sont « fctAppel »(une chaine de caracteres) et « rangExec » (un entier). Le champ fctAppel représente la fonction d'appel de l'étape d'exécution, il peut avoir comme valeurs : appel, refait, sortie, échec. Le champ rangExec représente le rang d'exécution de l'étape.

Résultat : Comme son nom l'indique, cette classe représente les résultats qui seront générés à la sortie de chaque exécution. Son champ « nom », une chaine de caractères, représente le nom de la trace. Ce nom sera utile pour faciliter la recherche des résultats stockés lorsque l'utilisateur souhaitera consulter d'anciennes traces. Le champ « résultatsTrace » contient une liste d'arborescences qui représentent les différentes étapes de l'exécution.

Les différentes classes sont reliées entres elles par des liens à travers les quelles se schématise notre prototype :

Arbre (1-*)-Nœud (1-1) : Un arbre est composé de plusieurs nœuds alors qu'un nœud ne peut appartenir qu'à un seul arbre pour dans une même exécution de l'application.

Arbre (1-*)-Résultat (1-1) : Un arbre de dérivation peut fournir un ou plusieurs résultats en sortie dépendamment des données du programme, alors qu'un résultat ne peut correspondre qu'à un seul arbre dans une même exécution de l'application.

EtapeExécution (1-1) –Nœud (1-1) : Une étape d'exécution est représentée par un nœud dans l'arbre, donc à chaque étape d'exécution correspond un nœud et vice versa.

EtapeExécution (1-1)-Résultat (1-*): Une étape d'exécution peut représenter un seul résultat mais, dans certains programmes, nous pouvons avoir plusieurs résultats, sous forme de plusieurs étapes d'exécution.

EtapeExécution (1..1)-Fonction (1-*) : Une étape d'exécution fait appel à une fonction maximum à la fois, alors qu'une même fonction peut être appelée dans plusieurs étapes pendant l'exécution d'un programme.

Fonction (1-*)-Paramètre (1-*) : Une fonction peut avoir besoin de un ou plusieurs paramètres pour s'exécuter, tout comme un paramètre peut être utilisé par une ou plusieurs fonctions pendant l'exécution d'un programme.

Après avoir détaillé les différentes classes qui composent notre modèle de prototype et les packages contenants ces classes, La prochaine section servira à présenter les différents états par les quels passe une trace d'exécution lors de son traitement dans notre futur application.

3.2.4 Diagramme d'état de transition

Ce diagramme sert à représenter des automates d'états finis, sous forme de graphes d'états, reliés par des arcs orientés qui décrivent les transitions. Les diagrammes d'états de transitions permettent de décrire les changements d'états d'un objet ou d'un composant, en réponse aux interactions avec d'autres objets/composants ou avec des acteurs.Wikipédia(3).

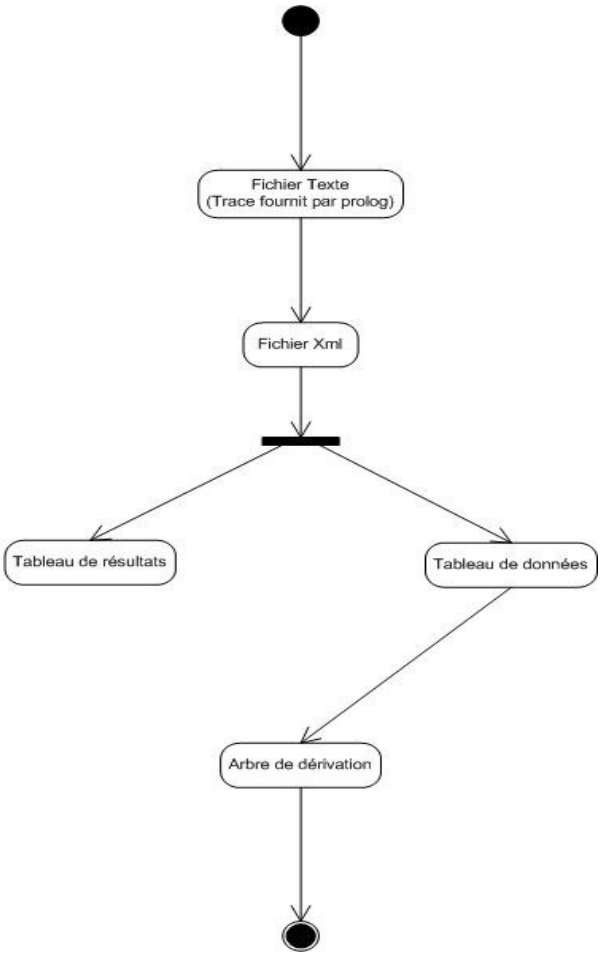


Figure 10 : Diagramme d'états de transitions de l'application envisagée

Le diagramme d'états de transitions décrit dans la figure précédente démontre que le cycle de transformation d'une trace passe par quatre phases distinctes :

La première phase du cycle de vie de notre trace consiste en la récupération de cette dernière à partir du logiciel de programmation sous la forme d'un fichier texte.

La deuxième phase sert à récupérer le fichier texte généré juste avant pour ensuite le convertir en un fichier XML, une extension qui offre beaucoup plus de flexibilité.

Dans la troisième phase et à partir du fichier XML de la phase précédente, la trace sera représentée sous la forme de deux tableaux. Un tableau contient le ou les résultats de l'exécution et un deuxième pour toutes les données qui composent la trace en entrée.

Dans la quatrième phase, finalement, le deuxième tableau est utilisé afin de générer l'arbre de dérivation qui constitue l'état final et l'objectif de l'application envisagée.

En conclusion, comme pour toute modélisation dans un projet, notre principal but était de faciliter le travail pour la partie concrète du projet qui est la réalisation. En même temps, nous avons bien expliqué notre vision du modèle de l'application envisagée. Nous avons atteint l'objectif de ce chapitre qui était de modéliser notre solution grâce aux outils de modélisation sélectionnés, notamment avec les différents diagrammes du langage UML.

Un diagramme de cas d'utilisation a été très utile afin de décrire les différentes taches que pourra réaliser l'utilisateur dans l'application envisagée. Ensuite, la paire de diagrammes inter-package et de classes ont permis d'offrir une bonne visibilité détaillée de la structure, grâce à la présentation en premier lieu des packages, puis des classes et enfin des liens entres toutes ces entités. Enfin, à travers un diagramme d'états de transition, nous avons détaillé les différents états par lesquels passe notre principale entité : la trace.

Suite à cette étude conceptuelle, qui a permis d'éclaircir les détails de la solution choisie, nous allons maintenant le prototype que nous avons réalisé à partir de ce modèle.

Chapitre 4 : Réalisation du prototype

Dans le chapitre précédent, nous avons spécifié un modèle d'application pour aider les utilisateurs à comprendre la résolution de programmes logiques. À partir de ce modèle, nous avons construit un prototype que nous avons testé et validé afin de montrer la faisabilité de la solution proposée. Dans ce chapitre, nous décrivons ce prototype en présentant d'abord l'environnement logiciel utilisé pour le réaliser. Ensuite, nous expliquons les fonctionnalités réellement élaborées. Cette description est faite en partie à l'aide de captures d'écran de l'interface du prototype réalisé. Nous concluons ce chapitre par une analyse des résultats obtenus avec la solution proposée.

