Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

وزارة التعليم العالي و البحث العلمي

BADJI MOKHTAR-ANNABA UNIVERSITY
UNIVERSITY BADJI MOKHTAR-ANNABA

جـامعـة بـاجـي مخـتـار- عـنـابـة

Faculté des Sciences de l'Ingéniorat
Département d'Informatique

Année : 2014/2015

THÈSE

Présentée en vue de l'obtention du diplôme de

Doctorat en science

Intitulée :

Un EIAH pour l'algorithmique

Filière : **Informatique**
Spécialité : **Génie Logiciel**

Par
Ismail BOUACHA

Devant le Jury

Pr Mohamed Tahar KIMOUR	Professeur à l'Université d'Annaba	Président
Pr Tahar BENSEBAA	Professeur à l'Université d'Annaba	Directeur de Thèse
Pr Balla Amar	Professeur à l'ESI d'Alger	Examinateur
Dr Yacine LAFIFI	MCA à l'Université de Guelma	Examinateur
Pr Mahmoud BOUFAIDA	Professeur à l'Université de Constantine	Examinateur

***A** tous ceux qui sont dans mon coeur*.

Remerciements

Mes premiers remerciements vont à Tahar Bensebaa, le directeur de ce travail. Je suis heureux d'avoir eu la chance durant ces dernières années de partager ses réflexions, de disposer de son expertise scientifique et de son expérience pour mener à bien ce travail.

Je tiens aussi à remercier Boufaida Mahmoud, professeur à l'université de Constantine, Balla Amar, professeur à l'ESI d'Alger, et Lafifi Yacine, MCA à l'université de Guelma, de m'avoir fait l'honneur d'examiner cette thèse.

Je tiens aussi à remercier Kimour Mohamed Tahar, professeur à l'université Badji Mokhtar Annaba, de m'avoir fait l'honneur de présider cette thèse.

Je remercie également les membres du laboratoire LIG, en particulier Denis Bouhineau et Cyrille Demoulins, Maîtres de conférences à l'Université de Joseph Fourier, avec lesquelles j'ai pu avoir de nombreux échanges.

Je tiens aussi à remercier le personnel de l'école préparatoire aux sciences et techniques, et aussi à la société Citrus Innovation.

Enfin, je souhaite associer à ces remerciements ma famille, mes amis, mes collègues et tous ceux que je connais de près ou de loin.

ملخص

نتناول في هذه المذكرة إحدى المحاور الرئيسية في EIAH *(البحث في كيفية استغلال بيئة الحوسبة في تعليم الإنسان) ، المتمثل في تقييم المتعلم. مع التطورات التكنولوجية يمكن للتقييم أن يتم بمساعدة الكمبيوتر .وعليه فإن التقييم الآلي يمكن أن يساعد المدرسين على خلق نماذج تعليمية، وبالتالي تركيز جهودهم الأساسية على التعليم و إلقاء الدروس فقط.*

و يعتبر تقييم المتعلمين او المتكونين بشكل عادل وصحيح تحد حقيقي، حيث أن الأساليب الموجودة غير كافية ولا تتأقلم مع مادة الخوارزميات (Algorithmique) ,كما أن إستخدام مترجم حقيقي (Compilateur)، رغم أنه يسمح بتنفيذ البرامج و تجريبها، إلا أنه لا يمكننا من الحصول على صورة مفصلة عن كفاءات المتعلم. وبالتالي لا يمكننا تحديد الأخطاء و الصعاب التي يواجهها، ومساعدته على تجاوزها.

في إطار تطويرEIAH لتعليم الخوارزميات، هدفنا الأساسي يمثل في تسهيل مهمة التقييم و جعلها الية. لهذا اقترحنا أداة لفهم الخوارزميات تمكن المتعلمين من كتابة الخوارزميات المقترحة والمعلمين من بناء نماذج من المقترحات الممكنة، ويمكن لهذه النماذج أن تكون صحيحة أو غير صحيحة لكنها ذات أهمية بيداغوغية. عملية الفهم تتمثل في مقابلة المقترحات مع النماذج الموجودة. هذه العملية لا تقتصر على مجرد المقارنة، بل تتعدى ذلك إلى إستخراج أقصى قدر من المعلومات من الخوارزمية مثل : نوايا المتعلم وطريقة تفكيره و عيوب الخوارزمية المقترحة. إذا لم يتم إيجاد نموذج يوافق الخوارزمية المقترحة، يتم التعرف عليها من طرف المعلم، الذي يظيفها إلى قاعدة النماذج إذا رأى أنها مهمة. بهذه الكيفية نضمن التعرف على جميع المقترحات. الطريقة التي نقترحها تشجع إبداع المتعلمين، حيث أنها تسمح لهم بالتعبير عن كفاءاتهم الخوارزمية.

كلمات مفاتيح : التقييم، المتعلم، التقييم بمساعدة الكمبيوتر، الخوارزمية، فهم البرامج.

Résumé

Ce manuscrit aborde l'un des axes principaux de recherche dans les EIAH (Environnement Informatique pour l'Apprentissage Humain), qui est l'évaluation de l'apprenant. Avec l'évolution technologique, l'évaluation peut être assistée par ordinateur. Cette automatisation de l'évaluation peut aider les enseignants à créer des modèles éducatifs, par conséquent de concentrer leurs efforts sur l'enseignement.

L'évaluation des apprenants représente un véritable défi. En effet, les méthodes d'évaluation jusqu'ici utilisées dans les EIAH (QCM ou Questions à Choix Multiples, Questions à trous) sont inadaptées à une discipline telle que l'algorithmique. D'un autre coté, si l'utilisation d'un véritable compilateur, même pour du pseudo code, permet d'éditer des algorithmes et de les tester, elle ne permet pas, en revanche, d'avoir un feedback détaillé sur les connaissances et compétences des apprenants. Par conséquent, on ne peut pas ni localiser les lacunes et difficultés qu'un apprenant peut avoir, ni y remédier.

Dans le contexte d'un EIAH dédié à l'algorithmique, notre objectif est d'éliminer cette tâche d'évaluation, voire la réduire. Pour cela, nous avons proposé un outil de compréhension des algorithmes, qui permet aux apprenants d'écrire des algorithmes propositions et aux enseignants de construire des modèles de propositions. Ces modèles peuvent être corrects ou incorrects, mais ayant une utilité pédagogique. La compréhension consiste à faire correspondre l'algorithme proposé avec les modèles existants. Elle ne se limite pas à une simple comparaison, mais plutôt elle consiste à extraire le maximum d'informations à partir du code de l'algorithme telles que : les intentions de l'apprenant, la défaillance de la proposition, etc. Ainsi, notre évaluation sera plus pertinente, puisqu'en plus de la note, on aura plus de détails sur les apprenants. Dans le cas où une proposition n'a pas de modèle correspondant, l'expert humain peut l'évaluer et si nécessaire enrichir la base de modèle. Par conséquent, l'évaluation est garantie. Notre approche favorise la créativité des apprenants, en leur offrant la possibilité d'engager réellement leurs compétences.

Mots clés : EIAH, Evaluation, Apprenant, Evaluation assistée par ordinateur, Algorithmique, Algorithme, QCM, Compréhension de programmes.

Abstract

This manuscript treats one of the most important researches in TEL, which is the assessment of the learner. With technological developments, the evaluation can be computerized. This automation of the assessment can help teachers to create educational models, thus concentrating their efforts on teaching.

The assessment of learners is a challenge. Indeed, the assessment methods so far used in TEL (MCQ, Questions with holes) are inadequate with a discipline such as algorithmic. On the other hand, if the use of a real compiler, even for pseudo-code, allows us to edit and test algorithms, it does not allow, however, to have a detailed feedback on the knowledge and skills of learners. Therefore, we cannot locate learner's gaps and difficulties or remedy them.

In the context of an algorithmic dedicated TEL, our goal is to facilitate this assessment task or eliminate it. For this, we proposed a tool for understanding algorithms, which allows learners to write algorithms and for teachers to build models of solution. These models can be correct or incorrect, but having an educational purpose. Understanding is matching the proposed algorithm with existing models. It is not limited to a simple comparison, but rather it consists on extracting the maximum information from the code of the algorithm such as the intentions of the learner, the failure of the solution, etc.

Thus, Assessment will be more relevant, because in addition to a mark, there will be more details on learner's solution. When a solution has no corresponding model, the human expert can evaluate it and if necessary expanded a base of models. Therefore, the evaluation is guaranteed. Our approach promotes creativity of learners, offering them the opportunity to really engage their skills.

Keywords: TEL, Evaluation, Assessment, Learner, Algorithmic, Algorithm, MCQ, Program Comprehension

Table des Matières

Table des Illustrations

Liste des Figures

Liste des Tableaux

Chapitre 1

Introduction générale

« Si l'on avait la science, on aurait le bonheur »

‘ Proverbe indien’

1.1.Contexte du travail

Ce travail s'inscrit dans le cadre d'un projet de réalisation d'un environnement informatique pour l'apprentissage humain (EIAH) dédié à l'algorithmique. Par ce projet, nous avons voulu valoriser les recherches menées au niveau de notre équipe EIAH depuis une dizaine d'années dans les domaines de recherche EIAH, EAO (Enseignement assisté par ordinateur), Systèmes tuteurs, etc.

Ce mémoire présente une partie essentielle de ce projet qui est l'évaluation. Les résultats dont il est fait état dans ce manuscrit viennent s'ajouter aux résultats obtenus dans les travaux antérieurs de l'équipe EIAH dans le domaine de l'évaluation. Puissent-ils contribuer à une meilleure assise de futurs travaux.

1.2.Problématique

L'évaluation est essentielle dans un processus d'apprentissage, elle ressemble à une boussole dans le sens où elle permet d'orienter les évaluateurs et leur donne des informations détaillées sur un ensemble d'apprenants. Ces informations peuvent être de différentes natures : les connaissances acquises ou non acquises par un (voire plusieurs) apprenant (s), les lacunes et difficultés, les classes des apprenants, les erreurs fréquentes, etc.

L'évaluation est une charge rébarbative pour l'enseignant. Plusieurs méthodes et outils peuvent être utilisés pour évaluer. Avec l'évolution technologique, l'évaluation peut être assistée par ordinateur. Les promesses d'une évaluation automatique, sont vraiment nécessaires pour réduire voire éliminer la charge de l'évaluation et produire un feedback commun des étudiants). L'évaluation automatique peut aider les enseignants à créer des modèles éducatifs, par conséquent concentrer leurs efforts sur l'enseignement qui nécessite fréquemment l'interaction Enseignant-Apprenant.

En ce qui concerne les EIAH, la question de l'évaluation automatique des productions d'élèves est, pour les concepteurs d'EIAH, complexe et, pour les enseignants et les apprenants, porteuse d'attentes comme dans la classe (Bouhineau et Puitg, 2011).

L'algorithmique est l'une des compétences essentielles en informatique. Un cours d'algorithmique consiste à présenter des concepts qu'on veut évaluer (apprendre) de plusieurs points de vue. En algorithmique, les productions des apprenants sont des

algorithmes. En fait, dans ce genre d'évaluation, il n'existe pas de corrigé type, les étudiants utilisent des stratégies différentes pour résoudre le même problème. Un algorithme a une certaine structure, et lors de l'évaluation plusieurs paramètres doivent être pris en compte (diversité des solutions, critères de qualité, etc), ce qui rend l'évaluation plus difficile. Certaines caractéristiques peuvent être évaluées automatiquement, d'autres obtenues en utilisant une approche semi automatique. La correction manuelle des algorithmes permet de comprendre toutes les solutions possibles. Néanmoins, une évaluation est une activité humaine, ce qui introduit de la subjectivité.

Dans le contexte d'un EIAH dédié à l'algorithmique, l'évaluation des apprenants représente un véritable défi. En effet, les méthodes d'évaluation automatique les plus courantes (QCM, Questions à trous, etc) sont inadaptées à une discipline telle que l'algorithmique (Bouhineau et Puitg, 2011) qui nécessite une méthode capable d'évaluer finement les compétences des apprenants. D'un autre coté, si l'utilisation d'un véritable compilateur, même pour du pseudo code, permet d'éditer des algorithmes et de les tester, elle ne permet pas, en revanche, d'avoir un feedback détaillé sur les connaissances et compétences des apprenants. Par conséquent, on ne peut pas ni localiser les lacunes et difficultés qu'un apprenant peut avoir, ni y remédier. Donc, il est très important d'avoir une méthode qui permette d'évaluer finement les connaissances et compétences des apprenants. Cette méthode ne doit pas être fermée comme dans les QCM, elle doit prendre en considération la diversité des solutions proposées par les apprenants.

1.3.Approche et motivations

Dans le but d'évaluer les apprenants en algorithmique, notre objectif est de comprendre des algorithmes (en pseudo code) proposés par ces derniers (apprenants). Comme nous savons ce que ces algorithmes devront faire, il ne nous reste que comprendre comment ils le font. Cela nous permet de prévoir les stratégies adoptées, donc d'avoir une idée profonde sur le raisonnement de l'apprenant, ses lacunes et difficultés.

Notre idée consiste à utiliser des modèles de propositions préconstruits, chacun représentant une proposition possible (algorithme) pour la résolution d'un problème donné, organisés sous forme de tâches. Ces modèles de propositions servent de base pour reconnaître la proposition de l'apprenant. Cette reconnaissance repose sur des techniques de compréhension de programme, utilisées en génie logiciel. Cependant, prévoir toutes les

propositions possibles d'un problème est une mission difficile, même pour l'expert le plus avisé. Aussi, nous profitons des productions des apprenants pour sélectionner les modèles de propositions, corrects ou incorrects, jugés pédagogiquement intéressants par l'expert. L'utilisation des modèles incorrects permet de localiser la défaillance du raisonnement des apprenants.

Pour être indépendant des langages nous allons utiliser le pseudo code, ce qui permet aux apprenants d'être dans un niveau d'abstraction élevé et de ne pas se perdre dans les détails techniques. Par conséquent, ils vont concentrer sur la résolution du problème.

Les modèles de propositions (corrects et incorrects) sont composés de tâches. Cette composition ressemble à une conception dans le génie logiciel, et traduit le raisonnement de l'apprenant. Il est ainsi possible de localiser les tâches dans lesquelles l'apprenant a échoué, ce qui assure une évaluation pertinente et détaillée

1.4.Organisation du document

Ce manuscrit est organisé de la manière suivante :

- Ce premier chapitre présente le cadre d'étude puis la problématique de l'évaluation en algorithmique. Il est clos par la présentation de l'organisation du mémoire.

- Une première partie propose ensuite un état de l'art de l'évaluation dans le contexte des environnements informatiques. Dans cette partie sont tout d'abord introduites quelques définitions de l'évaluation ainsi que les différentes formes d'évaluation existantes. Ce chapitre est suivi d'une synthèse sur l'existant en matière d'évaluation en programmation. En fin, un dernier chapitre présente qu'est ce la compréhension des programmes.

- Une deuxième partie présente nos différentes propositions.

- Enfin le dernier chapitre conclut sur les perspectives d'amélioration et les autres pistes de recherche suggérées par ce travail.

Première partie

État de l’art

Chapitre 2

L'évaluation

« *Le vrai génie réside dans l'aptitude à évaluer l'incertain, le hasardeux, les informations conflictuelles* »

'Winston Churchill'

2.1.Introduction

« L'évaluation est la lumière sur laquelle un processus d'apprentissage se développe. Elle est le compagnon indispensable de n'importe quelle formation, et d'une existence inévitable, que se soit dans un contexte professionnel ou éducatif. Elle joue un rôle primordial dans l'activité pédagogique, non seulement comme le moyen de vérifier les acquisitions mais aussi comme le moyen de motiver ces apprentissages et d'inciter les élèves à progresser » (Juwah, 2003).

L'évaluation est pratiquée dans de nombreux domaines mais nous nous intéressons ici à l'évaluation des apprentissages. Elle sert à produire de l'information éclairante pour prendre des décisions. Les modalités de l'évaluation sont fonction du type de décisions à prendre. Après avoir défini en quoi consiste l'évaluation, nous en distinguerons les différents types, en fin nous parlerons de quelques systèmes d'évaluation de programmes.

2.2.Qu'est ce qu'Evaluer ?

La définition du dictionnaire (Petit Robert, 2006) indique que l'évaluation est l'action d'évaluer, de déterminer la valeur ou l'importance d'une chose. Pour Neil Postman, théoricien en communication "l'évaluation est un élément inévitable, nécessaire et naturel dans toute communication humaine" (Postman, 1979). Cette évaluation évoquée par Postman est proche du jugement que nous portons les uns sur les autres, au cours de nos rencontres et de nos échanges. Elle repose souvent sur des préjugés, des idées reçues, des stéréotypes dont nous ne sommes pas toujours conscients mais auxquels nous échappons difficilement.

Nous entendons évaluer comme une prise d'informations sur un observable en regard de références pour éclairer des décisions (Stufflebeam et al, 1980) (Charles, 90).

Pour l'enseignant, pour qui il n'est "pas possible d'enseigner sans évaluer" (Dintilhac et RAK, 2005), il ne s'agit évidemment pas de ce type d'évaluation. L'évaluation a principalement pour objectif de permettre aux enseignants de vérifier que leurs élèves ont acquis les connaissances utiles et nécessaires à la poursuite d'une formation. Généralement, cette évaluation est réalisée périodiquement, c'est le contrôle continu (Rolland et al, 2000)

rencontré du premier cycle au supérieur. Chaque contrôle permet à l'enseignant d'apprécier la performance de ses élèves et de se faire ainsi une idée de l'état de leurs connaissances.

Pour Charles Hadji (Charles, 1977), évaluer signifie: interpréter, vérifier ce qui a été appris, compris, retenu, vérifier les acquis dans le cadre d'une progression, juger un travail en fonction des critères donnés, estimer le niveau de compétence d'un apprenant, situer l'apprenant par rapport à ses compétences, déterminer le niveau d'une production donnée par l'apprenant.

Cette appréciation se concrétise généralement par l'obtention d'une note. Les notes permettent de faire des moyennes, des moyennes de moyennes qui permettent de sélectionner l'orientation scolaire des élèves. La note est ainsi un élément crucial car elle conditionne le futur des élèves. La note est aussi un indicateur qui permet à l'élève d'adapter sa démarche d'apprentissage et lui permet de savoir si la méthodologie qu'il emploie est en accord avec ce qui est attendu par l'enseignant. C'est la volonté d'obtenir de bons résultats qui doit inciter l'élève à apprendre et c'est la satisfaction qu'il éprouve à réussir qui doit lui donner l'envie de persévérer et d'aller plus loin.