4.1 Environnement logiciel

Pour l'implémentation de notre prototype, nous avons eu recours aux divers logiciels et langages. Nous avons utilisé, comme langage et environnement de programmation, le langage Java. Ce langage nous a semblé le plus adéquat car il se caractérise par sa portabilité et est souvent utilisé pour la programmation orientée-objet.

Comme langage de balisage, nous avons utilisé le langage XML. Ce langage, à balises étendues ou extensibles, permet de définir des balises pour décrire un document textuel. Il est considéré comme un métalangage pouvant définir d'autres langages. Comme nous l'avons indiqué dans notre modélisation, il est utilisé pour mettre en forme la trace d'exécution en utilisant de nouvelles balises spécifiques à notre contexte.

Nous avons également utilisé l'environnement de développement intégré, NetBeans, logiciel libre offert par Sun à partir de juin 2000. En plus de Java, l'environnement NetBeans permet également de supporter d'autres langages, comme Python, C, C++, JavaScript, XML, Ruby, PHP et HTML. Il comprend toutes les caractéristiques d'un environnement de développement intégré moderne (éditeur en couleurs, projets multi-langages, réusinage de codes, éditeur graphique d'interfaces et de pages Web).

Tout ces langages et environnements ajoutés à un modèle bien conçu, nous ont permis ce concevoir un prototype répondent à la plus part de nos attentes et remplissant l'objectif principal de notre prototype.

La section à suivre contient une description générale de ce que nous avons conçu, un résumé simplifié de l'utilisation de notre application et de son fonctionnement interne et externe.

4.2 Description générale du prototype conçu

Nous avons conçu un prototype capable de récupérer en entrée un fichier texte contenant la trace générée par l'exécution d'un programme Prolog. Dans un premier temps et à travers un module de traduction, le prototype traduit le fichier texte en format XML et génère le fichier correspondant. Ce fichier est enregistré en mémoire. Dans un deuxième temps, le module de traitement extrait du fichier XML les informations requises pour la construction d'un arbre de dérivation. Grâce à son module d'exécution, notre prototype construit l'arbre attendu. Pour chaque étape d'exécution du programme, un sous-arbre est généré. La dernière étape génère l'arbre final représentant la trace complète. Cependant, nous gardons en mémoire tous les sous-arbres générés car, ensemble, ils représentent une trace complète de la construction de l'arbre final. En donnant la possibilité à l'utilisateur de les parcourir, il pourra suivre pas à pas la logique derrière le résultat de l'exécution.

Les données de la trace et les arbres générés sont ensuite tous envoyés au module d'affichage. Ce module commence par afficher sous forme de tableau une liste ordonnée des fonctions utilisées et de leurs paramètres pour la résolution de la question posée au programme Prolog. Puis, à la demande de l'utilisateur, il affiche le sous-arbre initial. Il laisse la main pour offrir la possibilité d'afficher un à un les sous-arbres suivants jusqu'au dernier.

Les fichiers XML générés au début du traitement sont gardés dans la mémoire interne du prototype. Leur sauvegarde est très utile pour la suite du traitement, car elle facilite la gestion des données, en plus de d'offrir une possibilité de réutilisation assez efficace. Un exemple de la structure de ces fichiers est présenté dans la figure 11.

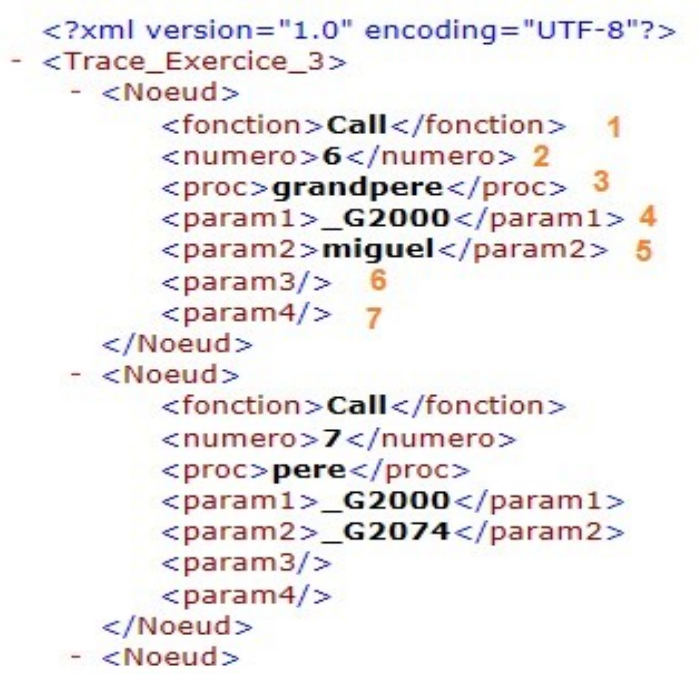


Figure 11 : Structure XML du prototype

Rappelons que le fichier XML est généré à partir de la trace textuelle qui est fournie à l'entrée par l'utilisateur. Ce fichier se compose de sept principales balises XML autour desquelles se fait la transformation de la trace en un arbre de dérivation.

La première balise« fonction» contient la fonction avec laquelle le langage Prolog résout un problème ou un but. Elle correspond à l'action exécutée à une étape donnée. Le langage Prolog utilise 4 fonctions qui sont : Call (appel), Fail (échec), Redo (refaire), Exit (terminé).

La deuxième balise « numero » indique le numéro de l'étape d'exécution en cours.

La troisième balise « proc » représente le but ou sous-but exécuté.

Enfin, les quatre autres balises « param1, param2, param3 et param4 » représentent les paramètres du but ou sous-but exécuté. Dans le cas où les quatre paramètres ne sont pas tous nécessaires, ceux qui ne le sont pas prennent la valeur « Null ».

Après une description générale de notre prototype et de la logique derrière son fonctionnement, nous allons dans la partie suivante faire description en profondeur de l'interface de notre application à travers des captures d'écrans de celle-ci, accompagnées de descriptions de toutes ces composantes.

4.3 Description des fonctionnalités

Dans cette partie, nous présentons un ensemble de captures d'écran montrant les différentes fonctionnalités de notre prototype. Pour commencer, nous présentons une vue globale de l'interface du prototype (voir figure 12).