Par ailleurs, L'enseignant n'est plus seulement confronté à la nécessité de comprendre ce qui fait obstacle à la réussite de ses élèves mais il peut aussi, lorsqu'il utilise certains outils informatiques, se servir de l'évaluation comme un "moteur" pour les apprentissages (Dintilhac & Rac, 2005), l'évaluation devenant un moyen pédagogique.

2.3.Types d'évaluations

L'évaluation est censée intervenir avant, pendant et à la fin de l'apprentissage sous diverses formes (Charle, 1977): l'évaluation pronostique, l'évaluation diagnostic, l'évaluation formative, l'évaluation sommative et l'auto-évaluation.

2.3.1. Avant la formation

Avant la prise de décision d'une orientation ou avant le début d'une formation, il est important, voire nécessaire, de savoir quelles sont les acquisitions du futur apprenant. Les différentes formes d'évaluation qui précèdent l'acte d'apprendre ont été mises en place justement pour dresser un bilan de compétences, des acquis des étudiants. Sont-ils aptes à suivre cette formation ? Se sont-ils bien orientés ? Voilà les questions que posent ces évaluations.

2.3.1.1.Evaluation pronostique

Cette évaluation permet de décider de l'admission, de la promotion, de l'orientation d'une personne en vue de suivre un temps de formation (Allal, 1991), (Perrnoud, 2001).

2.3.1.2.Evaluation diagnostique

« Evaluation qui, effectuée avant une action de formation ou une séquence d'apprentissage, a pour but de produire des informations permettant soit d'orienter le formé vers une filière adaptée à son profil, soit d'ajuster la formation à ce profil. » (charles, 2000). Cette évaluation permet de dresser un bilan des acquisitions faites par l'apprenant pour pouvoir s'assurer de ses réelles compétences en vue de suivre une formation ou de l'orienter. Elle définit les aptitudes, les motivations, les capacités à suivre. Certains chercheurs appliquent aussi cette évaluation pendant le temps de formation.

2.3.2. Pendant la formation

L'évaluation pendant le temps de formation est importante. C'est celle qui permet de vérifier l'état des acquisitions des élèves. Elle fait le lien entre ce que l'apprenant doit apprendre, ce qu'il a appris et comment il l'a appris. Elle permet une reconsidération des savoirs enseignés et des pédagogies appliquées. La principale forme d'évaluation est celle dite formative.

2.3.2.1.Evaluation formative

On parle d'évaluation formative quand celle-ci permet à l'apprenant d'améliorer ses démarches d'apprentissage et de renforcer ses connaissances et ses compétences. Elle aide l'élève à apprendre, en lui permettant de se rendre compte de ce qui se passe. Qu'est-ce qu'il apprend, pourquoi il apprend, et surtout comment il apprend (Geneviève, 1995).

D'après (Allal, 1991), l'évaluation formative permet à l'enseignant de prendre des décisions d'adaptation pédagogique. Comment modifier le contenu du cours pour être sûr que les élèves apprennent. Cette évaluation passe par une communication au sein de la classe. Le but est de fournir au maître et/ou à l'élève un regard antérieur concernant les progrès de l'élève et par la même repérer les problèmes d'apprentissage ou d'enseignement.

Pour faciliter la mise en place d'une pratique de l'évaluation formative, il est possible d'impliquer directement les élèves. Cette implication peut être analysée par :

1. L'auto évaluation,
2. L'évaluation mutuelle (entre élèves),
3. La co-évaluation (entre l'élève et le maître).

Ces manières d'impliquer les élèves directement développent la vigilance et la capacité à écouter, à se mettre en état de réceptivité. L'évaluation formative a pour ambition de contribuer à la formation. Elle cherche à guider l'apprenant pour faciliter ses progrès. C'est une évaluation centrée sur la gestion des apprentissages. Les étapes essentielles de l'évaluation formative sont :

1. Le recueil d'informations concernant les progrès et les difficultés d'apprentissage rencontrés par l'élève.

2. L'interprétation de ces informations dans une perspective diagnostique des difficultés de l'élève.

3. L'adaptation des activités d'enseignement, d'apprentissage en fonction de l'interprétation faite des informations recueillies.

Il est important, pour mener à bien une évaluation formative, d'établir une stratégie d'évaluation. Mais à chaque classe une stratégie. C'est pourquoi la tâche de l'enseignant est de construire une stratégie d'évaluation formative qui soit applicable dans sa classe. Une des fonctions de l'évaluation formative est d'animer l'interaction entre les personnes (enseignant et élèves) et les matériaux (savoirs et apprentissages), cela pour une meilleure prise de conscience des acquis des élèves et de l'efficacité de la pédagogie de l'enseignant. Pour un grand nombre de personnes, l'évaluation formative prépare l'évaluation sommative, celle effectuée en fin de formation.

2.3.3. Après la formation

Les évaluations de fin de formation sont celles qui permettent de dresser un tableau des compétences acquises pendant la formation et qui les valident. Elles ont souvent un rôle de certification. Ce sont des évaluations dites binaires. C'est oui ou non.

2.3.3.1.Evaluation sommative

Selon (Allal, 1991), cette évaluation permet la décision de certification intermédiaire ou finale. Elle permet d'établir un bilan de ce que l'élève a appris pour décider d'un passage en niveau supérieur ou de l'obtention d'un diplôme. Cette évaluation passe par une communication avec l'extérieur, parents d'élèves, autres institutions, etc.

La décision d'orientation ou de certification n'est prise que sur un échantillon représentatif des objectifs du cours. D'après (Charles, 2000), l'évaluation sommative est l'évaluation par laquelle on fait un inventaire des compétences acquises, ou un bilan, après une séquence ou une activité de formation d'une durée plus ou moins longue. Elle permet de vérifier que les objectifs sont atteints et surtout de l'attester socialement. Ce dernier point rejoint le point de vue de (Allal, 1991) sur la communication avec l'extérieur, surtout avec les parents d'élèves. Une note ou un passage en classe supérieure atteste du travail fait en cours et du niveau de l'élève. C'est à travers cela qu'on jugera de la crédibilité et de l'efficacité d'une formation et de celle de ses enseignants.

2.3.3.2.Evaluation normative

D'après (Geneviève, 1995) c'est une évaluation qui rend des comptes, qui a pour fonction de vérifier des acquis en comparant ce qui est observé à une norme extérieure attendue, à un étalon référent.

2.3.4. Evaluation critériée

Selon (Charles, 2000), c'est une évaluation dont le cadre de référence est constitué par des objectifs ou des performances cibles. L'évaluation ***critériée*** a pour premier objectif de permettre à l'apprenant de savoir ce que l'on attend de lui et de se situer en conséquence. Elle a pour but de fournir des informations sur l'écart entre le comportement visé et le degré de maîtrise des acquisitions attendues.

D'après la définition donnée par (Charles, 2000), cette évaluation nécessite une mise en place d'objectifs précis. Il faudra alors introduire des objectifs dans sa pédagogie, c'est à dire

adopter une pédagogie par objectifs. Cela signifie aussi que les objectifs doivent être formulés dès le début de la formation pour que l'apprenant puisse se situer par rapport à ce que l'on attend de lui.

2.4. Evaluation en EIAH

Pour pouvoir évaluer, l'enseignant construit et utilise des outils qui lui permettent de récolter les informations indiquant l'état du savoir de l'apprenant. L'évaluation est un besoin que l'informatique peut aider à mettre en œuvre. Que ce soit en classe ou au sein des EIAH, l'évaluation est un dispositif clé. En effet, cette évaluation dans le contexte d'un EIAH doit (Juwah, 2003) :

- ✓ être motivante pour l'apprenant,
- ✓ encourager une activité d'apprentissage soutenue,
- ✓ contribuer à la progression de l'apprenant,
- ✓ être faible en coût humain et facilement maintenable.

Cette évaluation assistée par ordinateur améliore la détection des progrès et lacunes des apprenants suivant la rapidité et la qualité des interactions qu'ils ont avec et dans l'EIAH (Bull, 1999).

Dans les EIAH, l'outil le plus utilisé est la formulation des questions. Il existe divers types de questions (Gilbert, 1980): question alternative, question d'ordre, question à choix multiple, question à réponse ouverte, etc.

Il est possible de définir plusieurs grandes familles de pratique d'évaluation en EIAH :

- ✓ L'auto-évaluation individuelle (David, 2003) ou collective (Juwah, 03). Cette évaluation formative permet aux apprenants de réguler eux-mêmes leurs apprentissages.
- ✓ L'évaluation diagnostic (Green et al, 1984) (Delozanne et Grugeon, 2004) en évaluant les productions de l'apprenant réalise une évaluation sommative de ses compétences.
- ✓ L'évaluation de la participation (Bratitsis et Dimitracopoulou, 2005), qui se rencontre le plus souvent dans les EIAH utilisant des forums, vise à évaluer la participation des apprenants en s'intéressant au ratio qualité/quantité d'interventions. C'est une évaluation qui est bien souvent normative.

- ✓ L'assistance à l'évaluation est plus du ressort de la mesure, qui consiste à recueillir et fournir à un tuteur des indicateurs relatifs à l'activité en cours (George, 2001) (MAZ et Milani ,05) (MER et Acef. ,04).

2.5. Evaluation et programmation

Dans les cours de programmation, l'évaluation automatique est un besoin pratique de beaucoup d'enseignants. Elle peut leur fournir des moyens de suivre facilement la progression des étudiants et d'identifier rapidement des besoins d'améliorations du cours. Plusieurs systèmes d'évaluation automatique en programmation ont été proposés tels que : CourseMarker (Higgins et al, 2003), Assyst (Jackson et Usher, 1999), Online judge (Cheang et al, 2003), et BOSS (Luck et Joy, 1999).

En programmation l'évaluation peut se faire sur plusieurs aspects, par exemple Ellsworth et ses collègues (Ellsworth et al, 2004) utilisaient le système Quiver de sorte que les étudiants de génie logiciel définissent des interfaces et des tests pour des méthodes et des classes, et les étudiants qui suivent des cours de programmation fassent l'exécution. De cette façon, les deux groupes apprennent des aspects concernant le logiciel de spécification, d'implémentation et de test avec l'assistance d'un système automatique d'évaluation.

L'évaluation automatique doit soutenir également l'organisation du processus d'attribution des notes, dans le peer-reviewing (Zeller, 2000), (Sitthiworachart et Joy, 2003), les étudiants présentent leurs observations sur d'autres programmes. Dans ce genre de processus, les étudiants apprennent à évaluer des programmes meilleurs et à recevoir plus de rétroactions que celles données par des enseignants.

2.5.1. Semi-automatique VS Automatique

Les enseignants conviennent souvent qu'il n'est pas possible d'évaluer automatiquement tous les aspects concernant la bonne programmation. D'une part, bien que l'évaluation automatique puisse ne pas être de haute qualité comme celle effectuée par un enseignant, elle est au moins en partie compensée par sa vitesse et sa disponibilité. Une approche éducative est d'évaluer seulement les aspects qui peuvent être entièrement automatisées. Ceci convient pour de petites tâches, où

l'objectif principal est d'enseigner à des étudiants les bases de la construction du programme et les fondations d'un langage de programmation.

Avec de plus grandes tâches, une approche plus commune semble être de combiner l'évaluation manuelle et automatique. Si le processus d'évaluation est bien soutenu par des outils, ce genre d'approche semi-automatique peut être très efficace. Il donne à des enseignants plus de temps pour se concentrer sur l'évaluation des tâches exigées, par exemple, donnant la rétroaction sur la conception du programme. Il fournit également une possibilité de vérifier une deuxième fois les résultats de l'évaluation automatique.

2.5.2. Evaluation Formative Vs Evaluation Sommative

Certains outils automatiques d'évaluation sont conçus principalement pour l'évaluation sommative, par exemple BOSS (Luck et Joy, 1999), tandis que d'autres montrent aux étudiants les résultats de l'évaluation automatique et leur permettent des ressoumissions, si l'étudiant n'est pas satisfait des résultats, par exemple CourseMarker (Higgins et autres, 2003).

La dernière approche peut être vue en tant qu'évaluation formative, puisque son rôle est d'aider des étudiants en fournissant un feedback sur leur travail et les laisse l'améliorer en conséquence. Bien que beaucoup d'approches soulignent la possibilité pour l'itération, ils s'attendent à ce que les étudiants soumettent une version complète du programme dès la première soumission.

L'évaluation automatique peut être employée pour des buts sommatifs dans des tâches de mini projet et dans des situations de programmation contrôlée appelées examens en ligne. Ceux-ci fournissent des moyens d'assurer des compétences de programmation pour les étudiants, puisqu'il y a moins de possibilités pour la fraude.

2.6. Conclusion

Nous voyons bien, après l'énumération de ses différentes formes, que l'évaluation, suivant le moment où elle intervient, joue un rôle très différent dans une formation. Elle permet de faire un bilan de compétence, une sélection avant, une réorientation ou un rééquilibrage pendant et une certification ou une orientation après.

L'évaluation joue un rôle important du point de vue des apprentissages et des pédagogies appliquées. Il semblerait que ce soit la fonction d'évaluation pendant le temps de formation qui suscite le plus de réflexion dans ce domaine. En effet, les évaluations formatives ou diagnostiques permettent une remise en question permanente des objectifs d'apprentissage et des savoirs enseignés. Qu'est que l'élève a appris ? Comment l'a-t-il appris ? Le fonctionnement de l'évaluation formative est un avantage pour l'ensemble du public, les enseignants comme les élèves, car elle permet de se situer dans la formation.

L'évaluation, ce n'est pas simple. C'est un exercice où l'on prend des risques : on ne peut jamais être sûr d'avoir bien évalué, toute évaluation renvoie des interrogations à l'évaluateur. Mais elle fait partie de la mission de tout enseignant.

Chapitre 3

L'évaluation en programmation

« Un langage de programmation est censé être une façon conventionnelle de donner des instructions à un ordinateur, et doit pouvoir être écrit et relu par des personnes différentes. Il n'est pas censé être obscur, bizarre et plein de pièges subtils (ça, ce sont les attributs de la magie) ».

'*Dave Small'*

3.1.Introduction

Dans la littérature, plusieurs outils pour l'évaluation automatique des programmes ont été proposés. L'utilisation de ces outils pour l'évaluation, assure plusieurs avantages tels que : rapidité, disponibilité, et objectivité. Cependant, une bonne conception pédagogique est indispensable, ainsi qu'une bonne expertise. En informatique, l'utilisation des travaux pratique est la façon la plus fréquente d'évaluer (Carter et al, 03). L'évaluation assistée par ordinateur est utilisée généralement pour diffuser l'information, ou bien comme outil d'évaluation généralement indépendant du contenu de la matière, les QCMs sont l'exemple le plus fréquent. Dans les cours de programmation, les étudiants créent des programmes obéissant à la syntaxe d'un certain langage de programmation. Ces programmes peuvent être analysés automatiquement sur les deux niveaux statique (code source) et dynamique (comportement). Cependant, *il demeure difficile d'assurer une évaluation pertinente sur le plan compétences de programmation.* Dans ce chapitre, après la présentation de certains travaux sur l'évaluation automatique des tâches de programmation, nous allons présenter les aspects évalués statiquement/dynamiquement. Nous présenterons aussi, les systèmes existants pour chaque aspect.

3.2.Evaluation automatique des tâches de programmation

L'évaluation en programmation est difficile et fréquemment mise en question. McCracken et ses collègues ont trouvé dans leur étude que les compétences de programmation des étudiants en première année étaient beaucoup plus faibles que prévu. En fait, la plupart des étudiants ne pouvaient pas accomplir la tâche de programmation demandée à temps (McCracken et al, 2001). Lister et Leaney ont critiqué les pratiques et les objectifs d'évaluation dans les premiers cours de programmation en général (Lister et Leaney, 2003). Ils ont proposé une classification, où chaque grade est clairement relié à certaines exigences, et les étudiants ont la possibilité de décider quels grades ils veulent poursuivre. Les tâches de programmation sont conçues selon différents niveaux cognitifs, et les tâches des niveaux les plus élevés correspondent aux meilleurs grades. Avec cette approche, il est admis que tous les étudiants ne possèdent pas des compétences de programmation de plus haut niveau à la fin du premier cours. Lister et Leaney encouragent des enseignants à concevoir l'évaluation selon les niveaux cognitifs définis en taxonomie des objectifs éducatifs (Bloom, 1956). Ces niveaux sont, du plus bas au plus haut : rappel, compréhension, application, analyse, synthèse, et évaluation.

On ne doit pas, cependant, oublier que la compréhension des concepts de programmation peut également être évaluée sans écrire de code. Cox et Clark (Cox et Clark, 1998) ont présenté des exemples de questions à choix multiple qui évaluent selon eux tous les niveaux cognitifs dans un cours d'introduction à la programmation. En plus des questions à choix multiple traditionnels, il y a également d'autres approches automatisées pour évaluer des concepts de programmation.

L'évaluation automatique peut, par exemple, être employée pour concevoir des diagrammes, comme implémenté dans CourseMarker (Higgins et al, 2002). Une autre approche est de simuler l'exécution d'algorithmes, par exemple, Trakla (Korhonen et Malmi, 2000) peut produire des exercices de simulation d'algorithme et évaluer les réponses automatiquement.

Cependant, la compréhension des concepts et des principes ne garantit pas la capacité de produire des programmes informatiques. Les programmeurs novices ont des problèmes communs pour exprimer leurs solutions en programmes informatiques, c.-à-d., en appliquant des concepts de programmation pour construire des programmes (Robins et al, 2003). Ainsi, si le but du cours est d'enseigner des compétences de programmation pratiques, celles-ci devraient être exercées et évaluées par des tâches de programmation pratiques. Woit et Mason ont également obtenu des résultats positifs en employant les jeux hebdomadaires qui ont été directement reliés aux exercices de programmation pratiques dans le cours (Woit et Mason, 2003).

Typiquement, des tâches de programmation sont évaluées par les programmes résultats, et les critères d'évaluation varient entre différents enseignants et universités. Par exemple, quelques approches d'évaluation holistiques et quelques critères d'évaluation analytiques sont plus ou moins détaillés. Olson (Olson, 1988) a étudié les différences entre ces approches et a noté qu'elles soulignent différents aspects dans la tâche. D'une vision holistique, un programme pourrait obtenir une note raisonnable même s'il a échoué dans quelques catégories analytiques, par exemple, compilation ou fonctionnalité de base et peut avoir une note d'échec dans l'évaluation analytique. Pendant que le travail d'évaluation dans de grands cours doit souvent être distribué à plusieurs enseignants, l'approche analytique permet de définir des critères d'évaluation détaillés d'usage courant. Par exemple, Becker (Becker, 2003) a proposé des rubriques pour définir les critères d'évaluation. Les critères détaillés sont une

nécessité pour les enseignants et peuvent également être publiés pour des étudiants, pour aider leur étude autodirigée.

En plus d'évaluer des compétences des étudiants à partir de leurs programmes, leurs habitudes de travail peuvent être prises en considération. Howles (Howles, 2003) a soulevé en question l'apprentissage de la culture de qualité logicielle. Elle a découvert d'une enquête locale sur les étudiants que seulement 5% des réponses des étudiants ont toujours conçus leur travail avant le codage et seulement 39% ont toujours examiné leur code de programme statiquement. D'ailleurs, la majorité des étudiants n'ont jamais exécuté des unités de tests. Le processus de fonctionnement des étudiants est difficile à évaluer, mais il peut être guidé par les tâches et la conception de l'évaluation. Howles exige aux étudiants de trouver, fixer et documenter tous les défauts trouvés dans l'évaluation. L'évaluation automatisée a pu être employée également dans ce genre de processus pour aider des professeurs et des étudiants à contrôler et comparer différentes versions de soumission de programme.

Bien que beaucoup d'auteurs semblent apprécier l'objectivité et l'efficacité de l'évaluation automatisée des tâches de programmation (par exemple, (Foxley, 1999), (Chen, 2004), Morris, 2004)), également des contre avis ont été présentés. Ruehr et Orr (Ruehr et Orr, 2002) ont discuté différents critères d'évaluation et ont considéré la démonstration interactive comme meilleure méthode d'évaluation des tâches de programmation. Ils la voient comme situation motivante pour des étudiants et des enseignants. Une situation personnelle de contact garantit le feedback souple et individuel sur le programme pour l'étudiant. L'approche est plus adaptée pour de petits groupes d'étudiants, mais pourrait également être employée sélectivement dans de plus grandes classes.

3.3.Evaluation statique des programmes

Dans ce mémoire l'évaluation statique signifie l'évaluation qui peut être effectuée en collectant des informations du code de programme sans l'exécuter. Depuis long temps, des tâches de programmation ont seulement été évaluées statiquement. Les étudiants ont soumis le listing du code et des sorties du programme, et les enseignants se sont basés dans leur évaluation sur l'inspection visuelle.

Avec la montée d'Internet, les soumissions sont devenues électroniques et l'évaluation a inclut l'exécution du programme. Mais il existe toujours un endroit pour l'évaluation statique, un programme peut être bon même s'il contient peu de fonctionnalités correctes.

En outre, l'analyse statique peut indiquer les aspects qui ont été laissés inaperçus par les cas de test limités. Un autre avantage de l'analyse statique est qu'elle peut être effectuée même s'il y a des problèmes dans le comportement dynamique du programme. Cependant, la plupart des méthodes d'analyse statique se fondent sur la structure formelle du langage de programmation, aussi elles exigent au programme d'être syntaxiquement et sémantiquement correct.

3.3.1. Style de codage

La condition de base pour analyser automatiquement un programme informatique est la syntaxe correcte. L'évaluateur le plus efficace et le plus évident pour ce dispositif est le compilateur du langage. De plus, les compilateurs ont souvent été complétés avec des outils additionnels pour trouver les insuffisances structurales, par exemple, Lint pour le langage C (Darwin, 1990).

De nos jours les compilateurs et leurs possibilités d'avertissement sont très efficaces et ne devraient pas être oubliés par les enseignants et les étudiants. Par exemple, le compilateur de GCC (GCC) peut fournir des rétroactions sur les variables inutilisées, les conversions de types implicite, et les aspects du langage qui ne suivent pas les normes. Ce sont des aspects automatiques d'évaluation qui peuvent facilement être implémentés pour n'importe quelle tâche de programmation en C ++.

En plus des exigences techniques, il y a également des exigences de style pour un bon code de programme. Le style de programmation, la lisibilité, etc. ont été intensivement recherchés depuis les années 80. Par exemple Oman et Cook (Oman et Cook, 1990) ont défini une taxonomie pour le style de programmation pour l'éducation. Rees (Rees, 1982) a présenté un analyseur automatique pour Pascal avec un modèle de graduation configurable de catégorie et 10 mesures différentes de code concernant la lisibilité du code. Son travail a servi de base à beaucoup d'outils exécutant l'évaluation automatique de style de programmation, par exemple, Ceilidh (Foxley, 1999) et Assyst (Jackson et Usher, 1997).

Dromey (Dromey, 1995) a publié un travail sur le style de codage à travers différentes perspectives, en la reliant aux attributs de qualité du logiciel classifiés dans la norme d'évaluation du logiciel ISO-9126. Il a relié des aspects au niveau code avec des facteurs de qualité généraux, tels que la fiabilité, la fonctionnalité et la maintenance du programme.

Un système automatique (PASS) a été implémenté pour évaluer ces aspects à partir des programmes en ADA, en C, et en Java. Style++ (Ala-Mutka et al, 2004) est un autre outil qui a été développé pour évaluer des facteurs de qualité des programmes en C ++.

En plus des aspects traditionnels du style de codage, l'outil se concentre sur les conventions de programmation orientée objet et sur les dispositifs compliqués du langage C++ qui mènent souvent aux erreurs dans la fonctionnalité du programme. Checkstyle (Checkstyle) est un logiciel libre pour vérifier des programmes en Java et peut être combiné à plusieurs environnements de programmation. Edwards (Edwards, 2004) a prévu de le combiner à leur système d'évaluation.

3.3.2. Erreurs de programmation

Bien que des erreurs fonctionnelles des programmes soient le plus généralement évaluées dynamiquement en exécutant le programme avec des données de tests, quelques erreurs ou au moins fragments soupçonneux du code peuvent également être identifiés statiquement. Ce genre d'analyse d'erreur se rapporte souvent à évaluer le style de programmation, par exemple, aux aspects de fiabilité du code de programme.

Avec des langages fonctionnels, la composition fonctionnelle et les réalisations de fonction définissent le style du programme aussi bien qu'ont des effets directs sur les fonctionnalités. Michaelson (Michaelson, 1996) a discuté les fondations des mesures du style et a implémenté un outil pour évaluer automatiquement les styles fonctionnels des programmes de SML. L'outil a été relié au système de Ceilidh et pouvait identifier plusieurs types d'erreurs typiques provoqués par des étudiants dans la programmation impérative. Pour un but semblable, Schorsch (Schorsch ,1995) a développé un outil pour identifier et donner la rétroaction descriptive sur des questions de modèle et la plupart des erreurs logiques communes des programmes Pascal. Son outil CAP identifie, par exemple, des erreurs en mettant à

jour une variable de commande de boucle ou des contradictions entre un type de paramètre et une utilisation.

Une autre approche intéressante est présentée par Xie et Engler (Xie et Engler, 2002), qui ont employé des redondances de code pour détecter des erreurs. En implémentant un outil pour détecter des opérations de quantité, des tâches superflues, le code mort, et des conditions redondantes, ils pouvaient trouver plusieurs erreurs du code source bien connu de Linux. Bien que leur travail ne soit pas à usage éducatif, il pourrait être employé en tant qu'assistant automatique pour que des enseignants ou des étudiants détectent probablement les tâches incorrectes dans le programme.

3.3.3. Métriques logicielles

Les métriques logicielles sont considérées comme des mesures générales qui caractérisent des programmes informatiques. Si la métrique a été identifiée pour mesurer des caractéristiques importantes du code d'un programme, elles peuvent également constituer une base pour évaluer et comparer des programmes.

En outre, la métrique numérique peut être facilement obtenue automatiquement. Cependant, pour l'usage éducatif, les mesures doivent également être appropriées pour l'étude et/ou l'enseignement. Par exemple, il n'y a aucun sens d'exiger aux étudiants de soumettre un programme qui a une complexité X, ou contient Y lignes de code. Par conséquent, ces mesures sont souvent reliées aux aspects concernant le style et la conception du programme. Par exemple, Mengel et Ulans (Mengel et Ulans, 1999) ont rassemblé automatiquement plusieurs métriques logicielles à partir des tâches des étudiants.

Ils ont considéré les métriques comme étant les indicateurs clairs de performance des étudiants et également les indicateurs possibles des besoins de développement institutionnel. Les métriques de Halstead (Halstead, 1977) sont généralement des mesures reconnues qui sont basées sur le calcul de différents attributs, tels que le nombre d'opérateurs et d'opérandes dans un programme. Une taille théorique de programme peut être calculée par le nombre d'attributs globaux et uniques, et peut être employée pour identifier des programmes inutilement grands. La taille de

programme peut également être mesurée simplement en comptant le nombre de lignes de programmation.

Hung, Kwok et Chan (Hung et al, 1993) ont étudié des métriques différentes avec des tâches de programmation et sont venus à la conclusion que le nombre de lignes de code était une bonne mesure des compétences de programmation des étudiants. Une autre métrique bien connue a été présentée par McCabe (McCabe, 1976), qui a proposé une mesure de complexité qui définit la complexité d'un programme par sa structure de contrôle. Cette mesure a été employée pour l'analyse automatique, par exemple, dans Assyst (Jackson et Usher, 1997) en l'employant pour comparer la différence de complexité entre les programmes des étudiants et la solution modèle. Si la solution d'un étudiant est beaucoup plus complexe que la solution modèle, il y a sûrement quelque chose d'incertain dans la conception du programme.

Là existe également plusieurs métriques orientées objet, par exemple, examinées par Purao et Vaishnavi (Purao et Vaishnavi, 2003). Bien que la littérature ne fournisse pas des rapports d'expérience d'employer ces métriques dans l'éducation, ceux-ci peuvent être vus pour avoir les mêmes possibilités que la métrique traditionnelle du génie logiciel. Les raccordements entre les classes, le nombre de variables etc. peuvent fournir une information sur la conception du programme. Cette information peut être comparée avec la solution modèle pour identifier les conceptions des propositions ou pour comparer les programmes soumis entre eux.

3.3.4. Conception

Les professeurs doivent souvent évaluer si les programmes soumis répondent à l'interface donnée ou aux exigences structurales. Ceci peut être évalué automatiquement, par exemple, en comparant la conception du programme soumis aux spécifications du problème, à la solution modèle du professeur ou à l'ensemble de solutions applicables.

Thorburn et Rowe (Thorburn et Rowe, 1997) ont implémenté un système qui identifie automatiquement la structure fonctionnelle du programme en C. Ils l'appellent le plan de solution du programme et le comparent au plan de solution du programme modèle, ou à un ensemble de plans possibles. L'équivalence de différentes fonctions est déterminée en comparant **des sorties de fonctions à un**

ensemble randomisé d'entrées. Egalement, l'outil a effectivement localisé des erreurs dans les programmes des étudiants en notant des appels de fonction mal placés. De même, Scheme-robo (Saikkonen et al, 2001) évalue si les étudiants suivent les conditions structurales par soustraire un arbre structural à partir d'une fonction donnée et de le comparer à la structure demandée. MacNish (MacNish, 2000) a employé les outils de réflexion du paquet de java.lang.reflect pour analyser si les interfaces de classe et les signatures de méthode dans des programmes Java des étudiants ont répondu aux exigences données.

En plus des outils complètement automatiques d'évaluation, il y a des outils qui produisent des valeurs et des diagrammes de mesure pour aider l'enseignant dans l'évaluation finale de la tâche.

Particulièrement la recherche en ingénierie logicielle a produit des idées intéressantes et des outils qui pourraient également être utilisés pour l'éducation. Par exemple, Antoniol et ses collègues (Antoniol et al, 2001) ont présenté un outil qui identifie les modèles communs de conception des spécifications de code de programme ou de conception UML. Cet outil peut être utilisé pour aider à vérifier que l'exécution du programme d'un étudiant correspond au modèle de conception donné par l'enseignant.

3.3.5. D'autres aspects

En plus des aspects plus généraux, les enseignants recherchent parfois un certain mot ou expression dans le code de programme. Ceci peut être employé, par exemple, pour examiner si un étudiant a utilisé la structure demandé du langage de programmation ou une certaine fonction d'une bibliothèque. Ce genre de recherche par mot-clé est implémenté dans Scheme-Robo (Saikkonen et al, 2001). Cet outil peut être employé, par exemple, pour évaluer si la structure du programme est purement fonctionnelle en recherchant des primitifs set !, set-car !, et set-cdr. Une approche plus souple a été implémentée dans Ceilidh (Foxley, 1999) en définissant des expressions régulières à rechercher dans le code du programme de l'étudiant.

Le plagiat a toujours été un problème avec des tâches de programmation, puisque les programmes informatiques sont des fichiers textes facile à copier. Plusieurs outils automatisés ont été implémentés pour la comparaison de programmes, et par exemple Online judge (Cheang et al, 2003), CourseMarker (Higgins et al, 2003) et BOSS (Luck et

Joy, 1999) incluent des outils pour détecter le plagiat. Verco et Wise (Verco et Wise, 1996) ont comparé des outils automatisés basés sur l'attribut comptant des mécanismes par opposition aux systèmes qui utilisent l'information structurale du programme. Cet attribut a effectivement identifié les copies qui étaient très étroites entre eux, mais généralement l'approche structurale était plus efficace. L'information structurale est incluse dans les méthodes de MOSS (MOSS) et JPLAG (JPLAG), les services bien connus de détection de plagiat d'aujourd'hui. MOSS est basée sur l'empreinte digitale de document (Schleimer et al, 2003) et JPLAG utilise des indications et des patterns (Prechelt et al, 2002).

3.4.Evaluation dynamique

Kay, Scott, Isaacson, et Reek (Kay et al, 1994) ont déjà déclaré qu'il n'est pas possible de classer uniformément et complètement les programmes des étudiants sans une assistance automatique. Ceci s'applique particulièrement aux aspects dynamiques du programme, puisque même les petits programmes ont typiquement un grand nombre de chemins possibles d'exécution. L'automatisation fournit des moyens de couvrir systématiquement un grand nombre de différentes possibilités d'exécution. Cependant, le lancement d'un programme écrit par un étudiant est une tâche risquée. Le programme peut avoir des bogues ou même des aspects malveillants. Par exemple, un programme peut essayer de supprimer ou lire des dossiers de l'environnement courant, base de données de système par exemple, du professeur de la machine ou de l'évaluation. Il n'est pas rare que les programmes des étudiants ont des bogues qui font réserver des espaces considérables de mémoire ou du temps- CPU, gênant d'autres processus fonctionnant sur l'ordinateur. Ce sont des soucis qui doivent toujours être pris en compte lors de l'examen des programmes des étudiants. Ainsi, une condition essentielle pour l'évaluation dynamique automatisée est de fournir un environnement courant pour exécuter des programmes des étudiants sans risques.

3.4.1. Fonctionnalité

La forme la plus commune d'évaluation des tâches de programmation consiste à vérifier que le programme fonctionne selon les conditions données. La fonctionnalité est habituellement examinée en lançant le programme avec plusieurs jeux de tests. La qualité de l'évaluation dépend de la conception des cas de test.

Les résultats sont typiquement comparés à des spécifications séparées ou en exécutant un programme modèle pour la comparaison. L'exactitude de la fonctionnalité est alors comparée par l'impression sur papier ou des valeurs de retour.

Des outils d'évaluation de fonctionnalité des tâches de programmation tel que Try (Reek, 1989), ont été implémentés déjà dans les années 80. Ce genre d'évaluation est de nos jours typiquement inclus à tous les outils versatiles d'évaluation, tels que Ceilidh (Foxley, 1999) qui était appelé par la suite CourseMarker (Higgins et al, 2003), Assyst (Jackson et Usher, 1997), HoGG (Morris, 2003), Online judge (Cheang et al, 2003) et BOSS (Luck et Joy, 1999). Ceux-ci évaluent la fonctionnalité du programme en comparant ses sorties. Ceilidh/CourseMarker vérifie également le statut de retour du programme.

L'implémentation de base pour la comparaison des sorties est de comparer le texte de sortie du programme au texte modèle de sortie, ignorant des caractères de blanc. Assyst emploie la configuration avec un modèle implémenté avec Lex et Yacc tandis que Ceilidh/CourseMarker et HoGG emploient des expressions régulières pour définir les critères d'évaluation pour les sorties du programme. Ces approches offrent à des professeurs une possibilité de fournir aux étudiants un certain degré de liberté dans le format des sorties, si considéré nécessaire.

Il est également possible d'évaluer automatiquement la fonctionnalité de plus petites entités que des programmes complets. Par exemple, Quiver (Ellsworth et al, 2004) et l'approche proposée par (Bettini et al, 2004) peuvent évaluer des fonctions et des méthodes simples dans Java. Les méthodes sont exécutées avec les classes de réflexion de Java qui fournissent des moyens d'appeler des méthodes basées sur leur signature.

WebToTeach (Arnow et Barshay, 1999) et ELP (Truong et al, 2003) fournissent une possibilité pour évaluer même des instructions. Ceci est réalisé en combinant le fragment de code d'étudiant à un modèle de l'enseignant avant la compilation. L'Scheme-robo (Saikkonen et al, 2001) évalue des programmes implémentés dans le langage Scheme. Puisque le langage suit le paradigme fonctionnel, l'interpréteur peut exécuter n'importe quelle fonction. Ainsi, il est possible d'examiner des programmes partiels sans arrangements spéciaux. Lors de l'évaluation d'une fonction, l'exactitude est décidée par la sortie de la fonction, au lieu d'étudier la sortie du programme (fonction).

Les programmes informatiques typiques ne sont plus des programmes de traitement en lots appelés à partir de la ligne de commande, mais ont généralement des interfaces utilisateurs graphiques. Par conséquent, les cours de programmation ont également de telles tâches. L'évaluation de la fonctionnalité d'un programme avec une interface graphique exige des moyens de traiter et de mesurer des actions et des réponses communiquées par l'interface utilisateurs. JEWL (English, 2004) est une bibliothèque de langage qui est conçue pour la programmation en Java avec les interfaces utilisateurs graphiques. Elle fournit à des étudiants les composants graphiques semblables à ceux dans la bibliothèque standard. En même temps, la bibliothèque fournit aux professeurs une possibilité pour contrôler les événements du programme avec des textes d'entrée et pour étudier le résultat des actions sous forme textuelle. Par conséquent, la fonctionnalité du programme peut être évaluée automatiquement en comparant le texte produit à certains événements d'entrée, comme si évaluait un programme en ligne de commande.