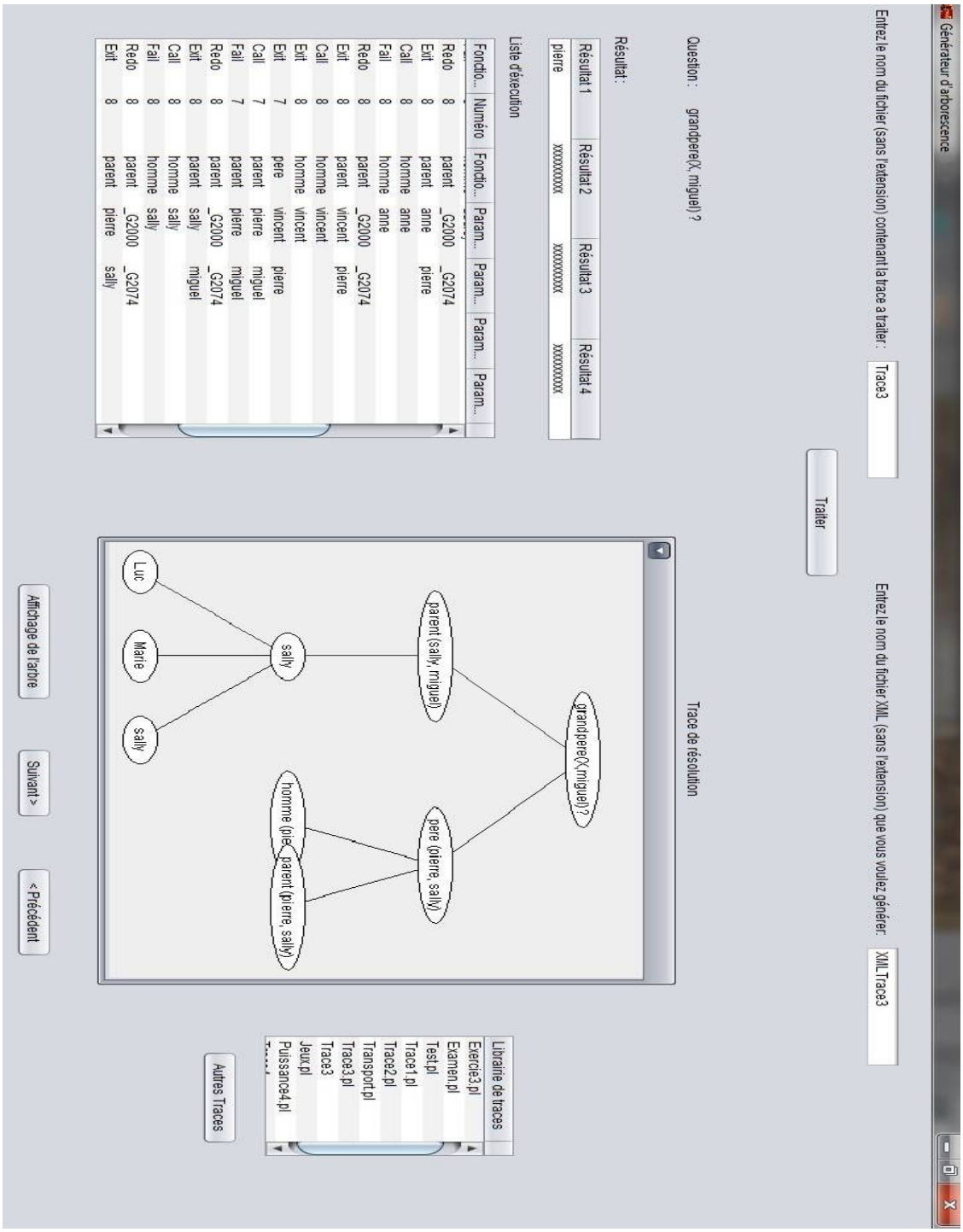


Figure 12 : Interface de l'application

Pour mieux présenter cette interface, nous avons choisi de la décomposer en quatre parties distinctes qui sont l'entête, le résultat (sous forme de liste des opérations exécutées), la trace d'exécution ainsi que la bibliothèque des traces. Chacune des parties est détaillée sous la figure qui la représente.

4.3.1 L'entête de l'interface

L'entête de l'interface (figure 13) représente la porte d'entrée du prototype. En effet, afin de pouvoir générer le résultat d'une trace, il faut avant tout entrer le nom du fichier texte contenant la trace et ceci dans le champ indiqué (en haut à gauche de l'interface). L'utilisateur n'a pas besoin d'indiquer l'extension du fichier, le prototype se charge de le traiter comme un fichier ".txt ".

Figure 13 : Entête de l'interface

Afin de pouvoir garder en mémoire et localiser le fichier XML généré par l'application, un deuxième champ de saisi est offert à l'utilisateur pour qu'il puisse nommer lui-même ce fichier. Comme pour le fichier en entrée, l'utilisateur n'a pas besoin de mentionner l'extension du fichier, l'application se chargera de l'enregistrer sous la forme ".xml".

Lorsque les noms de fichiers ont été saisis, l'utilisateur clique sur le bouton 

pour lancer le traitement de la trace. Le but de ce traitement consiste en l'affichage du résultat, celui du tableau de nœuds et la préparation de l'arbre de dérivation.

4.3.2 Le résultat et la liste d'exécution

La figure 14 montre plus précisément cette partie de l'interface. Elle se décompose elle-même en trois sous-parties qui sont la question ou le but à résoudre, point de départ de la trace, le résultat obtenu suite à la résolution du but et la liste des étapes de cette résolution.

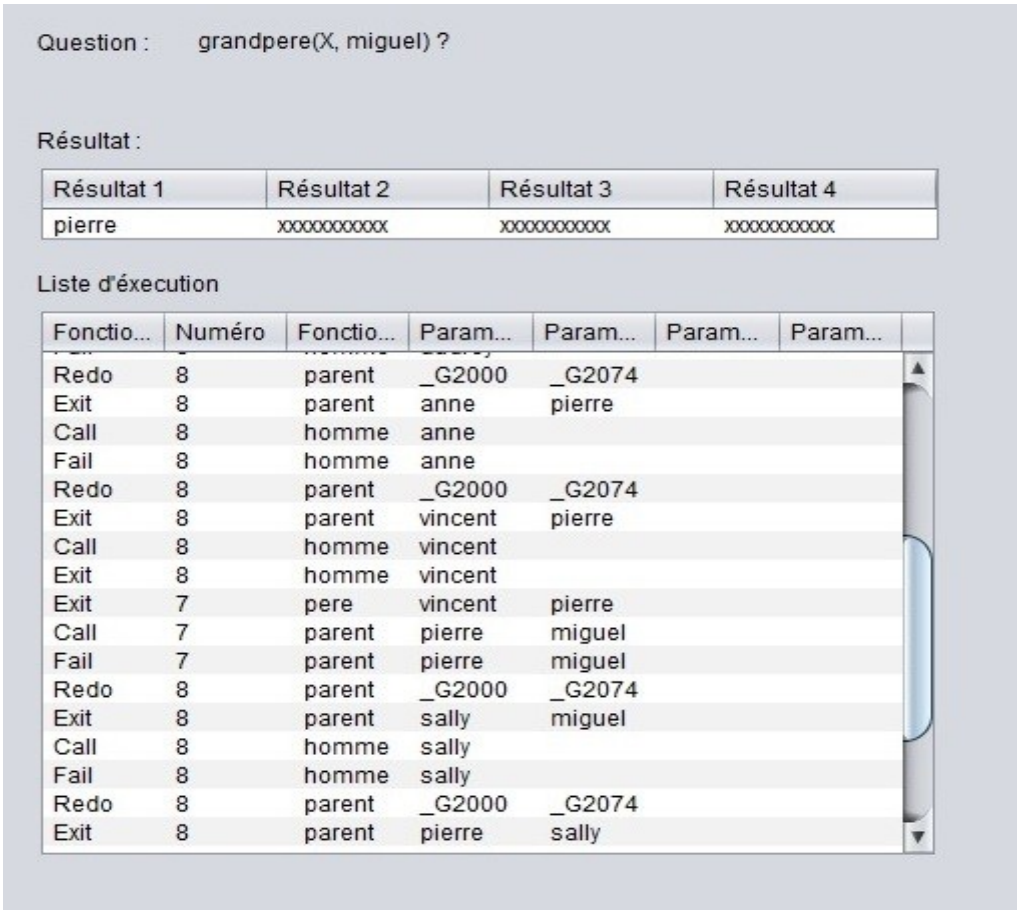
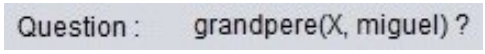


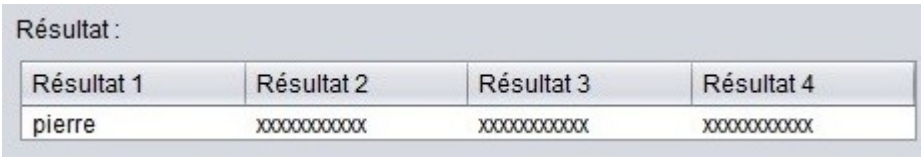
Figure 14 : Résultats et liste d'exécution

- **La question :**



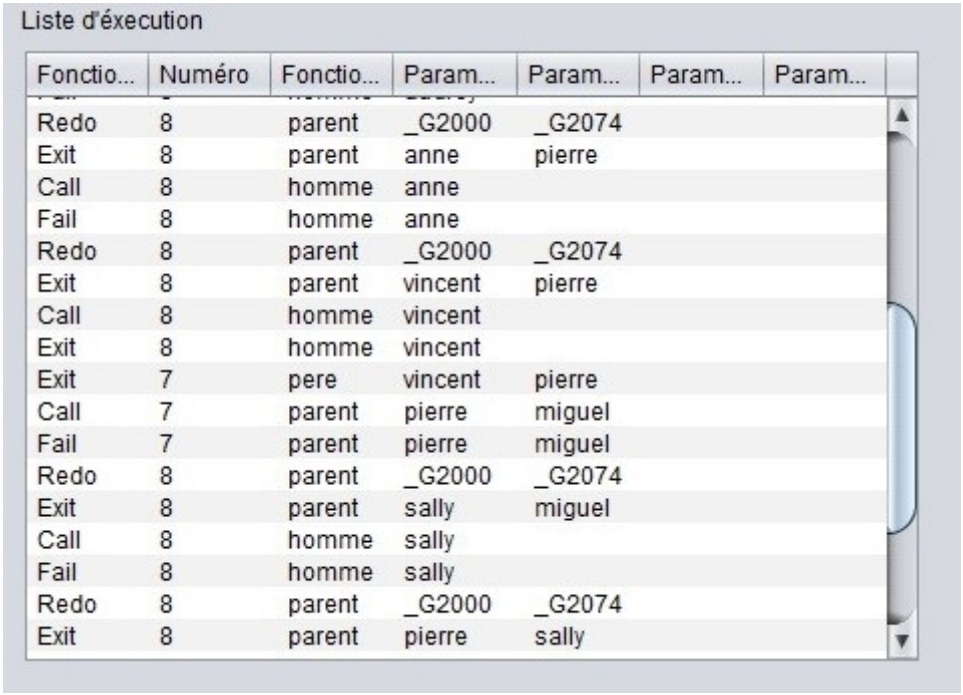
Cette section affiche à chaque fois la question ou la requête posée par l'utilisateur à Prolog qui a permis de générer la trace.

- **Le résultat de la résolution :**



Cette section contient un tableau contenant les résultats fournis par la résolution Prolog. Une résolution Prolog peut générer en sortie un ou plusieurs résultats, soit vrai ou faux ou bien les valeurs des paramètres de la question. Le tableau est donc rempli en fonction de ce nombre et accepte jusqu'à quatre résultats. Lorsqu'un champ résultat n'est pas utilisé alors il prend la valeur "xxxxxxxxxxxxx".

La liste d'exécution



La troisième et dernière sous-partie de l'interface représente, ligne par ligne, les étapes exécutées jusqu'à la solution du problème. La différence que présente ce tableau par rapport à une trace fournie en Prolog lui-même est que chaque élément et sous la forme d'une balise XML et non pas une chaine de caractère simple, ce qui facilite la décomposition de la trace et la transformation de chaque ligne en un nœud pour l'arbre de dérivation et augmente la lisibilité de la trace.

Les différentes colonnes de ce tableau reprennent la structure du fichier XML généré au début du traitement. Chaque colonne représente une des balises du fichier, nous avons donc cinq colonnes obligatoires « Fonction Prolog », « Numéro », « Fonction/procédure appelée », « paramètre 1 » et « paramètre 2 ».

- « Fonction Prolog » : Il représente la fonction avec laquelle le langage Prolog exécute la ligne qui suit. Ce champ peut prendre seulement quatre valeurs possibles : « Call », « Redo », « Fail » et « Exit ».
- « Numéro » : Ce champ indique à quel niveau ou enfant de l'arbre s'exécute cette étape.
- « Fonction/procédure » : Il représente le but ou sous-but exécuté dans cette étape de la résolution.
- « Paramètre 1/ Paramètre 2 » : Ils représentent les deux premiers paramètres avec lesquels s'exécute le but ou sous-but courant. Si les paramètres ne sont pas utilisés, la cellule du tableau correspondante est laissée vide.

Si un but ou sous-but a besoin de plus de deux paramètres pour s'exécuter, deux autres colonnes facultatives ont été rajoutées afin de répondre aux besoins de ce type de situations.

La figure 15 représente un exemple d'arbre de dérivation que notre prototype a pour but de générer. Une fois que l'utilisateur a défini les noms des fichiers en entrée et en sortie et cliqué sur le bouton Traiter, l'interface lui donne la possibilité de générer l'arbre correspondant et l'afficher en cliquant sur le bouton Affichage de l'arbre.