3.4.2. Efficacité

Des approches automatisées d'évaluation de l'efficacité du programme sont typiquement basées sur l'exécution du programme avec différents cas de test et mesurent son comportement pendant l'exécution. Les résultats sont souvent comparés à une solution modèle existante. Ainsi, le succès de l'évaluation d'efficacité dépend largement de la qualité de la conception du cas de test et de la solution modèle.

Une mesure simple d'efficacité est le temps de fonctionnement du programme, mesuré par l'horloge ou le temps- CPU utilisé. Le temps d'horloge est employé souvent pour s'assurer que le programme se termine après un certain délai. Mesurer le temps- CPU pour évaluer l'efficacité est utilisé, par exemple, dans Assyst (Jackson et Usher, 1997) et Online judge (Cheang et al, 2003). Cependant, ce genre d'évaluation d'efficacité est affecté par plusieurs aspects du programme. Par exemple, bien que le but de la tâche soit d'implémenter une structure de données efficace pour stocker et rechercher des données, la mesure d'efficacité peut être tordue par différentes réalisations pour des actions d'entrée-sortie de données.

Ces problèmes peuvent être réduits en concevant les tâches pour souligner l'issue exigée dans l'implémentation. Par exemple, Hansen et Ruuska (Hansen et Ruuska, 2003) ont résolu le problème en offrant à des étudiants un module d'entrée/sortie commun pour l'usage dans les tâches qui se concentrent sur des algorithmes informatiques efficaces.

L'efficacité peut également être évaluée en étudiant le comportement d'exécution de différentes structures à l'intérieur du programme. Ceilidh (Foxley, 1999) et Assyst (Jackson et Usher, 1997) fournissent le profilage dynamique pour l'évaluation d'efficacité. Ceci est fait en calculant combien de temps certains blocs et instructions sont exécutés et en comparant les résultats aux valeurs obtenues à partir de la solution modèle.

3.4.3. Jeux de test construits par des étudiants

Le but pour développer des approches automatiques efficaces de test n'est pas de laisser des étudiants être paresseux avec leur travail. Les étudiants devraient apprendre à concevoir des cas de test et à examiner leurs programmes complètement avant la soumission. Le test est une phase essentielle dans le développement des programmes.

Les enseignants ont noté qu'en offrant les outils automatiques d'évaluation aux étudiants, ils doivent s'assurer par une stratégie d'évaluation que les étudiants conçoivent des jeux de tests et apprennent à examiner leurs programmes seuls (Chen, 2004), (Edwards, 2004). Ceci est typiquement évalué en exigeant aux étudiants qu'ils soumettent des ensembles de données de tests ainsi que la tâche de programmation et puis en évaluant la qualité des données de test soumises.

Assyst (Jackson et Usher, 1997) était le premier outil qui a fourni l'évaluation des données de test des étudiants. L'évaluation a été basée sur une mesure du taux de couverture de l'ensemble de données de test de l'étudiant par rapport aux lignes dans le programme de l'étudiant. Chen (Chen, 2004) évalue la suite des tests des étudiants en lançant un ensemble de programmes avec des erreurs proposées par l'enseignant. Le grade est donné selon le nombre de programmes avec des erreurs indiquées par les tests. Edwards (2004) décrit un système qui focalise sur l'amélioration des compétences de test des étudiants avec l'évaluation automatique.

Quand un étudiant soumet un ensemble de données de test, il est évalué avec la solution référence du professeur pour vérifier sa validité contre des spécifications de problème et pour mesurer à quel point elles couvrent tous les différents chemins d'exécution. Alors la fonctionnalité du programme de l'étudiant est mesurée avec ce test et ces trois points sont multipliés ensemble pour donner les points finaux. Par

conséquent, un étudiant doit développer un test complet et valide en plus du programme qui fonctionne correctement afin d'obtenir tous les points.

3.4.4. D'autres aspects

Les aspects d'évaluation présentés ci-dessus incluent les objectifs généraux, qui sont généralement mentionnés en tant que besoins d'évaluation de programme. Il y a également d'autres expériences d'évaluation, conçues pour répondre à des problèmes spécifiques concernant l'exécution dynamique des tâches de programmation. Par exemple, dans la plupart des systèmes l'évaluation dynamique est effectuée avec plusieurs ensembles de données de tests, et chacun d'eux est évalué individuellement, à partir de l'état initial et nécessite la terminaison du traitement avant d'effectuer l'évaluation de la valeur de retour. Ce genre d'approche ne permet pas l'évaluation au milieu du traitement, c.-à-d. définissant un cas de test avec un rapport prévu avec l'état du programme créé pendant l'entrée du test précédent. Par exemple, le Quiver (Ellsworth et autres, 2004) fournit à l'enseignant une possibilité de définir la persistance d'état entre les cas de test pour enchaîner plusieurs différents tests ensemble.

Les aspects spécifiques au langage peuvent être difficiles à apprendre et à évaluer. Un bon exemple est la gestion de la mémoire dynamique avec le langage C ++. Pour des raisons concernant la syntaxe du langage et la conception des programmes, les étudiants abusent souvent des fonctions de gestion de la mémoire et des pointeurs, et ne désaffectent pas tous les blocs de mémoire réservée. La bibliothèque de Tutnew fournit des moyens pour évaluer ces aspects simplement en incluant la bibliothèque au programme C++ (Rintala, 2002). La bibliothèque dépasse des méthodes normales de gestion de la mémoire principale et peut fournir ainsi l'évaluation d'exécution pour l'utilisation de la mémoire par le programme.

Les résultats d'évaluation sont imprimés après que le programme soit terminé, bien qu'en cas d'erreur sérieuse de gestion de la mémoire, le programme sera terminé avec un statut d'erreur. C'est un exemple d'un aspect qui est impossible pratiquement à évaluer par un évaluateur humain pour un programme complexe, mais peut être efficacement tracé par l'ordinateur pendant le temps d'exécution.

3.5.Conclusion

Plusieurs approches d'évaluation automatiques des programmes ont été proposées dans la littérature. L'évaluation automatique permet souvent d'offrir un feedback aux enseignants et mêmes aux étudiants, et ce d'une façon plus rapide et plus objective. Cependant, tous les aspects relatifs à la programmation ne peuvent pas être évalués automatiquement. Des approches semi-automatiques peuvent être utilisées pour évaluer certains aspects laissant d'autres à l'évaluation manuelle.

Chapitre 4

La compréhension des programmes

« Apprendre, comprendre : l'un va souvent sans l'autre »

'Charles Régismanset'

4.1. Introduction

Pour juger un programme, l'évaluateur essaye tout d'abord de le comprendre. Cette compréhension se fait soit en analysant son code, ou bien en l'exécutant. S'il arrive à comprendre ce que l'apprenant veut faire dans son programme, l'enseignant pourra le juger facilement. Ce processus de compréhension est essentiel pour maintenir des programmes existants en génie logiciel, où le coût de la maintenance est très élevé. La compréhension d'un programme s'attelle à extraire des informations de haut niveau d'abstraction à partir du code source. Or comprendre ce que d'autres programmeurs ont écrits est une tâche très difficile notamment lorsque le programme n'est pas documenté, ou surtout lorsqu'on n'a pas assez d'informations sur le problème en question. Pour évaluer des apprenants en algorithmique, l'enseignant connaît le problème en question, il a les compétences nécessaires pour le résoudre de plusieurs façons, et il peut comprendre tous les algorithmes de chaque apprenant, la situation est nettement plus favorable. Dans ce chapitre nous présenterons l'une des activités essentielles dans la maintenance logicielle qui est la compréhension de programmes.

4.2.Comprendre le code d'une autre personne

Au cours des dernières décennies, on a assisté à une prolifération de systèmes logiciels dans un large éventail d'environnements de travail. Les secteurs de la société qui ont exploité ces systèmes dans leur fonctionnement au jour le jour sont nombreux et comprennent les industries manufacturières, les institutions financières, les services d'information, services de santé et les industries de la construction **(**Lientz et Swanson, 1980**), (**Tamai et Torimitsu, 1992**).**

Il y a une dépendance croissante vis à vis des systèmes de logiciels (Corbi, 1989) et il est de plus en plus important que ces systèmes fassent le travail qu'ils sont censés faire, et le fasse bien. En d'autres termes, il est essentiel que les systèmes soient utiles. S'ils ne parviennent pas à être utiles, ils ne seront pas acceptés par les utilisateurs et ne seront pas utilisés.

Dans le monde d'aujourd'hui, l'utilisation et le fonctionnement d'un système logiciel approprié peut être une question de vie ou de mort. Certains des facteurs qui influent sur l'utilité de systèmes logiciels sont la flexibilité, la disponibilité continue et le bon fonctionnement **(**Lehman, 1989**)**.

Des changements sont généralement nécessaires pour maintenir ces facteurs au cours de la durée de vie d'un système **(**Sommerville et Thomson 1989**)**. Par exemple, des changements peuvent être nécessaires pour satisfaire les demandes d'amélioration de la performance, l'amélioration fonctionnelle, ou pour faire face aux erreurs découvertes dans le système **(**Lientz et Swanson 1980**).**

Un des plus grands défis auxquels sont confrontés les ingénieurs logiciels est la gestion et le contrôle de ces changements **(**Hoffnagle et Beregi, 1985**)**. Ceci est clairement démontré par le temps et les efforts nécessaires pour maintenir les systèmes logiciels opérationnels après la livraison. Les résultats des études menées pour étudier les caractéristiques et les coûts de modifications effectuées sur un système après sa livraison montrent que les dépenses sont estimées à 40-70 % des coûts de l'ensemble du cycle de vie du système logiciel **(**Alkhatib, 1992), (Blum, 1991), (Boehm, 1981), (Lientz et Swanson,1980**)**. Les ingénieurs logiciels sont confrontés lors de la modification des systèmes logiciels complexes à plusieurs défis :

- La maintenance du logiciel est reconnue comme un domaine clé dans le génie logiciel **(**Arnold, 1994) , (Layzell et Macaulay, 1990**)**.

- Le premier travail de nombreux diplômés entrés dans l'industrie logicielle concerne la maintenance des systèmes existants, plutôt que le développement de nouveaux systèmes **(**Martin et McClure, 1982), (Waters et Chikofsky, 1994**).**

La discipline concernée par les changements liés à un logiciel après sa livraison est traditionnellement appelée la maintenance du logiciel. Une appréciation de la discipline est importante parce que les coûts sont très élevés. De nombreuses questions, y compris la sécurité et le coût, signifient qu'il est urgent de trouver des moyens de réduire ou d'éliminer les problèmes d'entretien **(**Osborne. et Chikofsky, 1990**)**.

Il peut y avoir plusieurs types de changement auxquelles un système logiciel peut être soumis : correctives (en raison de défauts) ; adaptatif (en raison de changements dans son environnement) ; perfectif (pour répondre aux nouvelles exigences) ; et préventive (pour faciliter les futurs travaux de maintenance). Cependant, une activité qui est fondamentale pour un processus de changement efficace, est la compréhension. Avant de mettre en œuvre tout changement, il est essentiel de comprendre le logiciel dans son ensemble et les

programmes touchés par le changement en particulier **(**Brown, 1991), (Corbi, 1989), (VonMayrhauser, 1994), (Pennington et Grabowski, 1990**)**.

Lors de la modification, il s'agit d'avoir une connaissance générale de :

1. Ce que le système logiciel fait et comment il se rapporte à son environnement,
2. Identifier où les changements du système doivent être effectuées, et
3. Une connaissance approfondie de la façon dont les parties à corriger ou modifier le fonctionnent.

La compréhension du programme consomme une part importante des efforts et des ressources de maintenance. À Hewlett Packard, il a été estimé que la lecture du code (un élément fondamental dans la compréhension) coûte 200 millions de dollars par an **(**Padula, 1993**).** Les données de l'industrie et d'autres sources indiquent également que près de la moitié de l'effort total consacré à effectuer des changements est utilisé dans compréhension des programmes **(**Corbi, 1989), (Parikh et Zvegintzov, 1983**)**. Cette dépense a tendance à augmenter dans le cas d'un programmeur modifiant des programmes écrits par quelqu'un d'autre, imprécis, ancien ou même lorsque la documentation du système est inexistante. Malheureusement, ces problèmes ne sont que trop familiers au personnel de maintenance **(**Dean et McCune, 1983)**, (**Padula 1993**).**

4.3.Modèles de processus de compréhension

Les programmeurs varient dans leurs façons de penser, de résoudre des problèmes et dans leur choix des techniques de résolution et des outils employés. En général, cependant, les trois actions impliquées dans la compréhension d'un programme sont : la lecture du sujet du programme, la lecture du code source, et l'exécution **(**Corbi 1989)**.** La Figure 4.1 survole ces actions avec des exemples.

1 Étape 1 - Lisez sur le programme : A ce stade du processus, «Celui qui va comprendre» navigue, parcourt différentes sources d'information telles que la documentation du système - Spécification et documents de conception – afin de développer une vue d'ensemble ou une compréhension globale du système. L'aide de documentation tel que des graphiques et des données de structure et des diagrammes de flux de contrôle peuvent être utilisés. Dans de nombreux systèmes importants, complexes et anciens (développés avant l'avènement de bons

outils de documentation, techniques et pratiques), cette phase peut être omise si la documentation du système est inexacte, obsolète ou inexistante.

2 Étape 2 - Lire le code source : Au cours de cette étape, les vues globales et locales du programme peuvent être obtenues. Le point de vue global est utilisé pour acquérir une compréhension de haut niveau du système et également déterminer la portée qu'un changement pourrait avoir sur d'autres parties du système. La vue locale permet aux programmeurs de concentrer leur attention sur une partie spécifique du système. Avec ce point de vue, l'information sur la structure du système, les types de données et les modèles algorithmiques est obtenue. Des outils tels que des analyseurs statiques - utilisés pour examiner le code source - sont utilisés au cours de cette phase. Ils produisent des listes de références croisées, qui indiquent où des identifiants différents - les fonctions, les procédures, les variables et constantes - ont été utilisés (ou appelés) dans le programme. De cette façon, ils peuvent mettre en évidence des anomalies dans le programme et donc permettre au programmeur de détecter les erreurs. Gardant à l'esprit que la documentation du système peut ne pas être fiable, la lecture du code source d'un programme est généralement le principal moyen d'obtenir des informations sur un produit logiciel.

3 Étape 3 - Exécutez le programme : L'objectif de cette étape est d'étudier le comportement dynamique du programme, y compris, par exemple, l'exécution du programme et l'obtention de données de trace. Le bénéfice de l'exécution du programme est qu'il peut révéler certaines caractéristiques du système qui sont difficiles à obtenir par la simple lecture du code source.

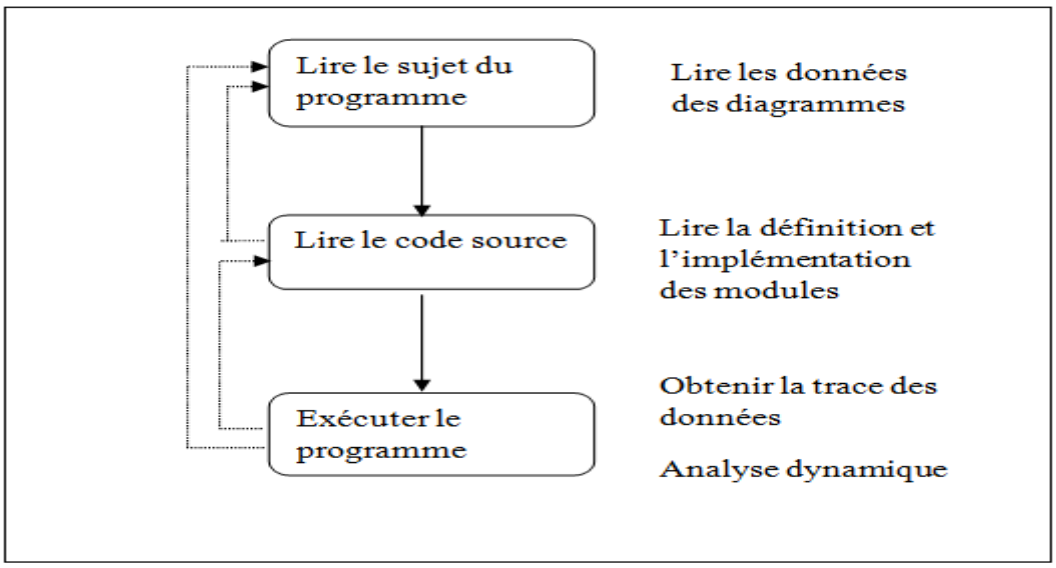

Figure 4. 1 Modèle de processus de compréhension

Il est généralement admis que la maintenance passe par les étapes décrites ci-dessus (indépendamment de l'ordre) dans une tentative de comprendre un programme. Ces étapes permettent la construction d'un modèle mental - représentation interne - du programme **(**Johnson-Laird 1983**)**.

4.3.1. Modèle mental

Notre compréhension d'un phénomène dépend dans une certaine mesure de notre capacité à former une représentation mentale, qui sert de modèle de fonctionnement du phénomène à comprendre **(**Johnson-Laird 1983**)**.

Le phénomène (comment un ordinateur fonctionne, le comportement des liquides, un algorithme) est connu comme le système cible, et sa représentation mentale est appelé un modèle mental. Par exemple, si vous comprenez comment fonctionne un ordinateur, alors vous avez un modèle mental qui le représente et, sur la base de ce modèle, vous pouvez prédire ce qui se passera lorsque l'ordinateur sera allumé. En utilisant le modèle, vous pouvez aussi expliquer certaines observations telles que l'apparition d'une image bleue ou le blocage de l'ordinateur.

L'exhaustivité et la précision du modèle dépend dans une large mesure des informations dont les utilisateurs ont besoin. Dans le cas d'un ordinateur, un utilisateur ordinaire - qui l'utilise uniquement pour le divertissement n'aura pas à comprendre la composition interne de l'écran et de l'unité centrale et comment ils fonctionnent, afin d'être en mesure de l'utiliser.

Un technicien, cependant, qui dessert l'ensemble en cas de panne a besoin d'une meilleure compréhension de comment un ordinateur fonctionne, et nécessite un modèle mental plus élaboré et précis.

Le contenu et la formation des modèles mentaux dépendent des structures cognitives et des processus cognitifs. Le modèle mental est formé après l'observation, l'inférence ou une interaction avec le système cible. Il se change continuellement quand plus d'informations sur le système cible sont acquises. Sa complétude et son exactitude peuvent être influencées par des facteurs tels que l'expérience précédente de l'utilisateur avec des systèmes similaires et une formation technique **(**Johnson-Laird, 1983), (Lindsay et Norman 1977**)**.