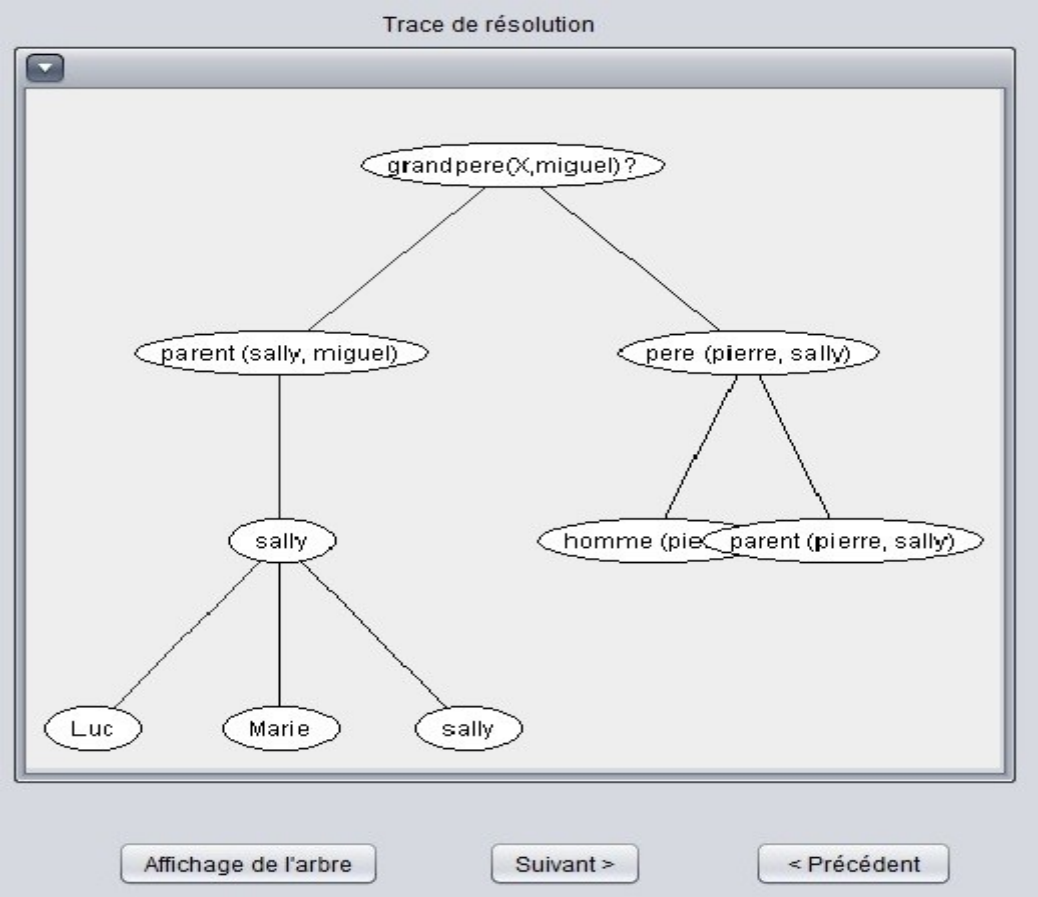


Figure 15 : L'arbre de dérivation

En dessous de cette arborescence, notre prototype offre deux boutons Suivant > et < Précédent qui permettent de parcourir les différents états de l'arbre de dérivation, afin de suivre son cheminement dès la création du premier nœud jusqu'à la construction de l'arbre complet.

L'utilisateur a la possibilité de naviguer entre les différentes étapes de construction de l'arbre. Il peut suivre pas à pas la résolution du problème et même revenir en arrière pour choisir d'aller vers un nœud différent de celui choisi précédemment. Contrairement à la simple lecture d'une trace, la description graphique sous forme arborescente permet de voir les ancêtres et les enfants de chaque nœud, donc de connaitre le prédicat qui a été utilisé et ceux qui seront utilisés.

La figure 16 représente un exemple de cheminement de génération et d'affichage d'un arbre de dérivation, en utilisant le bouton Suivant >.

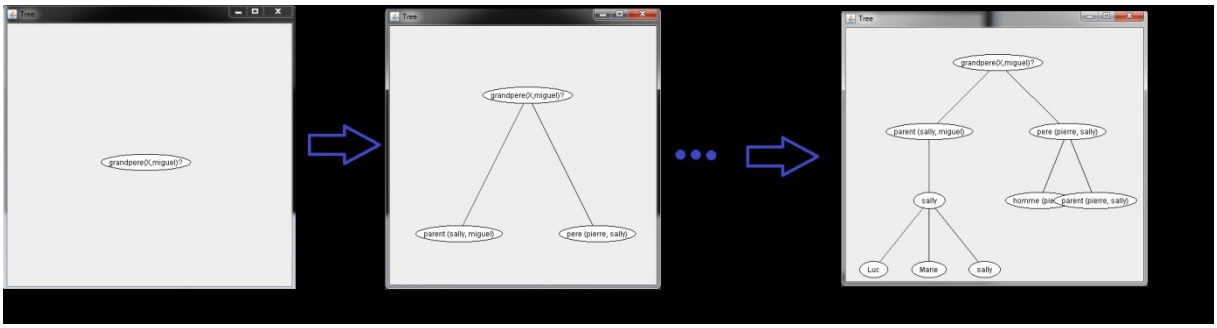


Figure 16 : Évolution d'une dérivation

La première copie d'écran, à gauche, correspond à la racine de l'arbre. La deuwième, au centre, représente la première dérivation effectuée. La dernière montre une autre étape de la dérivation. Il est aussi possible de revenir en arrière donc de droite à gauche, grâce au bouton < Précédent.

4.3.3 La Bibliothèque de traces

La dernière sous-partie de notre interface représente la liste des traces qui ont été exécutées auparavant. Chaque fois que l'utilisateur donne un nouveau fichier trace en entrée, en générant le fichier XML, le prototype fait aussi une copie de la trace dans sa bibliothèque. Nous pouvons voir apparaitre le nom du document dans la liste appelée « Librairie de traces » à l'extrême droite de l'interface principale (figure 14).

Un bouton 

a été ajouté pour pouvoir consulter d'anciennes traces n'apparaissant pas dans cette liste, mais accessibles dans un autre répertoire

Après beaucoup de travail, nous pouvons conclure que nous avons réussi à générer un prototype affichant l'ensemble des éléments nécessaires à notre contexte. L'interface proposée est complète, simple à utiliser et interactive, comme nous allons l'expliquer en suivant.

4.4 Discussion

Au départ de notre projet de recherche, nous nous sommes fixé comme but de modéliser, programmer et donc concevoir une application à visée éducative qui serait fonctionnelle, fiable et facile à utiliser. En d'autres termes, nous souhaitions développer une application qui permettrait à son utilisateur de pouvoir suivre en temps réel le déroulement d'une exécution d'un programme en Prolog, à travers la transformation d'une trace textuelle en un arbre de dérivation. Le but d'une telle application était de faire mieux comprendre le fonctionnement de la programmation logique.

Tout au long de la réalisation de notre projet, nous avons rencontré certaines difficultés qui nous ont montré que l'implémentation d'une telle application ne serait pas si facile. Nous avons cependant réussi à surpasser ou contourner la plupart de ces difficultés. Nous analysons dans cette section ce que nous avons réussi à accomplir, ce qu'il reste dans notre liste de tâches à réaliser, et finalement les avantages et inconvénients de notre solution.

4.4.1 Discussion des résultats

Nous avons proposé un modèle d'outil graphique pour représenter les traces d'exécution de programmes logiques d'une manière dynamique, ceci en vue d'aider à comprendre ce type de résolution. Nous avons construit un prototype à partir de ce modèle afin d'en montrer la faisabilité. Ce prototype dispose de différents modules pour représenter graphiquement et dynamiquement des traces d'exécution de programmes écrits en langage Prolog. Un premier module prend en entrée une trace d'exécution d'un programme Prolog fournie par l'utilisateur sous forme d'un fichier texte. Un deuxième module convertit ce fichier texte en une représentation graphique, soit un arbre de dérivation, et en mémorise les différentes étapes. Ensuite, un dernier module qui permet d'afficher cet arbre en offrant la possibilité à l'utilisateur de suivre pas à pas sa construction et donc le déroulement d'exécution de son programme Prolog.

En permettant à l'utilisateur de naviguer entre les différentes étapes de la construction de l'arbre, nous ajoutons un aspect très important à notre prototype comparé aux travaux semblables. L'interaction entre l'outil et l'utilisateur est un aspect qui peut jouer un rôle très important pour faciliter la compréhension de l'exécution des programmes logiques traités car l'utilisateur peut découvrir cette représentation à son rythme.

Même si l'objectif principal de notre projet a été atteint, il nous reste quand même quelques aspects à améliorer afin que notre prototype soit le plus performant possible pour son utilisateur. Trois principaux points restent à traiter dans notre liste :

- ✓ une meilleure représentation de l'arborescence sous forme d'animation : au départ de notre projet, nous voulions mettre en place un module qui générerait la représentation de l'arbre de dérivation en sortie sous la forme d'une animation. Ceci permettrait de visualiser en boucle le déroulement de la résolution sans avoir à cliquer sur les boutons « suivant » et « précédent »; 

pour parcourir les différents états de l'arbre. En effet, ces deux boutons pourraient être remplacés par un bouton « Départ » qui lancerait l'animation une fois la trace traitée et un autre bouton « Stop » qui arrêterait l'animation quand le désire l'utilisateur, afin d'avoir une image fixe sur une des étapes de l'exécution.

- ✓ Interaction directe avec les nœuds : c'est la tâche principale parmi celles qui reste à accomplir. En effet, le fait d'avoir généré les arborescences sous forme d'images vectorielles simples ne nous a pas permis de pouvoir donner à l'utilisateur la possibilité de cliquer sur un des nœuds pour en savoir plus sur son contenu. C'est une étape importante de notre projet que nous pourrions corriger en utilisant un logiciel de dessin graphique plus performent. Il faudrait en effet localiser l'emplacement (le nœud) sur lequel l'utilisateur a cliqué afin de lui fournir les données adéquates qui aideraient à comprendre pourquoi la résolution a suivi telle ou telle étape.

  Une autre solution serait de remplacer la fenêtre contenant l'image de l'arbre par une grille de nœuds. À chaque arbre généré, les nœuds prendraient l'état "Visible" et pourraient être reliés par des droites. Le reste des nœuds serait "Invisible", ce qui permettrait à l'utilisateur de ne voir que l'arborescence voulue à chaque fois.

- ✓ Mise en place d'une interaction avec une base de données : à ce jour, notre application permet à un utilisateur de consulter à la demande d'anciennes traces déjà générées auparavant dans une bibliothèque de la mémoire interne. Les fichiers XML sont sauvegardés à chaque exécution d'une nouvelle trace. Afin d'avoir plus d'espace mémoire et de faciliter la gestion de nos données gardées en mémoire, nous pourrions mettre en place un module qui se chargerait de la pertinence des données et communiquerait directement avec un serveur de base de données MySql,Apache ou encore Oracle.

4.4.2 Avantages et limites

Après avoir décrit l'état présent de notre prototype et celui qu'il pourrait prendre dans le futur grâce à quelques améliorations, nous allons ici discuter des avantages, inconvénients et améliorations possibles de la modélisation proposée.

En tant qu'application à visée éducative, notre modèle présente quelques avantages importants qui permettent d'envisager une application plutôt complète et agréable à utiliser. Le modèle permet d'offrir à l'utilisateur une interface assez fiable et très facile à utiliser grâce à un nombre d'éléments réduits mais efficaces. Cette efficacité est basée sur des boutons qui décrivent bien leurs fonctionnalités, des sections bien définies et une structure assez claire des composantes de l'interface. La disposition de l'interface fait en sorte de mettre en valeur le résultat principal de l'application (l'arborescence) et offre une grande facilité à interagir avec ce dernier, une liste d'étapes parcourus accompagne l'arborescence, ainsi son utilisateur peut suivre pas à pas le déroulement de l'exécution. Tous ces outils font que notre application envisagée soit idéal pour faciliter la compréhension de l'utilisateur.

Un deuxième avantage qu'offre notre modèle consiste en la persistance des données. En effet, notre modèle garde en mémoire les arbres de dérivation générés dans le passé par l'utilisateur lui permettant de pouvoir les consulter quand il le souhaite. Ainsi il n'aura pas besoin de redonner en entrée une trace déjà exécutée dans le passé afin de voir le résultat.

Un autre avantage est que en générant un fichier XML à partir de la trace texte fournie par l'utilisateur, et en sauvegardent une copie de ce dernier en mémoire, notre modèle offre à l'utilisateur l'accès à ce fichier pour d'éventuelles autres utilisations. La structuration des données sous forme XML est très utilisée dans le domaine de l'informatique, elle offre à son détenteur de multiples possibilités de traitement de l'information.

Un dernier avantage assez important dans notre prototype consiste dans le fait que les traces pouvant être utilisées en entrée peuvent provenir que de différente sources, donc l'utilisateur n'est pas obligé d'utiliser un langage de programmation logique en particulier. Il est possible de générer des arbres de dérivation à partir de trace venant d'autres langages de programmation logique comme Python, OZ…

Et même si la structure des traces générées par un langage ou un autre venait à changer avec l'évolution des logiciels qui les exécutent, une légère mise à jour du module de traduction suffira à adapter au nouveau format.

Comme tout prototype, le notre montre aussi quelques limites; La première est plus importante réside dans le manque d'informations explicatives pour chaque nœud distinctement, il aurait été intéressant d'avoir une bulle

d'information contenant des explications comme « pourquoi le choix de ce paramètre? » ou encore « pourquoi l'appel à cette fonction et pas une autre? » qui apparaitrai quand l'utilisateur clique sur l'un des nœuds de l'arbre.

Une autre limite est que l'utilisateur ne peut consulter la représentation graphique de l'arbre résultant en dehors de l'application, c'est-à-dire qu'il lui est impossible d'imprimer, modifier ou utiliser l'arbre de dérivation dans une autre application.

Ainsi, il serait possible d'améliorer le modèle proposé en ajoutant un processus pour sauvegarder la représentation graphique de l'arbre dans un format standard comme JPEG ou GIF.