Le modèle mental peut contenir des informations insuffisantes, contradictoires ou inutiles sur le système cible. Bien qu'il n'est pas nécessaire pour que ce modèle soit complet, il doit

transmettre des informations essentielles sur le système cible. Par exemple, s'il modélise un morceau de logiciel, il devrait au moins incarner la fonctionnalité du logiciel.

4.3.2. Stratégies de compréhension du programme

Une stratégie de compréhension de programme est une technique utilisée pour former un modèle mental du programme cible. Le modèle mental est construit en combinant les informations contenues dans le code source et la documentation à l'aide de l'expertise et des connaissances de domaine que le programmeur produit. C'est la représentation courante qu'a le programmeur sur le programme.

Un certain nombre de modèles descriptifs de la façon dont les programmeurs utilisent lors de la compréhension de programmes, ont été proposés sur la base des résultats des études empiriques de programmeurs.

Des exemples de ces modèles suivent des démarches : de haut en bas ou top-down **(**Brooks, 1983), (Hoc et al, 1990**)**, de bas en haut ou bottom-up (Shneiderman et Mayer, 1979), (Shneiderman, 1980) et des modèles opportunistes **(**Hoc et al, 1990), (Pennington et Grabowski, 1990**).**

4.3.2.1.Modèle top-down

Le principe de ce modèle est que celui qui comprend commence par comprendre les détails de haut niveau d'un programme , comme ce qu'il fait quand il s'exécute , et travaille à la compréhension des détails de bas niveau tels que les types de données , le contrôle et les flux de données et progressivement les modèles algorithmiques dans un mode de haut en bas. Un exemple d'un modèle de compréhension de haut en bas est celle proposée par Brooks **(**Brooks, 1983**)**. Les principales caractéristiques du modèle de Brooks sont:

Il considère la structure de la connaissance à comprendre organisée en domaines distincts reliant le domaine du problème (Représenté par la fonctionnalité du système) et le domaine de la programmation (représenté par le programme) ;

La compréhension du programme consiste à reconstruire les connaissances sur les deux domaines cités précédemment et la relation entre eux. Le processus de reconstruction est de haut en bas, impliquant la création, la confirmation et le raffinement des hypothèses sur ce qu'est un programme et comment il fonctionne.

La structure cognitive et processus cognitif d'un modèle mental résultant d'une stratégie top-down peuvent être expliqués en termes de métaphore de conception. Le développement de logiciels dans son intégralité peut être considéré comme une tâche de conception qui se compose de deux processus fondamentaux - la composition et la compréhension **(**Pennington et Grabowski, 1990**)**. La composition représente la production d'une conception et la compréhension est l'acte de comprendre cette conception.

La composition implique le passage de ce que le programme fait dans le domaine du problème, dans une collection d'instructions informatiques de la façon dont cela fonctionne dans le domaine de la programmation, en utilisant un langage de programmation. La Figure 4.2 montre des exemples de domaines de connaissances qui peuvent être rencontrées au cours de ce processus et la façon dont ils sont liés.

La compréhension est l'inverse de la composition. C'est une transformation du domaine de programmation en domaine du problème impliquant la reconstruction de connaissances sur ces domaines (y compris les domaines intermédiaires) et la relation entre eux. Le processus de reconstruction est concerné par la création, la confirmation et le raffinement successif des hypothèses.

Il commence avec la perception d'une hypothèse vague et générale, connu comme l'hypothèse primaire. Elle est confirmée et affiné avec l'acquisition de plus d'informations sur le système à partir du texte de programme et d'autres sources telles que la documentation du système.

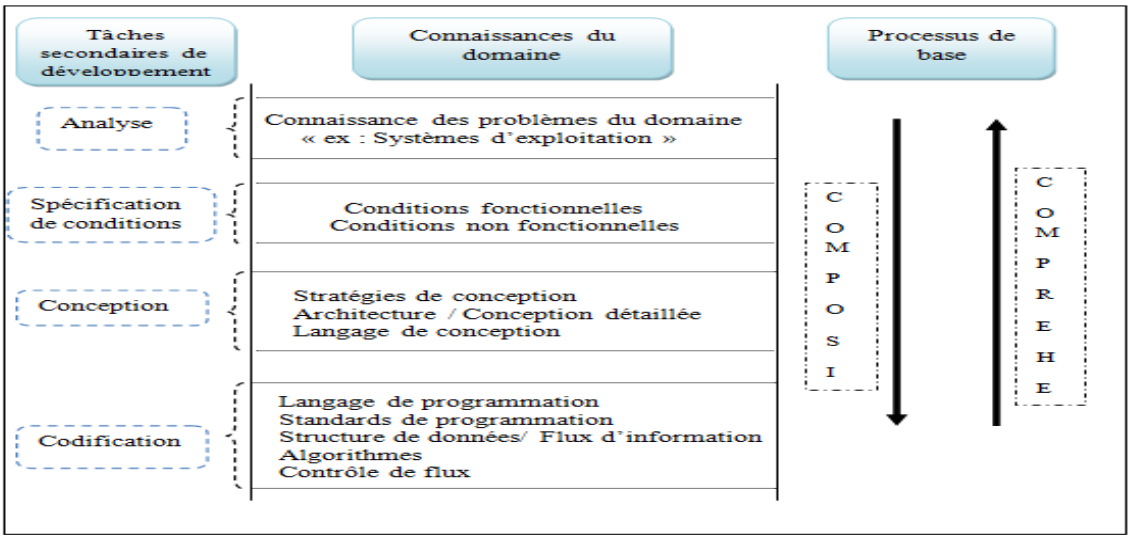

Figure 4.2 Domaines de connaissance rencontrés pendant la compréhension

La première hypothèse est généralement générée dès que le programmeur rencontre des informations concernant tous les aspects du programme, par exemple un nom de module. Ainsi, le modèle mental du programme commence à être construit au début, avant même que le programmeur prend conscience des détails sémantiques et syntaxiques de bas niveau du programme.

Les informations nécessaires à la formulation d'hypothèses et du raffinement se manifestent dans les principales caractéristiques - internes et externes pour le programme - appelées balises (Beacons) qui servent comme indicateurs typiques de la présence d'une structure ou d'une opération (Brooks, 1983) notamment. Quelques exemples de balises sont donnés dans le tableau 4.1. L'utilisation de cette approche pour comprendre un programme est évocatrice de l'écrémage d'un morceau de texte pour obtenir une compréhension générale, de haut niveau **(**Wiedenbeck, 1986**)**, puis en relisant le texte en détail pour mieux comprendre.

Numéro	***Désignation***
Interne au programme	
1	Commentaires, inclusion de données et des dictionnaires de variables
2	Variable, structure, procédure et les noms des labelles.
3	Déclaration ou division de données
4	Les commentaires d'interligne
5	Impression
6	Sous-routine ou Structure de module
7	Formats d'E/S, en-tête, dispositif ou tâches de canal
Externe au programme	
1	Manuels d'utilisateurs
2	Manuels du programme logique
3	Organigrammes
4	Edition des descriptions des algorithmes et des techniques

Tableau 4.1 Balises du programme « Adapté de (Brooks, 1983), p 55 »

4.3.2.2. Modèle Bottom-up

En utilisant cette stratégie, le programmeur reconnaît successivement les modèles dans le programme. Ceux-ci sont regroupés d'une façon itérative en structures de haut niveau, sémantiquement plus significatives **(**Basili et Mills, 1982), (Pennington, 1987), (Shneiderman, 1980**)**, Les structures de haut niveau sont ensuite regroupées en séquences ensemble dans des structures encore plus grandes dans un mode bottom-up répétitive jusqu'à ce que le programme entier soit compris. Voir Figure 4.3 pour une représentation schématique du modèle bottom-up.

Le processus de regroupement tend à être plus rapide pour les programmeurs plus expérimentés que les novices car ils reconnaissent des modèles (pattern) plus rapidement. Par exemple, les déclarations du programme suivant:

```
MaxValue := Table[1]
For Index := 2 To 100 Do
        IF Table[Index] > MaxValue THEN
                MaxValue = Table[Index]
        EndIF
EndFor
```

Seraient regroupés par un programmeur expérimenté dans un morceau «trouver élément maximum dans le tableau.

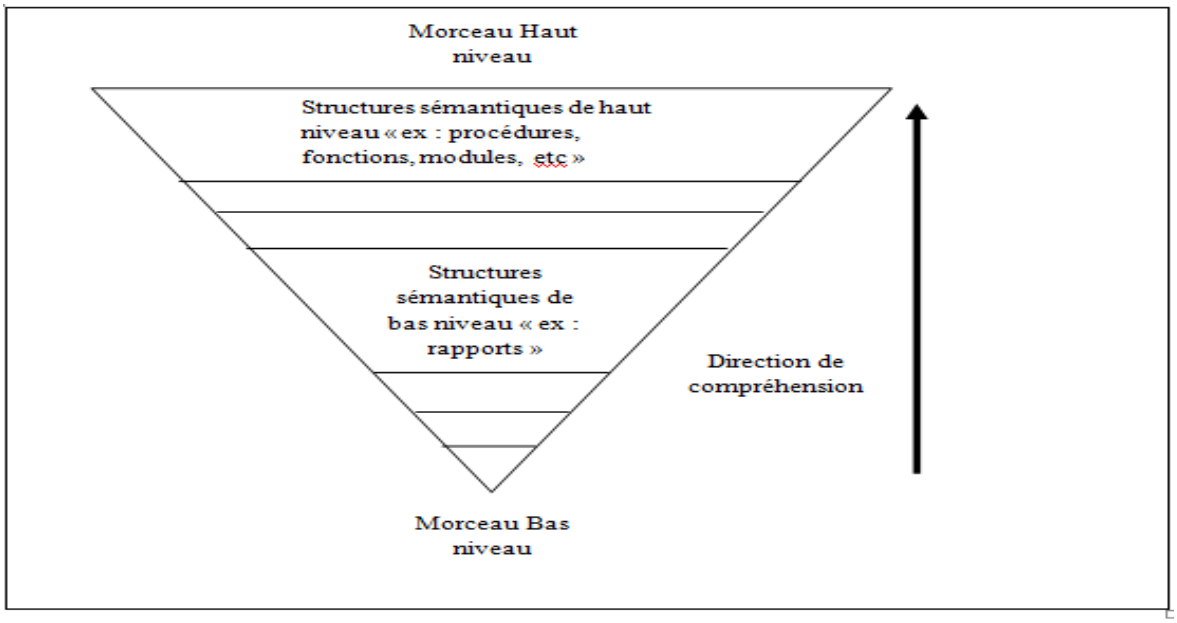

Figure 4.3 Processus de compréhension Botton-up

Les principales faiblesses à la fois des stratégies de compréhension top-down et bottom-up sont :

1. Absence de la contribution que d'autres facteurs tels que les outils d'assistance disponibles font à la compréhension. ; et

2. Le processus de la compréhension d'un programme se déroule rarement de façon bien définie que ces modèles décrivent. Au contraire, les programmeurs ont tendance à profiter de tous les indices qu'ils rencontrent de façon opportuniste.

4.3.2.3. Modèle opportuniste:

En utilisant ce modèle, celui qui comprend utilise à la fois les deux approches ascendante (bottom-up) et descendante (top-down), mais pas simultanément.

" ... L'entendeur humain est mieux considéré comme un processeur opportuniste capable d'exploiter à la fois les signaux ascendants et descendants quand ils deviennent disponibles " Letovsky **(**Letovsky, 1986**)** pp.69-70.

Selon ce modèle, la compréhension s'articule autour de trois éléments clés et complémentaires - une base de connaissances, un modèle mental et un processus d'assimilation :

1. **Une base de connaissances** : Il s'agit de l'expertise et des connaissances de base que celui qui comprend apporte à la tâche de compréhension.

2. **Un modèle mental**. Cela exprime la compréhension actuelle du programme cible du programmeur.

3. **Un processus d'assimilation**: Ceci décrit la procédure utilisée pour obtenir des informations à partir de diverses sources telles que le code source et la documentation du système.

Lorsque des développeurs veulent comprendre un morceau de programme, le processus d'assimilation leur permet d'obtenir des informations sur le système. Cette information déclenche alors l'invocation de plans à partir de la base de connaissances pour leur permettre de former un modèle mental du programme à être compris. Comme indiqué précédemment, le modèle mental se change continuellement quand une information est obtenue.

4.4. Outils pour la compréhension et l'ingénierie inverse

La compréhension de programme et l'ingénierie inverse ont été combinées, car ils sont étroitement liés.

La compréhension du programme implique d'avoir une connaissance générale de ce qu'est un programme et comment il se rapporte à son environnement ; identifier où dans le système les changements doivent être effectuées ; et savoir comment les différents composants à modifier fonctionnent. L'ingénierie inverse va encore plus loin en permettant l'analyse et différentes représentations du système pour promouvoir cette compréhension.

En raison de la grande quantité de temps utilisée pour étudier et comprendre les programmes **(**Corbi, 1989), (Parikh et Zvegintzov, 1983**)**, des outils favorisant la compréhension jouent un rôle majeur dans la mise en œuvre du changement. Les outils pour les tâches de l'ingénierie inverse et services connexes tels que redocumentation inverse, reprise de la conception, la récupération de la spécification et la réingénierie atteignent également le même objectif.

La plupart des outils de cette catégorie sont des outils de visualisation, c'est-à dire des outils qui aident le programmeur à former un modèle mental du système en cours d'examen en vertu de l'impact visuel qu'ils créent. Ci-dessous nous présenterons quelques outils utilisés pour la compréhension de programme et l'ingénierie inverse.

4.4.1. Slicer

Un des problèmes majeurs avec la maintenance du logiciel fait est la taille du code source de programme.

Il est important qu'un programmeur puisse choisir et regarder seulement les parties du programme qui sont affectées par un changement proposé sans être perdus par les parties non pertinentes. Une technique qui aide pour ce problème est connue comme Slicing ou découpage en tranches - un processus mécanique marquant toutes les sections d'un code qui peut influencer la valeur d'une variable à un point donné dans le programme **(**Weiser, 1984**)**.

L'outil utilisé pour soutenir le découpage en tranches est connu sous le nom de Slicer **(**Gallagher, 1990**)**. Le slicing montre également des liaisons de transmission de données et

des caractéristiques relatives pour permettre au programmeur de dépister l'effet des changements.

4.4.2. Analyseur statique

Pour comprendre un programme, il est généralement nécessaire d'obtenir des informations sur différents aspects du programme tels que les modules, les procédures, les variables, les éléments de données, les objets et les classes, et la hiérarchie de classe. Un analyseur statique permet la dérivation de cette information par un examen attentif et profond du pseudo code.

Certains auteurs se réfèrent également à ce type d'outil comme un « navigateur » **(**Oman, 1990**)**. Généralement, un analyseur statique: permet la visualisation générale du texte de programme, sert comme navigateur, génère des résumés de contenu et d'utilisation des éléments sélectionnés dans le texte du programme tels que des variables ou des objets.

4.4.3. Analyseur dynamique

Lorsqu'on étudie un système logiciel dans le but de le changer, l'examen seul du texte du programme - analyse statique - peut ne pas fournir toutes les informations nécessaires. Il existe un besoin de contrôler et d'analyser les différents aspects du programme quand il est exécuté. Un outil qui peut être utilisé pour soutenir ce processus est connu comme un analyseur dynamique. En général, l'analyseur dynamique permet à celui qui comprend de retracer le chemin d'exécution du système en cours d'exécution - il agit comme un traceur. Cela permet au mainteneur de déterminer les voies qui seront touchés par un changement et celles par lesquelles un changement doit être effectué.

4.4.4. Analyseur du flux de données

Un analyseur de flux de données est un outil d'analyse statique qui permet à celui qui comprend de suivre tous les chemins possibles des flux de données et des flux de contrôle dans le programme et également de revenir en arrière **(**Vanek et Davis, 1990**)**. Cela est particulièrement important lorsqu'il existe un besoin d'une analyse de l'impact : étude de l'effet d'une modification sur les autres parties du système. Par le suivi des flux de données et de contrôle, le mainteneur peut obtenir des informations telles que l'endroit où une variable a obtenu sa valeur et quelles parties du programme sont affectées par la modification de la variable.

En général, un analyseur de flux de données :

1. Permet l'analyse du texte du programme pour promouvoir la compréhension de la logique sous-jacente du programme;

2. Aide à la présentation de la relation entre les différentes composantes du système;

3. Fournit des impressions qui permettent à l'utilisateur de sélectionner et d'afficher différentes vues du système.

4.4.5. Cross – Referencer Référenceur

Le référenceur est un outil qui génère un index de l'utilisation d'une entité du programme. Par exemple, il peut produire des informations sur les déclarations d'une variable et toutes les sections du programme dans lequel il a été créé et utilisé. Au cours de l'implémentation d'un changement, les informations que cet outil génère aide le mainteneur à se concentrer et se focaliser sur les parties du programme qui sont touchées par le changement.

4.4.6. Analyseur de dépendances

Un analyseur de dépendance permet au mainteneur d'analyser et de comprendre les relations entre les entités d'un programme. Cet outil est particulièrement utile dans les situations où les entités liées logiquement, comme les variables, peuvent être physiquement éloignés dans le programme. En général, l'analyseur de dépendance :

1. Peut fournir des possibilités qui permettent à un mainteneur de mettre en place et d'interroger une base de données des dépendances dans un programme. Les informations sur les dépendances peuvent également être utilisées pour déterminer l'effet d'un changement et pour identifier des relations entre des entités redondantes **(**Wilde, 1990**)**.

2. Fournit une représentation sous forme de graphe des dépendances dans un programme où le noeud dans le graphe représente une entité de programme et un arc représente la dépendance entre les entités.

4.5.[2] Outil de transformation

Un outil de transformation convertit les programmes entre différentes formes de représentations, généralement entre le texte et les graphiques, par exemple, la transformation du code pour une forme visuelle et vice versa **(**Rajlich 1990**)**. En raison de l'incidence que peuvent avoir des représentations visuelles sur la compréhension, l'utilisation d'un outil de transformation peut contribuer à la vue de soutien et permettre de comprendre le système d'une manière qui ne serait pas possible avec la représentation textuelle par exemple. L'outil vient habituellement avec un navigateur et un éditeur, qui sont utilisés pour modifier le programme dans l'une de ses représentations.

4.6.Conclusion

La compréhension de programmes est une tâche essentielle dans la phase de maintenance des applications existantes. Elle consiste à analyser le code écrit par d'autres personnes pour en extraire des informations de haut niveau d'abstraction. Généralement, les développeurs utilisent ces informations pour mettre à jour le code existant, le documenter, etc. Dans ce manuscrit nous allons utiliser ces informations pour évaluer des apprenants.

Deuxième partie

Problème et proposition

Chapitre 5

Une évaluation basée sur la compréhension des algorithmes

« Comprendre c'est avant tout unifier »

‘Albert Camus’

5.1. Introduction

Les systèmes d'évaluation des productions de l'apprenant, souffrent généralement d'une modélisation soit trop rigide, les destinant à l'apprentissage d'un champ disciplinaire déterminé a priori, soit trop grossière, ne permettant pas la construction d'un modèle suffisamment riche.

L'évaluation automatique des algorithmes est difficile. Cette difficulté est due généralement à la nature de la matière, les étudiants n'ont pas de modèle empirique et spontané, comme en physique où l'objectif est de comprendre des phénomènes qu'on connait au préalable (Ben-Ari, 1998), (Guibert et al, 05). Les personnes essayant d'écrire des algorithmes doivent raisonner comme une machine, or la plupart de ces personnes n'ont pas suffisamment de connaissances. Comme exemple de ce problème nous prenons les instructions de lecture et écriture : Lire (variable) ; Ecrire (variable) ; En effet, ces deux instructions peuvent créer une confusion chez l'apprenant.

Une autre caractéristique importante de l'algorithmique est l'abstraction, où l'apprenant doit abstraire les stratégies permettant la résolution d'un problème donné, avant de les traduire en instructions. En d'autre terme, l'apprenant résout le problème mentalement, en suite il traduit la solution en algorithmes. En plus, il faut prendre en compte qu'un problème peut avoir plusieurs solutions. Dans les sections qui suivent nous proposerons une approche d'évaluation des apprenants en algorithmique.

5.2.Evaluation automatique des algorithmes

Il n'existe pas de méthode d'évaluation standard en algorithmique. Certaines techniques telles que : QCM, Question à Trous, Question Ouvertes, etc., bien qu'elles soient efficaces dans certaines domaines, sont inadéquates pour l'algorithmique. Parmi les méthodes d'évaluation utilisées en algorithmiques :

1. Tester l'algorithme proposé avec des données de test, si le résultat est bon alors l'algorithme l'est aussi et inversement. Cette technique présente les inconvénients suivants :
   - Elle donne une vision globale sur la solution, on ne peut pas connaître les détails.
   - On ne peut pas vérifier des critères de qualité telles que la lisibilité, l'optimisation, etc.

2. Donner un algorithme contenant des lignes vides et demander aux apprenants d'ajouter les instructions manquantes. Cette technique limite la créativité et n'offre pas aux apprenants la possibilité d'engager réellement leurs compétences.

En présentiel, les enseignants demandent aux apprenants d'écrire des algorithmes, cela leur permet d'être créatifs et d'exprimer leurs idées. Malgré la pertinence de cette méthode d'évaluation, comprendre ce que chaque apprenant a fait est une tâche difficile. D'un autre coté, sa mise en œuvre et son automatisation représentent un challenge.

Le problème central est le fait que nous ne pouvons pas prévoir les solutions proposées par les apprenants. Que ce soit sur machine ou bien sur papier les évaluateurs essayent toujours de comprendre les propositions des apprenants. Un programme qui ne fonctionne pas, peut contenir de bonnes idées.

5.3.Vers une compréhension des algorithmes

La façon traditionnelle d'évaluer des algorithmes, *consiste à vérifier si l'algorithme proposé est bon ou mauvais,* la figure ci-dessous illustre cette vision de l'évaluation.

Figure 5.1 Evaluation des algorithmes

Après avoir jugé un algorithme, l'évaluateur lui attribue une note. Cependant une note ne peut pas refléter les compétences/faiblesse des apprenants. Cette vision de l'évaluation n'est pas détaillée puisqu'elle nous donne une idée globale sur l'apprenant.

En algorithmique, évaluer ne se limite pas à attribuer une note. En fait, il s'agit de vérifier du savoir-faire. Vérifier des compétences s'avère difficile et donne un caractère spécial à l'évaluation. Donc, pour évaluer des apprenants en algorithmique, il serait intéressant de comprendre ce qu'un apprenant a fait dans son algorithme, ou plutôt ce qu'il veut faire ?

Figure 5.2. Compréhension des algorithmes

En ayant une idée sur les intentions des apprenants, nous pouvons les aider à atteindre leurs objectifs, et surtout nous pouvons localiser les obstacles qui leur empêchent d'y arriver. Nous pouvons même localiser les erreurs qu'ils ont commisses, et aussi les compétences qu'ils ont acquises. Ainsi, l'évaluation sera plus détaillée, et deviendra pertinente. Cependant comprendre ce que d'autres personnes ont écrit est une tache difficile, en particulier quand le code n'est pas documenté, ou surtout quand on n'a pas assez d'informations sur le problème.

Dans notre cas, pour évaluer les productions des apprenants, l'enseignant connait le problème, et peut le résoudre de plusieurs manières. En plus, il est capable de comprendre toutes les solutions proposées. Par conséquent, la situation est plus favorable.

5.4. Objectifs

En algorithmique, il est possible de résoudre les problèmes de plusieurs façons, certaines sont bonnes, d'autres sont mauvaises. La majorité des systèmes d'évaluation existants, évaluent les algorithmes par des méthodes qui vérifient les sorties correctes, ou par des mesures sur le code, tels que sa complexité. Toutes ces méthodes ne tiennent pas compte de la manière dont un problème a été résolu. Elles génèrent des notes égales pour les bons et mauvais algorithmes, à condition qu'ils produisent les mêmes sorties.

Notre objectif est de proposer une approche qui analyse les algorithmes en utilisant une méthodologie différente des systèmes existants (Bouacha et Bensebaa). Il ne s'agit pas d'un système qui agit comme un remplacement pour un évaluateur humain, puisqu'il ya des aspects de l'évaluation des algorithmes qui ne sont pas appropriés pour l'automatisation tels que les commentaires et l'interface utilisateur. Au lieu de cela, il servira à faciliter la tâche d'évaluation des algorithmes. Il analysera les algorithmes et fournir rapidement un feedback. L'évaluateur sera alors en mesure d'utiliser ce feedback pour prendre des décisions pédagogiques. Notre objectif principal est de libérer l'évaluateur de la concentration sur l'attribution des notes, d'accélérer le processus d'évaluation et d'améliorer la précision de l'évaluation à l'aide de l'automatisation.

5.5.Propositions

Pour avoir un outil d'évaluation qui comprenne les algorithmes, nous avons commencé avec l'idée suivante : nous allons associer à chaque problème que nous proposons aux

apprenants, toutes les propositions possibles, y compris les propositions fausses mais ayant une utilité pédagogique.

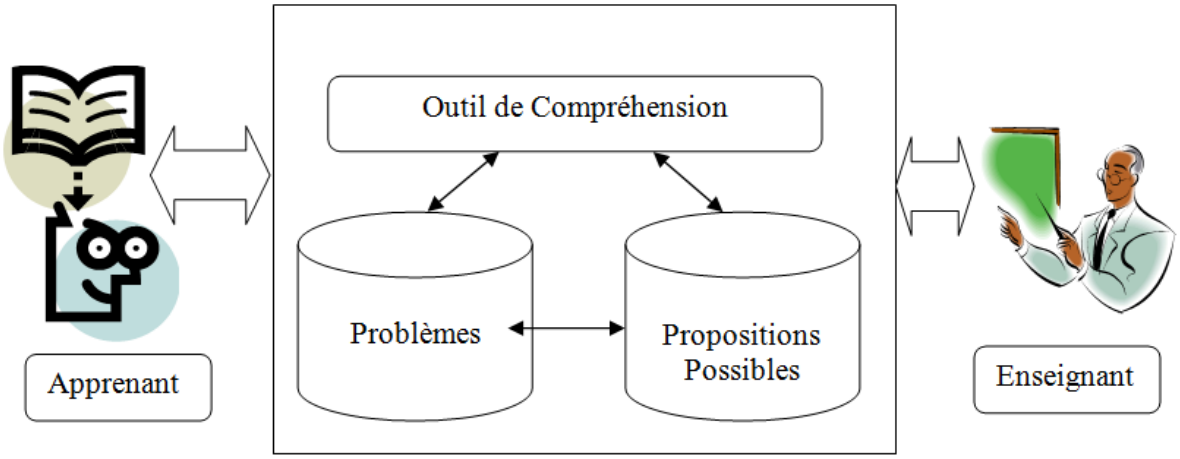

Figure 5.3 Architecture générale de l'outil d'évaluation version 0

Cette première architecture est composée de trois composantes principales :

1. *Base de problèmes* : contient des problèmes utilisés pendant l'évaluation.
2. *Base de propositions* possibles: contient les propositions possibles correspondantes aux problèmes. Ces propositions peuvent être correctes, ou incorrectes ayant une utilité pédagogique.
3. *Outil de compréhension* : permet de comprendre les propositions des apprenants.

Avant d'aller plus loin, il faut prendre en considération les problèmes suivants :

1. Comment différencier entre des propositions identiques sur le plan comportemental, mais qui se différencient au niveau de l'ordre des opérations, par exemple :

X← 1	et	Y←2
Y← 2		X←1

Ou bien, qui se différencient dans le nommage des variables, par exemple :

X, Y		A, B
X← 1	et	A← 1
Y← 2		B← 2

2. Il n'est pas possible de prévoir toutes les propositions possibles, on en oublie toujours.
3. Comment exprimer un algorithme?
4. Quelles sont les informations que nous allons capturer ?

Nous pouvons améliorer notre idée en utilisant des modèles de propositions, et nous allons associer les apprenants vers ces modèles. Pour automatiser la compréhension des algorithmes, l'idée consiste à utiliser des modèles préconstruits. Chaque modèle représente une proposition (Algorithme) qui résout un problème donné. Ces modèles sont crées en utilisant un éditeur d'algorithmes, et ils sont utilisés comme base pour reconnaitre les propositions des apprenants. Cette reconnaissance s'appuie sur des techniques de compréhension de programmes, elle est faite par notre outil d'évaluation. Envisager les modèles de toutes les propositions possibles est quasiment impossible, même pour l'expert le plus avisé. Nous profitons des anciennes évaluations pour extraire les modèles de propositions corrects et incorrects considérés pédagogiquement utiles. Notre système devient comme suit :

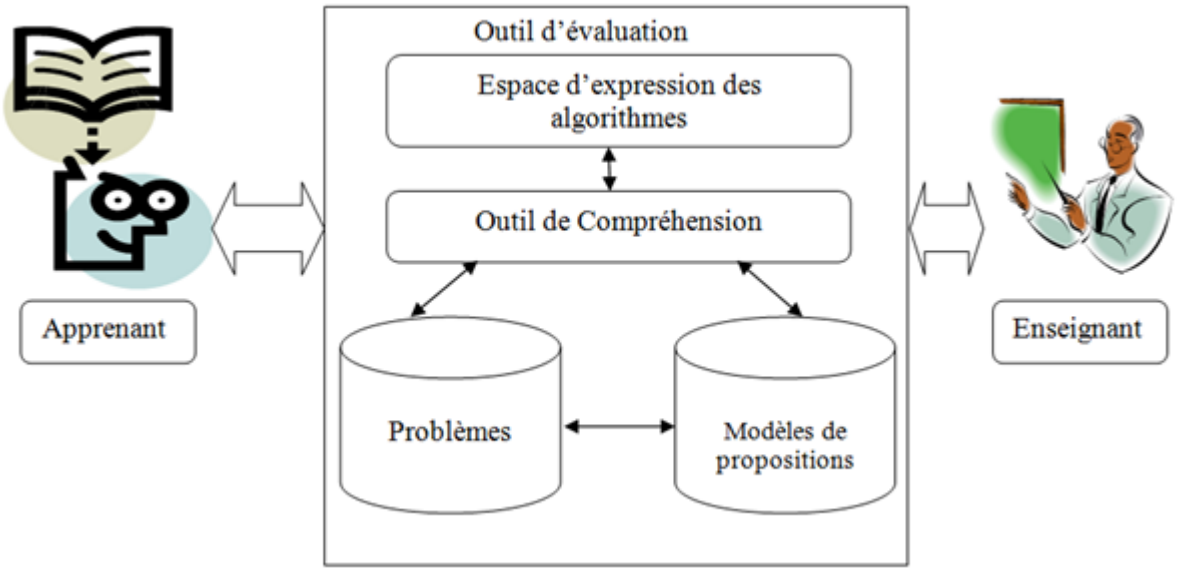

Figure 5.4 Architecture générale de l'outil d'évaluation version 1

Une quatrième composante est ajoutée à notre système, c'est l'espace d'expression des algorithmes. Il permet aux apprenants d'éditer un algorithme, et aux enseignants d'éditer des modèles.

5.6.Conceptualisation

Pour comprendre nos idées et propositions, nous allons définir les concepts que nous allons utiliser tout au long du reste de ce manuscrit.

5.6.1. Instruction Est l'élément de base qui compose un algorithme. Nous distinguons plusieurs types d'instructions : lecture, écriture, conditions, itération, etc.

Instruction	Description
A= B	Affectation
Lire(A)	Lecture
Ecrire(A)	Ecriture
Si Condition *Alors* Action	Conditions
Pour ; Tant que	Répétition

Tableau 5.1 Instructions

5.6.2. Stratégie C'est la façon avec laquelle sont combinées des instructions pour résoudre un problème donné. C'est la démarche de résolution, elle peut être composée de plusieurs étapes, chacune est appelée tâche.

5.6.3. Tâche C'est une étape d'une stratégie. Elle est composée d'instructions.

5.6.4. Algorithme C'est une **c**ombinaison d'instructions selon une stratégie précise.

5.6.5 Modèle Un modèle est un algorithme d'une future proposition qui représente une stratégie *validé* par un expert.

En résumé, un algorithme (proposition ou modèle) est une combinaison d'instructions. Cette combinaison peut être faite en adoptant une stratégie qui définit les étapes nécessaires pour la résolution du problème. Chaque étape correspond à une partie de l'algorithme (certaines instructions), nous l'avons appelée *tâche*. Pour mieux comprendre ces concepts, nous utiliserons l'exemple suivant. Nous avons demandé aux étudiants d'écrire un algorithme qui teste si les éléments d'un tableau (d'entiers) sont consécutifs (se suivent) ou pas.

Exemple : 3, 4, 5, 6 sont consécutifs

1, 3, 4, 5 ne sont pas consécutifs

```
1.  Algorithme modele 7;
2. Var          Nombre : i;
3.              Nombre : j;
4.              Tableau de nombres : t[10];
5.              Booleen : Consec;
6. Debut
7.    For (i = 0; i<10; 1)
8.    BeginFor
9.        Read(t[i]);
10.   EndFor
11.   j=0;
12.   Consec=Vrai;
13.   While ( j<9ANDConsec== Vrai )
14.   BeginWhile
15.       If ( t[j]+1!=t[j+1] ) Then
16.       BeginIf
17.           Consec=Faux;
18.       EndIf
19.       j=j+1;
20.   EndWhile
21.   If ( Consec==Vrai ) Then
22.   BeginIf
23.       Write(Les elements du tableau sont consecutifs);
24.   Else
25.       Write(Les elements du tableau ne sont pas consecutifs);
26.   EndIf
27. Fin
```

Figure 5.5 Exemple d'algorithme

Par exemple notre problème peut être résolu à l'aide des tâches suivantes : lecture (lignes : 7→10), Initialisation (lignes : 11→12), Test (lignes : 13→20) et Résultat (lignes : 21→26). Dans le chapitre suivant, nous présenterons d'autres concepts en décortiquant cet exemple.

5.7.Evaluation basée compréhension

Pour évaluer automatiquement des apprenants en algorithmique, la première étape consiste à leur (apprenants) demander de résoudre des problèmes en écrivant des algorithmes en pseudo code, voir la figure ci-dessous. Les problèmes sont stockés dans une base de problèmes construites par des experts en algorithmique. Comme présenté dans les sections précédentes, nous ne pouvons pas prévoir toutes les propositions possibles. Par conséquent, les stratégies que les apprenants utilisent se différencient, on peut se retrouver face à de nombreuses propositions.

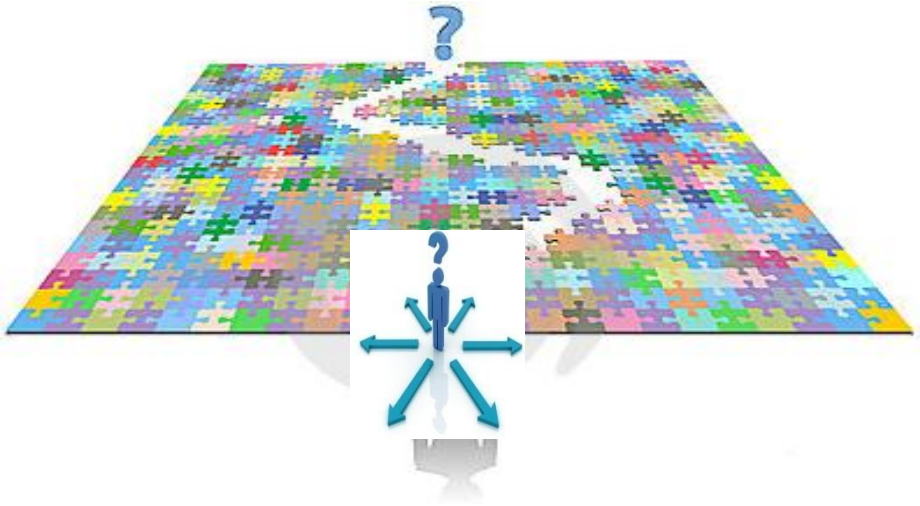

Figure 5.6 Evaluation : Essayer de résoudre un problème

Lorsque les enseignants ajoutent des problèmes, ils doivent spécifier pour chaque problème les modèles de propositions correspondants (qui peuvent être correctes ou incorrectes). Ces modèles de propositions construisent la base de modèles.

Lors de l'évaluation, l'apprenant propose une solution d'un problème sous forme d'un algorithme en pseudo code. Une fois les propositions des apprenants obtenues, l'enseignant peut lancer l'opération d'évaluation, qui se base sur un processus de compréhension des propositions. Cette compréhension consiste à déduire la stratégie adoptée par l'apprenant pour résoudre le problème en question, et ce en analysant l'algorithme proposition, voir la figure ci-dessous.