Conclusion générale

Dans ce mémoire de maitrise, nous avons présenté un projet de recherche concernant le domaine de l'intelligence artificielle et plus précisément, la programmation logique.

Ce domaine est tellement vaste qu'il touche directement ou indirectement à presque tous les aspects de notre vie quotidienne. C'est aussi un domaine qui ne cesse de s'agrandir et dans lequel des découvertes et des avancées spectaculaires sont réalisées de jour en jour. Nos recherches nous ont permis de constater qu'il est impossible de tout cerner de ce milieu, tellement les retombées sont immenses, mais que plus on en apprend plus on se passionne pour ce domaine d'étude jeune et révolutionnaire.

Nos recherches portaient en particulier sur les origines même de ce jeune domaine qui est l'intelligence artificielle, ses impacts dans notre monde actuel et les éventuelles avancées technologiques qu'il nous amènera dans le futur. Ensuite, nous nous sommes tourné vers la programmation logique utilisé en arrière de cette intelligence artificielle. Nous avons exploré les différents langages utilisés pour ce type de programmation et en avons effectué une comparaison. Celle-ci nous a permis de découvrir à quel point ces langages peuvent être différents et efficaces quand ils sont utilisés pour des projets adéquats à leurs caractéristiques.

En avançant dans nos recherches, nous avons constaté que la problématique dans ce type de programmation était que la logique de fonctionnement et de résolution des algorithmes est assez délicate à assimiler et que le simple fait de faire exécuter des programme logique et voir le résultat en sortie ne suffisait généralement pas a comprendre comment le compilateur est arrivé à ce résultats. Nous nous somme donc fixé comme objectif de modéliser un prototype d'application qui à travers une représentation graphique de l'exécution d'un programme logique faciliterai la compréhension de la résolution des programme logique.

Notre modélisation représentait une application qui prend en entrée une trace générée par les compilateurs de programme logique, et à travers une série de traitements et de transformations elle offrait à l'utilisateur une description graphique sous forme d'une image animée qui parcourait l'ensemble des étapes de résolution du

programme sous la forme d'un arbre de dérivation. Le graphique était accompagné d'une liste des ces étapes parcourus pour permettre à l'utilisateur de suivre le cheminement jusqu'au résultat.

Nous avons ensuite procédé à la mise en place d'un prototype qui avait pour but de valider la faisabilité du modèle décrit précédemment. L'utilisateur fournit un fichier texte contenant la trace d'une exécution d'un programme écrit en langage Prolog. Après traitement, cette trace est affichée en une suite d'arbres de dérivation qui décrivent chaque étape d'exécution de la trace donnée en entrée. Ainsi l'utilisateur peut parcourir les différents arbres dans l'interface tout en faisant le lien avec chaque ligne de la trace et comprendre à quoi correspond chaque opération. Notre prototype garde aussi en mémoire les traces déjà exécutées afin de donner la possibilité à l'utilisateur de les consulter en tout temps. Bien qu'il soit fonctionnel, ce prototype pourrait être amélioré par l'utilisation d'une animation et l'affichage d'informations relatives à chaque nœud lorsque l'utilisateur clique sur les nœuds.

Le modèle proposé pourrait permettre de développer des outils qui serait intégrés aux différents environnements de programmation logique, qui à la place de la simple génération du résultat sous forme textuelle offrirait une image animée représentant l'arbre de dérivation qui décrit le déroulement de l'exécution menant à ce résultat. De plus, de tels outils offriraient une description textuelle de chaque nœud de l'arbre représentant un nouveau but à résoudre, la description graphique serait alors elle-même décrite. Ceci dit, pour pouvoir développer de tels outils, il faudrait ajouter un module de traduction automatique de la trace en format XML. Pour l'instant, nous n'avons pas proposé de traducteur indépendant du langage de programmation utilisé, ni même de traducteur spécifique pour tous les langages existants. Enfin, comme nous l'avons souligné dans ce mémoire, il serait intéressant dans une prochaine étape de vérifier l'efficacité d'un tel outil pour aider à comprendre la résolution de programmes logiques en recueillant l'avis d'utilisateurs.

Bibliographie

[1] Équipe de Recherche En Ingénierie des ConnaissAnEes

http://www.ift.ulaval.ca/~ericae/

[2] B. G. Buchanan, *A (Very) Brief History of Artificial Intelligence*, AI Magazine 26(4):53-60, 2005.

[3] D. Defays, R. M. French et J. Sougné. *Apports de l'Intelligence Artificielle à la Psychologie.* Faculté de Psychologie et des sciences de l'éducation. 1997.

[4] D. Lenat et R. V. Guha. *Building Large Knowledge-Based Systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, US 1989.

[5] Les robots industriels. 05 jan. 2013.

http://archive.wikiwix.com/cache/?url=http://www.gralon.net/articles/materiel-et-consommables/materiels-industriels/article-les-robots-industriels-285.htm&title=%C2%AB%C2%A0Les%20robots%20industriels%C2%A0%C2%BB

[6] TPE sur l'intelligence artificielle. Consulté le 03 mai. 2012

https://sites.google.com/site/int3llig3nc3artifici3ll3/

[7] P. Lévine et J.C. Pomerol. Systèmes interactifs d'aide à la décision et systèmes experts. Hermes, 1989.

[8] Système-expert. Consulté le 03 mai. 2012

http://www.hypergeo.eu/spip.php?article84

[9] R. Bartak. *Constraint Programming*. First Edition. 1998.

[10] K.C. Santosh, B. Lamiroy et J. P. Ropers: *Inductive Logic Programming for Symbol Recognition,* Iniria, Version1, 2009.

[11] R. Quiniou, E. Fromont, F. Portet, Reconnaissance d'arythmie cardiaque. Consulté le 23 mars. 2012

https://interstices.info/jcms/c_16718/reconnaissance-darythmies-cardiaques

[12] G. Laurent, A. Tassadit et H. Olivier, *Interaction, Connaissances et Langage Naturel,* LERIA. Consulté le 23 mars. 2012

http://www.info.univ-angers.fr/leria/icln.php

[13] SWI Prolog. Consulté le 27 mars. 2012

http://www.swi-prolog.org/

[14] Logtalk. Consulté le 20 mars. 2012

http://logtalk.org/

[15] P. Graham, *ANSI common LISP*, Prentice Hal, 1995.

[16] NASA CLIPS RULE-BASED LANGUGE. Consulté le 20 mars. 2012

http://www.siliconvalleyone.com/founder/clips/index.htm

[17] Introduction au système CLIPS. Consulté le 20 mars. 2012

http://www-lium.univ-lemans.fr/~lehuen/master2/webclips/etape1/index.html

[18] The Mozart Programming System. Consulté le 23 mars. 2012

http://mozart.github.io/

[19] Python Programming Language – Official Website. Consulté le 27 mars. 2012

http://www.python.org/

[20] Mercury. Consulté le 20 mars. 2012

http://www.mercury.csse.unimelb.edu.au/

[21] HOPE. Consulté le 07 mars. 2012

http://www.soi.city.ac.uk/~ross/Hope/

[22] Haskell Programming Language. Consulté le 27 mars. 2012

http://www.haskell.org/haskellwiki/Haskell

[23] R. K. Dybvig. *The Scheme Programming Language*, Third Edition, 2003.