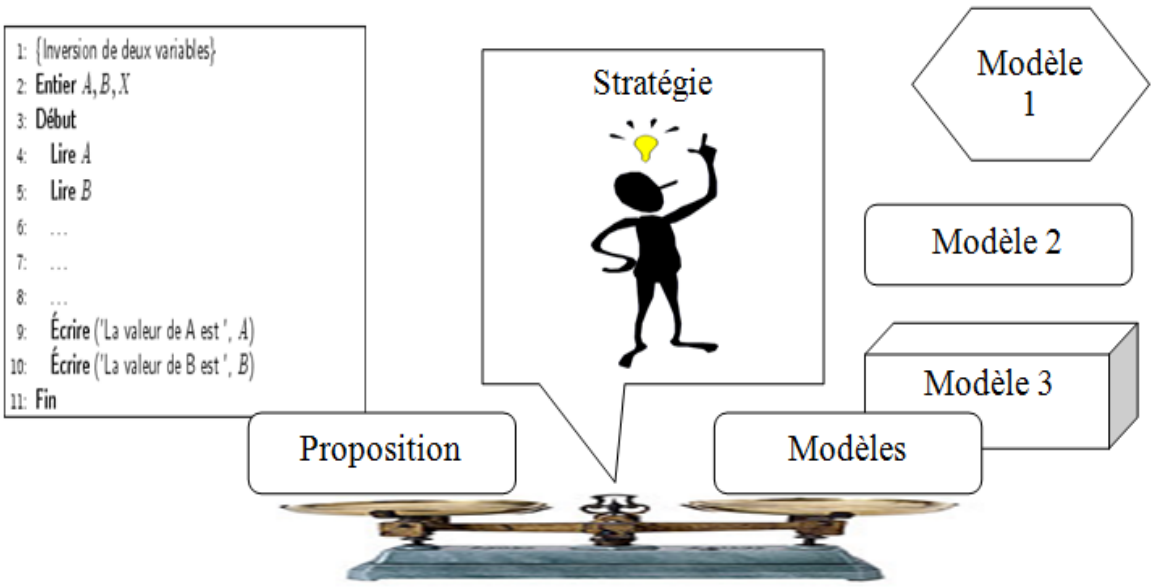

Figure 5.7 Compréhension d'une proposition

L'analyse de l'algorithme permet de comprendre ce qu'un apprenant veut faire, en comparant sa proposition avec la base de modèle. Cette compréhension ne se limite pas à une simple comparaison mais plutôt, elle s'appuie sur des techniques de compréhension de programmes, voir le chapitre suivant.

A la fin de ce processus d'évaluation basée compréhension, deux cas de figure sont possibles : soit la solution est reconnue, ou bien elle ne l'est pas. Dans ce dernier cas on va opter pour une *suspension* de l'évaluation, c'est-à-dire qu'on doit attendre l'intervention de l'expert humain, voir la figure ci-dessous.

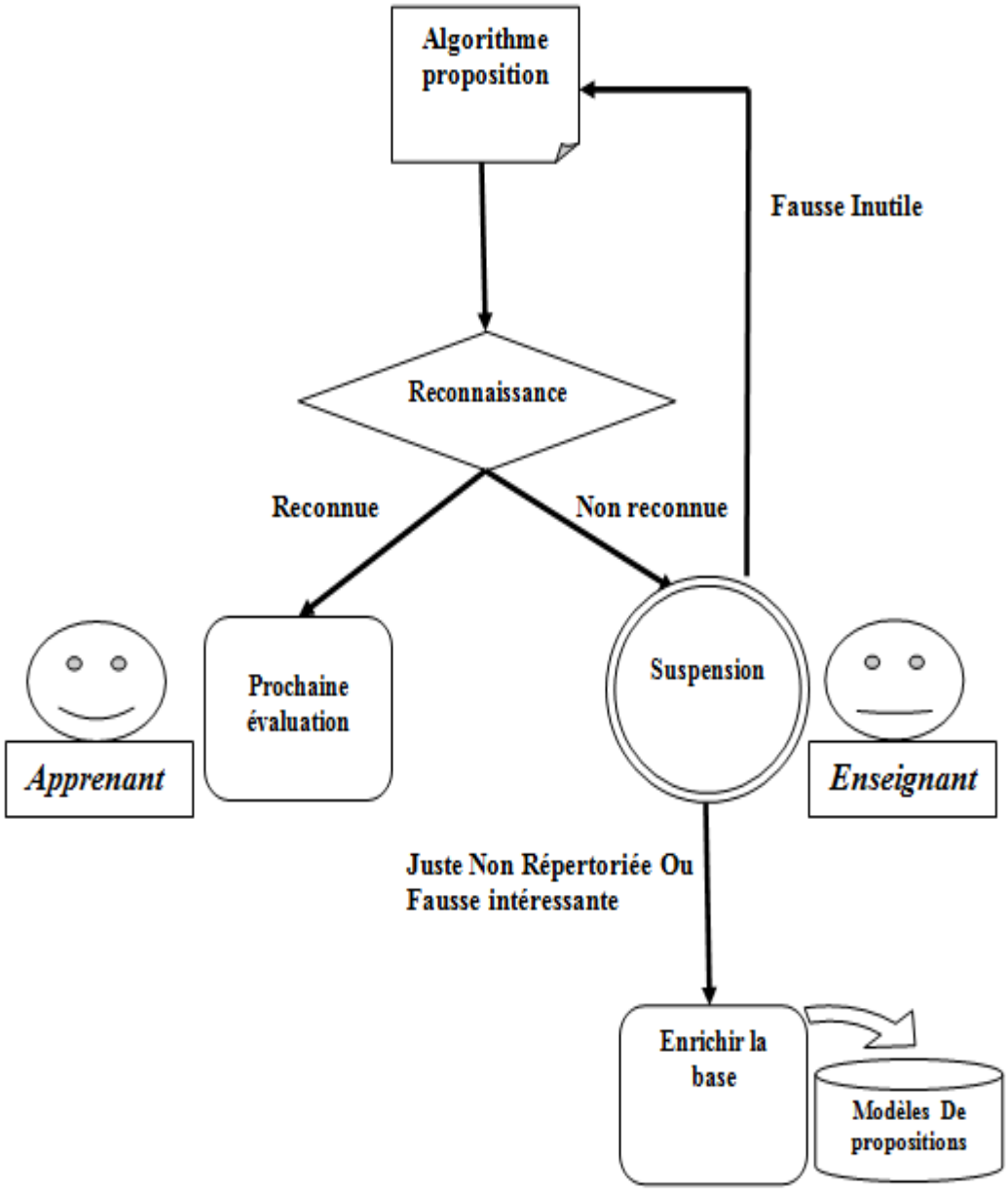

Figure 5.8 Processus d'évaluation

L'enseignant a l'habilité de juger la solution non reconnue par le système, et le résultat de ce jugement humain sera un des 3 trois cas suivants :

1. *Proposition Juste non répertoriée* : dans ce cas la proposition est juste mais son modèle n'est pas présent dans la base.
2. *Proposition Fausse utiles* : dans ce cas la proposition est fausse, mais utile.
3. *Proposition Fausse Inutile* : dans ce cas la proposition est fausse et inutile.

Dans les deux premiers cas, l'enseignant doit construire le modèle de la proposition et enrichir la base.

5.8.Conclusion

Dans ce chapitre nous avons présenté les grandes lignes d'une approche d'évaluation des algorithmes basée sur l'analyse du code d'un algorithme proposition. Cette analyse permet de comprendre comment les apprenants ont résolus un problème donné. Cela permet d'avoir une idée claire sur leurs intentions, il permet aussi de leur offrir de l'aide s'ils ont commis des erreurs. Une telle vision de l'évaluation semble plus pertinente que l'attribution d'une note, qui n'est pas parlante du point de vue pédagogique. Dans le chapitre suivant nous allons présenter cette approche avec plus de détails.

Chapitre 6

Compréhension des Algorithmes

« Quand on imagine, on ne fait que voir ; quand on conçoit, on compare »

'Jean-Jacques Rousseau'

6.1.Introduction

Dans le chapitre précédent, nous avons présenté les grandes lignes de notre approche d'évaluation des apprenants en algorithmique. Notre idée consistait en l'utilisation des techniques de compréhension de programmes pour comprendre des algorithmes proposés par les apprenants en vues de déduire ce qu'ils veulent faire pour résoudre un problème donné. Nous avons aussi présenté l'architecture d'un d'outil d'évaluation qui se composait principalement de 4 composantes : une base de problèmes, un espace d'édition des algorithmes, une base de modèles de propositions possibles, et un module de compréhension. Dans ce chapitre nous allons revenir sur cette architecture et voir les détails.

6.2.Base de problèmes

Notre population cible sont des apprenants débutants en algorithmique, souhaitant suivre des cours d'initiation à la programmation. Dans ces cours, les enseignants cherchent à apprendre leurs étudiants à écrire des algorithmes. Donc, les problèmes que nous allons utiliser doivent être en adéquation avec ce profil. Comme exemples de problèmes nous pouvons citer : manipulation d'un tableau (tri, parcours, …), maitrise des conditions et boucles, etc. Tout au long de ce manuscrit nous allons utiliser l'exercice présenté dans le chapitre précédent, qui consistait à vérifier si les éléments d'un tableau sont consécutifs ou pas, voir le chapitre précédent.

6.3. Edition des algorithmes

Pour comprendre les algorithmes, il est nécessaire de les éditer et de les traiter. En effet, nous avons besoin d'une représentation de ces algorithmes et de toutes les informations que nous voulons stocker tout en conservant la facilité d'utilisation. Pour cela nous avons développé un outil qui s'appelle AlgoEditor, voir la figure ci-dessous. Il permet d'éditer des algorithmes (proposition, et modèles de proposition) de façon visuelle, en d'autre terme l'écriture du code se fait automatiquement, ce qui libère l'élève et le module de compréhension des contrôles syntaxiques. En fait, cela permettra de se concentrer sur la compréhension automatique d'algorithmes.

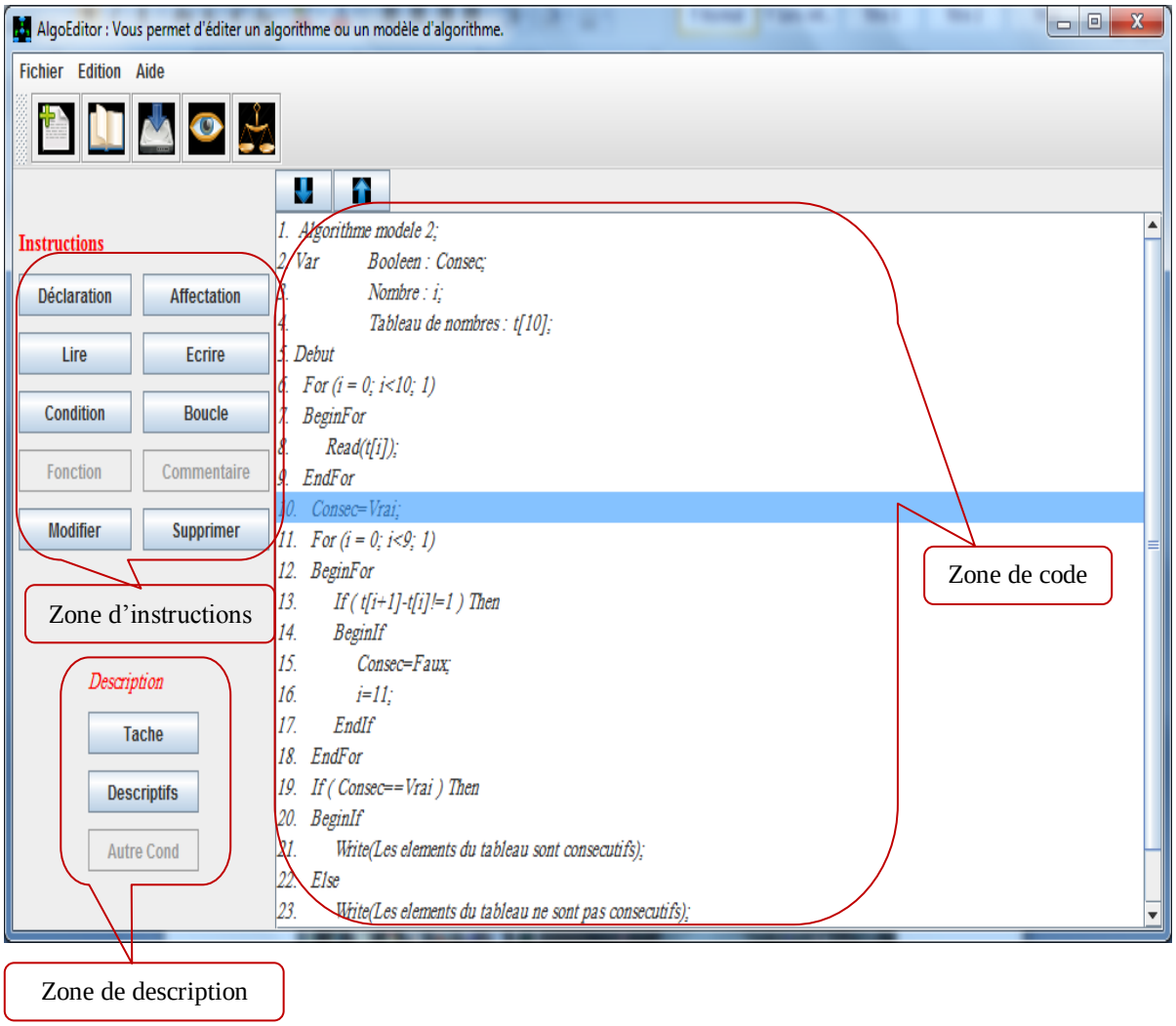

Figure 6.1 Interface de l'outil AlgoEditor

Comme présenté dans la figure 6.1, AlgoTest est composé principalement de trois zones:

1. *Zone de code :* Cette zone se trouve à droite, elle permet de voir le code de l'algorithme, en pseudo code.
2. *Zone d'Instructions* : Elle permet d'ajouter des instructions, de les mettre à jour ou de les supprimer. Elle est sur la gauche juste en dessous de la barre d'outils.
3. *Zone de description* : Cette zone est utilisée par les enseignants. Elle leur permet de décrire un algorithme en lui ajoutant des informations supplémentaires. Ces informations seront utilisées dans le processus de compréhension.

AlgoTest utilise plusieurs types d'instructions : déclaration de variable, lecture, écriture, les conditions, et les boucles.

6.3.1. Déclaration d'une variable

Dans AlgoEditor nous pouvons déclarer six types : Nombre, Caractère, Constant, tableau de nombres, un tableau de caractères et booléen. Chaque type nécessite un nom et un type voir la figure 6.2.

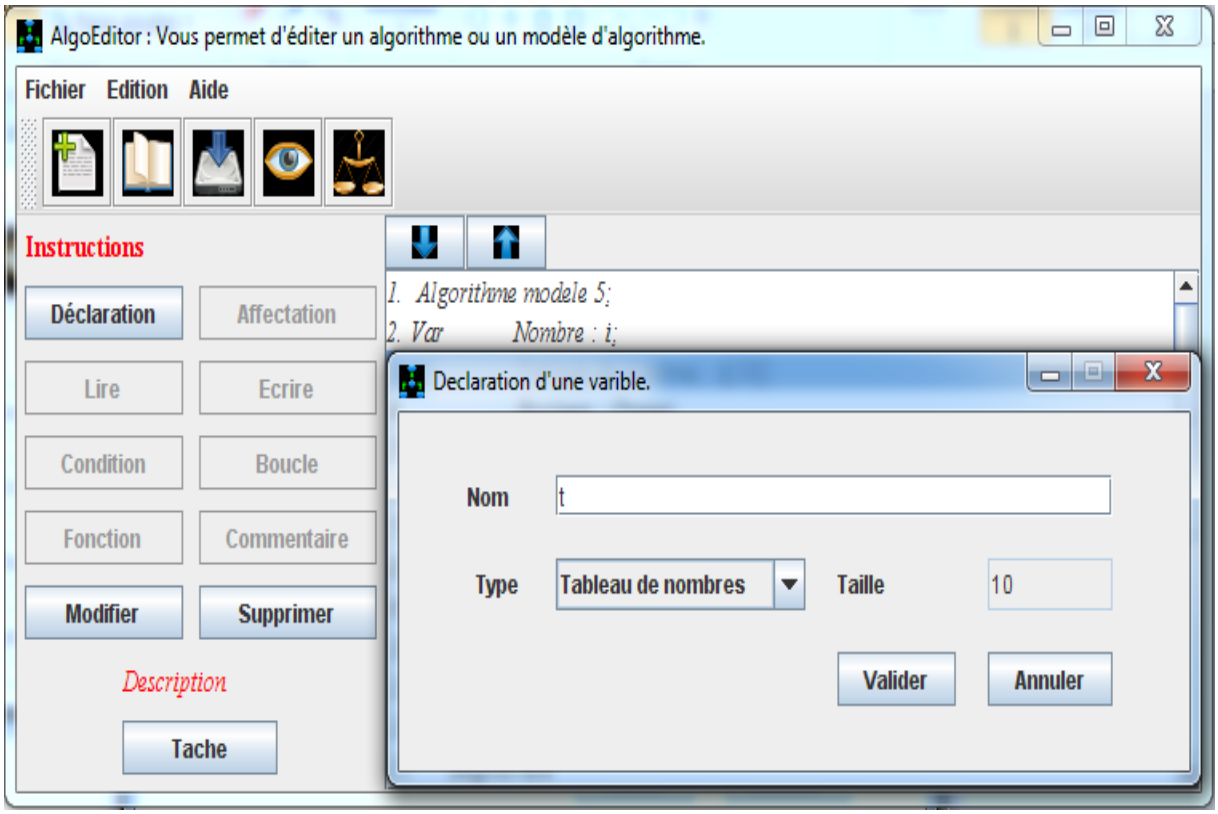

Figure 6.2 Déclaration d'une variable

Les variables de type tableau ont besoin d'information supplémentaire qui est la taille. Le champ contenant cette information sera grisé pour les autres types.

6.3.2. Lecture et écriture d'une variable

La fenêtre lecture permet de lires des variables qui peuvent être soit des variables simples (nombres) ou bien des tableaux. Dans ce dernier cas, il faut indiquer l'indice de l'élément à lire. Cet indice peut être un numéro ou bien être stocké dans une variable. Dans le premier cas, il faut remplir l'information Rang sinon il faut choisir la variable en question, voir la figure 6.3.