[24] Racket. Consulté le 20 mars. 2012

http://racket-lang.org/

[25] O'Caml. Consulté le 23 mars. 2012

http://ocaml.org/description.html[26] A. O. Bilska, K. H. Leider, M. Procopiuc, O. Procopiuc, S. H. Rodger, J. R. Salemme et E. Tsang, *A Collection of Tools for Making Automata Theory and Formal Languages Come Alive* **,**Duke University, Durham, NC.

[27] P. Lopez, *LTAG Workbench: A General Framework for LTAG*, Saarbrcken, Allemange, 2000.

[28] LLP2 : Loria LTAG Parser 2. 18 oct. 2012

http://www.loria.fr/~azim/LLP2/help/fr/

[29] F. Barthélemy, P. Boullier, P. Deschamp, L. Kaouane et É. Villemonte, *Atelier ATOLL pour les grammaires d'arbres adjoints*, Tours, France, 2001.

[30] A. O. Bilska, K. H. Leider, M. Procopiuc, O. Procopiuc, S. H. Rodger, J. R. Salemme et E. Tsang, *A Collection of Tools for Making Automata Theory and Formal Languages Come Alive*, Duke University, Durham, NC, 1994.

[31] Clocksin, F. William, Mellish, Christopher S. *Programming in Prolog*. Berlin ; New York: Springer-Verlag, 2003.

[32] Covington, R. Bagnara; O'Keefe, J. Wielemaker et S. Price, *Coding guidelines for Prolog*, 2010.

[33] F. Benhanou, D. McAllester et P. Van Hentenryck, *Revisited ii, proceedings of ILPS'94*, International Logic Programming Symposium, Rapport **de** recherche, No.94-**2,** Ithaca, USA, MIT Press 1994.

[34] A. Colmerauer, *An introduction to Prolog-III ii*, communications of the ACM, July 1990

[35] P. Gloor, *AACE - Algorithm Animation for Computer Science Education*, IEEE Workshop on Visual Languages, 1992.

[36] M. Procopiuc, O. Procopiuc, et S. Rodger, *Visualizationand Interaction in the Computer Science Formal Languages Course with JFLAP,* Frontiers in Education Conference, Salt Lake City, Utah, 1996.

[37] P. Bonhomme et P. Lopez. *TagML: XML encoding of resources for lexicalized tree adjoining grammars*, In Proceedings of LREC 2000, Athens, 2000.

[38] P. Lopez*. LTAG workbench: A general framework for LTAG.* In Proceedings of the International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+5), Paris, May 2000.

[39] A. Boyd, M. Dickinson et D. Meurers. *Increasing the recall of corpus annotation error detection.* In Proceedings of the Sixth Workshop on Treebanks and Linguistic Theories, Bergen, Norway, 2007.

[40] A.K. Joshi et Y. Schabes., *Tree-adjoining grammars*, *Handbook of Formal Languages, Volume 3: Beyond Words*, Springer, New York 1997.

[41] J Hernandez-Orallo and D. L. Dowe, *"Measuring Universal Intelligence: Towards an Anytime Intelligence Test*, *Artificial Intelligence Journal* **174**, 2010.

[42] TPE : l'intelligence artificielle. 05 jan. 2013.

https://sites.google.com/site/tpelintelligenceartificielle/la-reconnaissance-facial

[43] E. Albert, M. Hanus F. Huch, J. Oliver et G. Vidal: *Operational Semantics for Declarative Multi-Paradigm Languages. Journal of Symbolic Computation* 40, 2005.

[44] S. Antoy et M. Hanus: *Set Functions for Functional Logic Programming. In: Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming,* ACM Press, 2009.

[45] B. Brassel, M. Hanus, F. Huch, J. Silva et G. Vidal: *Run-Time Profiling of Functional Logic Programs* LNCS, vol. 3573, Springer, Heidelberg, 2005.

[46] C. Ferri-Ramírez, J. Hernández-Orallo et M.J. Ramírez-Quintana: *Incremental Learning of Functional Logic Programs*. FLOPS 2001, LNCS, vol. 2024, Springer, Heidelberg 2001.

[47] S. Fischer: A *Functional Logic Database Library*. In: Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming, ACM Press, 2005.

[48] A.J. Fernández, M.T. Hortalá-González, F. Sáenz-Pérez et R. del Vado-Vírseda*: Constraint Functional Logic Programming over Finite Domains*. Theory and Practice of Logic Programming 7, 2007.

[49] J. Straub et J. Huber: *A Characterization of the Utility of Using Artificial Intelligence to Test Two Artificial Intelligence Systems*. Computers. 2013.

[50] J. Straub et J. Huber: *A Characterization of the Utility of Using Artificial Intelligence to Test Two Artificial Intelligence Systems*. Computers. 2013.

[51] D. Defays, R. M. French et J. Sougné, *Apports de l'Intelligence Artificielle à la Psychologie*, Maison d'édition, 1997.

[52] J. Legrand, *Le langage Prolog : Exemple en Turbo Prolog,* Edition TECHNIP, 1992.

[53] G. Palski, *La naissance de la democartographie analyse historique et semiologique,* Université de Paris 1.

[54] J.C.Pouget, *Présentation du langage objets Java*, Mission au Cemagref, Lyon, 1998.

[55] C.Baral et M.Gelfond, *Logic programming and knowledge representation Journal of Logic Programming* , Vol. 19, 1994.

[56] M. Alpuente, F.J. Correa, M. Falaschi: *A Debugging Scheme for Functional Logic Programs. Electronic Notes in Theoretical Computer Science*, vol. 64, 2002

[57] P. Blackburn, J. Bos et K. Streignitz, *PROLOG tout de suite*, College Publications, août 2007.

[58] J. Legrand, *Le langage Prolog - Exemples en Turbo Prolog*, Edition Technip, 1992.

Wikipédia

Wikipédia(1), Histoire de l'intelligence artificielle. 12 mars. 2013.

http://fr.Wikipédia.org/wiki/Histoire_de_l'intelligence_artificielle

Wikipédia(2) Intelligence artificielle. 12 mars. 2013.

http://fr.Wikipédia.org/wiki/Intelligence_artificielle

Wikipédia(3), Diagramme états-transition. 12 mars. 2013.

http://fr.wikipedia.org/wiki/Diagramme_%C3%A9tats-transitions

Wikipédia(4), Diagramme des paquetages. 12 mars. 2013.

http://fr.wikipedia.org/wiki/Diagramme_des_paquetages

Wikipédia(5), Diagramme de classes. 12 mars. 2013.

http://fr.Wikipédia.org/wiki/Diagramme_de_classes

Wikipédia(6), Diagramme de cas d'utilisation. 12 mars. 2013.

http://fr.Wikipédia.org/wiki/Diagramme_des_cas_d'utilisation