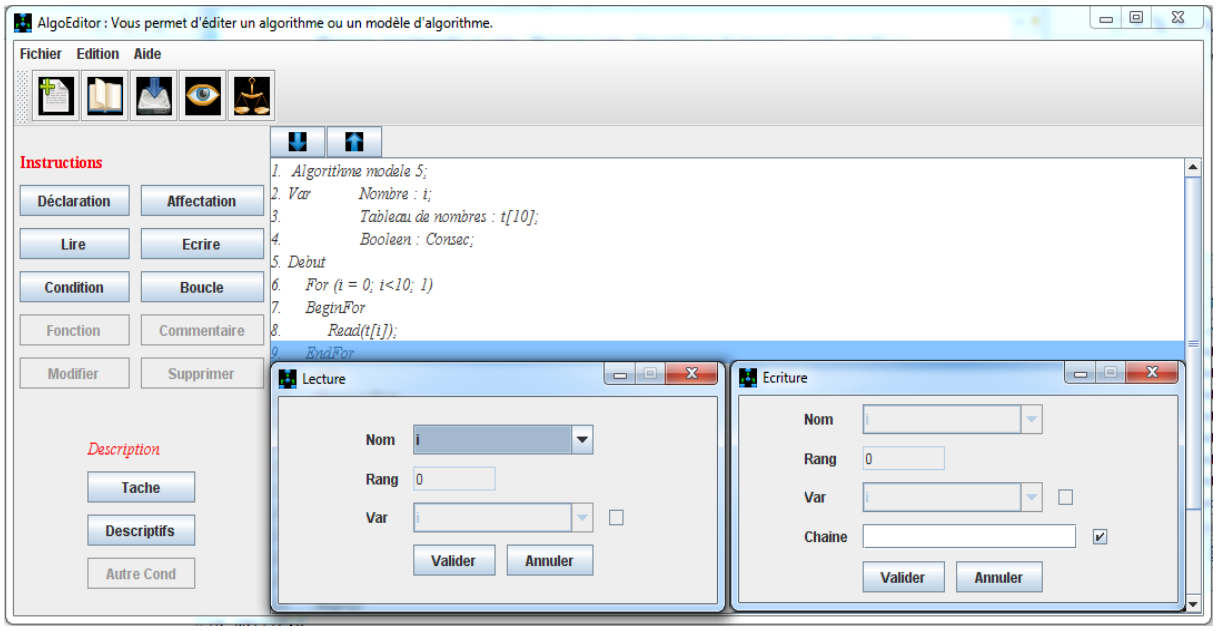

Figure 6.3 Lecture et écriture d'une variable

La fenêtre Ecriture permet d'afficher le contenu d'une variable. Si la variable est un tableau, il faut choisir l'indice. Il est possible aussi d'afficher des chaines de caractères. Dans ce cas, il faut cocher la case à cocher qui se trouve à droite du champ chaine.

6.3.3. Conditions

Pour faire des conditions, la fenêtre condition contient une case à cocher et un bouton (exp), voir la figure 6.4.

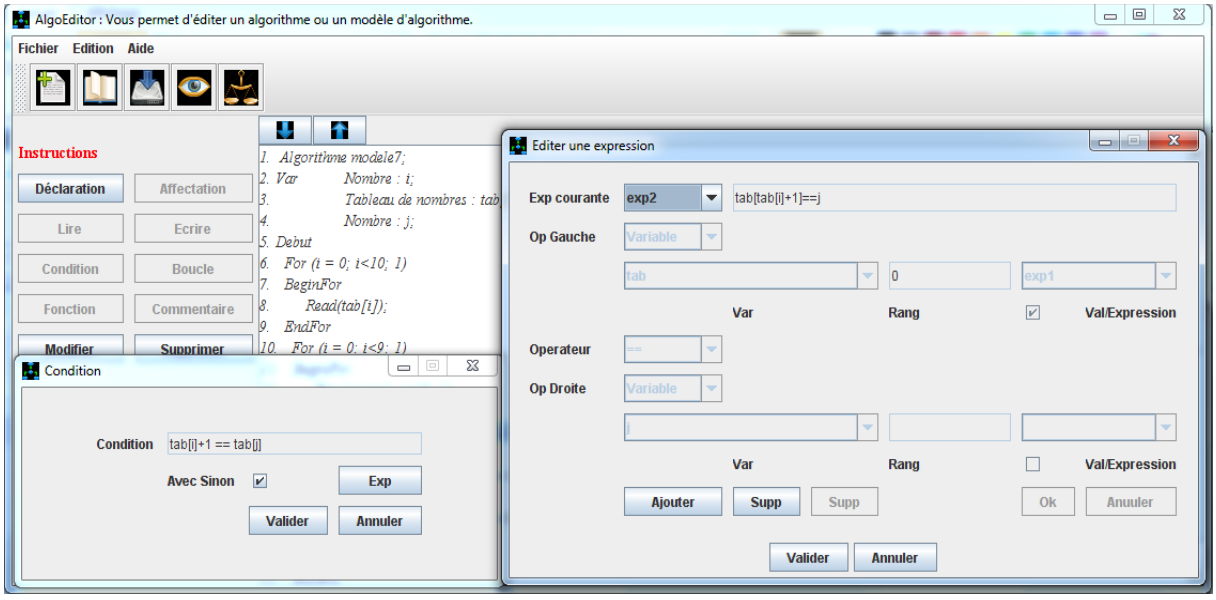

Figure 6.4 Conditions

La case à cocher permet de signaler la présence ou l'absence d'une instruction sinon. Quant au bouton exp, il permet de définir l'expression de la condition. Une expression est composée d'un opérateur (arithmétique ou logique), d'un opérande gauche et d'un opérande droit. Un opérande peut à son tour être une valeur ou récursivement d'une expression. Dans le cas d'une valeur, il faut spécifier s'il s'agit d'une valeur stockée dans une variable ou bien saisie par l'utilisateur. Dans le premier cas il faut choisir la variable en question. Donc pour avoir une condition, il faut définir itérativement des expressions jusqu'arriver à l'expression voulue.

6.3.4. Répétition

Pour faire une répétition, la fenêtre boucle peut être utilisée. Pour cela, il faut choisir le type de la boucle. Nous avons utilisé deux types de boucle : `Pour, et tant que`. La boucle pour nécessite le choix du compteur, la condition et le pas, voir la figure 6.5.

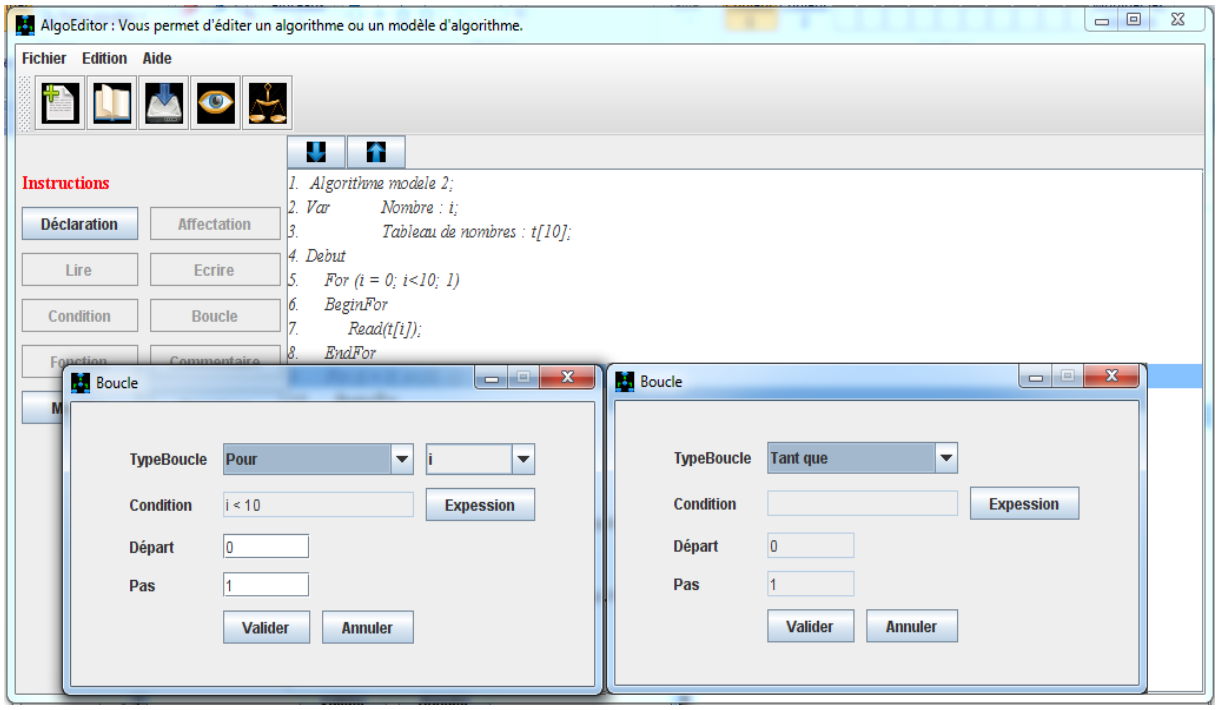

Figure 6.5 Boucles

La boucle Tant que nécessite une condition d'arrêt, les autres champs seront grisés, voir la figure 6.5.

En plus des instructions, AlgoEditor permet la description des algorithmes. Cela est nécessaire pour construire la base des modèles de propositions.

6.4.Modèle : Description d'un algorithme

Un modèle représente un algorithme dont la stratégie est fréquemment adoptée par les apprenants, voir la figure 6.6.

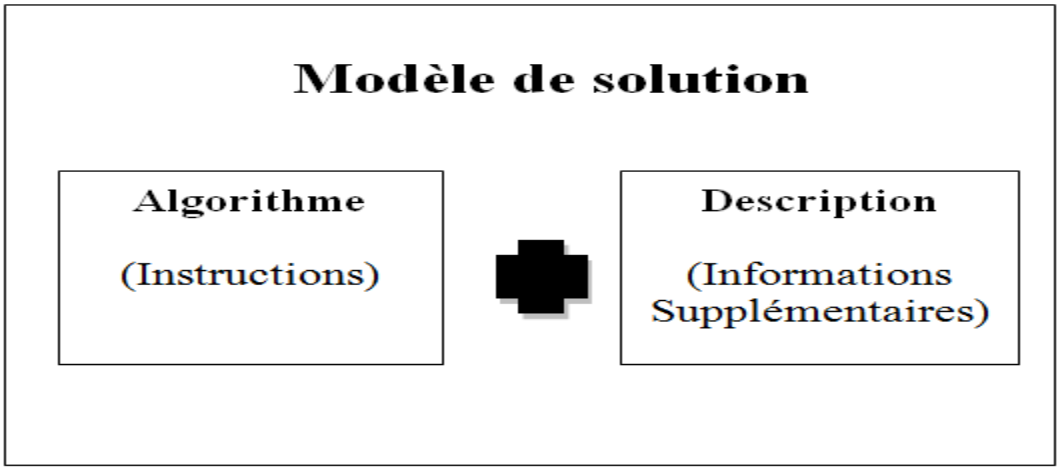

Figure 6.6 Modèle de solution

En plus des instructions, Un modèle peut contenir des informations supplémentaires qui décrivent les intentions des apprenants, les défaillances de la solution, etc. Donc, un modèle est une abstraction de l'algorithme, il permet d'avoir plusieurs instances de la même proposition, par exemple : les trois instructions suivante permettent de lire deux variables et affiche leur somme.

Modèle	Version A_B	Version X_Y
1. Read(V1) 2. Read(V2) 3. Write(V1+V2)	1. Read(A) 2. Read(B) 3. Write(A+B)	1. Read(Y) 2. Read(X) 3. Write(X+Y)

Tableau 6.1 Versions et Modèle

Les informations qui nous intéressent dans ce travail sont : une note, les tâches (voir chapitre précédent), les descriptifs. Elles seront utiles pendant la compréhension des propositions des apprenants.

1. *Note :* Note globale du modèle.
2. *Tâches :* Pour chaque modèle, on doit définir toutes les tâches qui le composent. Chaque tâche peut avoir un nom, une note (locale de tâche), des indicateurs de

présence de la tâche (Lignes critiques, voir plus loin), et les instructions qu'elle contient.

3. *Descriptifs :* ils permettent d'avoir plus de détails sur le modèle, tels que : autoriser l'absence de certaines instructions voire de certaines tâches, autoriser le désordre entre certaines instructions voire entre certaines tâches, pénaliser l'absence, pénaliser le désordre, plusieurs alternatives d'une même condition.

6.4.1. Tâches et lignes critiques

La figure ci-dessous montre un modèle composé de quatre tâches : Lecture (7→10), Initialisation (11→12), Test (14→20), et Résultat (21→26). Si on suppose que la note de cet exercice est sur 6 points, nous pourrions avoir la description suivante :

```
1.  Algorithme modele 7;
2.  Var         Nombre : i;
3.              Nombre : j;
4.              Tableau de nombres : t[10];
5.              Booleen : Consec;
6.  Debut
7.      For (i = 0; i<10; 1)
8.      BeginFor
9.          Read(t[i]);
10.     EndFor
11.     j=0;
12.     Consec=Vrai;
13.     While ( j<9ANDConsec== Vrai )
14.     BeginWhile
15.         If ( t[j]+1!=t[j+1] ) Then
16.         BeginIf
17.             Consec=Faux;
18.         EndIf
19.         j=j+1;
20.     EndWhile
21.     If ( Consec==Vrai ) Then
22.     BeginIf
23.         Write(Les elements du tableau sont consecutifs);
24.     Else
25.         Write(Les elements du tableau ne sont pas consecutifs);
26.     EndIf
27. Fin
```

Tâche : Lecture (note : 1, déb : 7, fin 10, lc : 9)

Tâche : Initialisation (note : 1, déb : 11, fin 12, lc : 12)

Tâche : Test (note : 2, déb : 13, fin 20, lc : {13, 15, 17})

Tâche : Résultat (note : 1, déb : 21, fin 26, lc : {21,23})

Figure 6.7 Description des tâches d'un algorithme

La tâche lecture dont la note est 1 sur 6, et dont les instructions se trouvent entre 7 (début) et 10 (fin), contient un indicateur de présence ou ligne critique qui est la ligne 9 (lc). Une ligne

critique est une instruction dont l'apparition dans le code de l'algorithme proposition indique la possibilité de présence de certaines tâches d'un modèle donné, par exemple : si dans un algorithme on trouve l'instruction Read(t[i]) qui est la ligne critique de la tâche lecture, il est fort possible qu'il y aura une boucle de lecture des éléments d'un tableau.

```
7. For(int i=0; i<10;i++)
8. BeginFor
9.       Read(t[i]);    // * Est une ligne critique de la tâche lecture
10. EndFor
```

Il faut noter qu'une tâche peut avoir plusieurs lignes critiques, c'est le cas pour la tâche test. Cette tâche contient trois lignes critiques qui sont : 13,15, et 17.

Donc pour décrire un algorithme il faut définir les tâches qu'il contient. AlgoEditor permet de le faire à l'aide la fenêtre tâche, voir la figure 6.8.

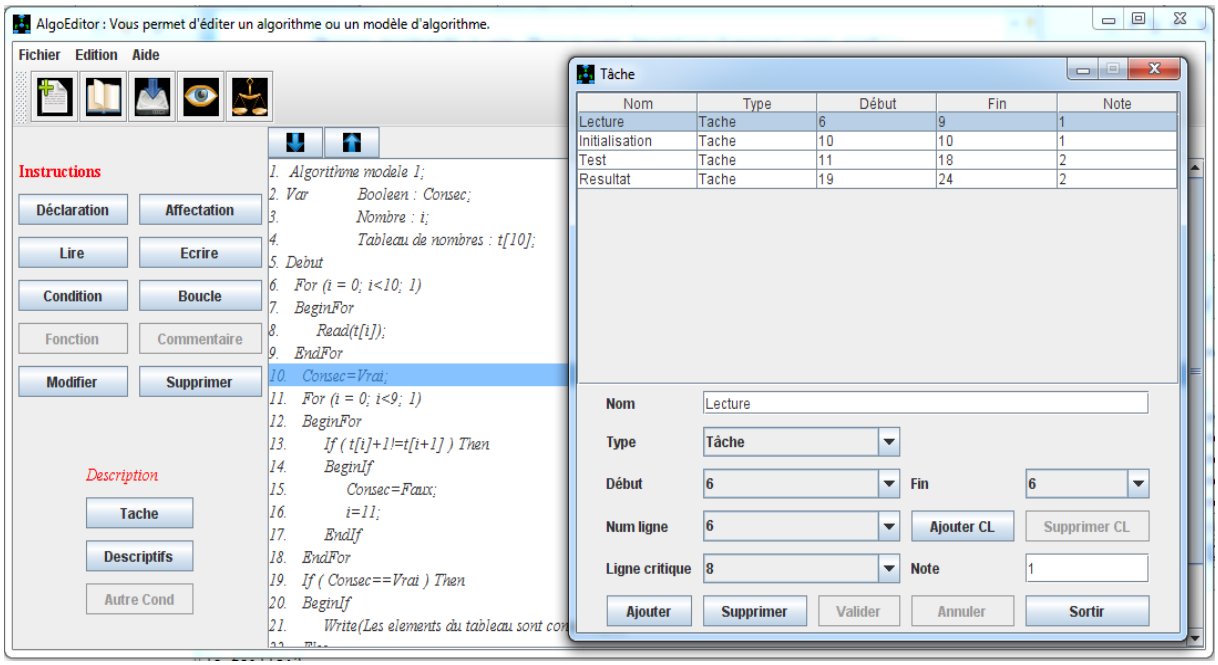

Figure 6.8 Définition des tâches d'un modèle

La définition des tâches nous permet de nous situer à un niveau d'abstraction plus élevé que celui du niveau code. Elle nous permet d'avoir une idée sur les étapes abordées pour résoudre le problème, c'est le niveau conception. Cela reflète une stratégie qui peut être bonne ou mauvaise. Dans les deux cas, nous aurons une idée sur ce que les apprenants veulent faire si leurs propositions collent à cette description.

Après la définition des tâches du modèle, il faut définir certains descriptifs.

6.4.2. Descriptifs

Ils nous permettent d'aller plus loin dans la description, et avoir des alternatives d'un modèle. Les types de descripteurs que nous avons utilisés sont :

- **Descriptif d'absence d'une ligne :** indique la possibilité d'absence d'une ligne, avec la possibilité de pénaliser.
- **Descriptifs d'absence d'une tâche :** indique la possibilité d'absence d'une tâche, avec la possibilité de pénaliser.
- **Descriptifs de désordre entre lignes :** indique la possibilité de désordre entre lignes, avec la possibilité de pénaliser.
- **Descriptifs de désordre entre tâches :** indique la possibilité de désordre entre tâches, avec la possibilité de pénaliser.

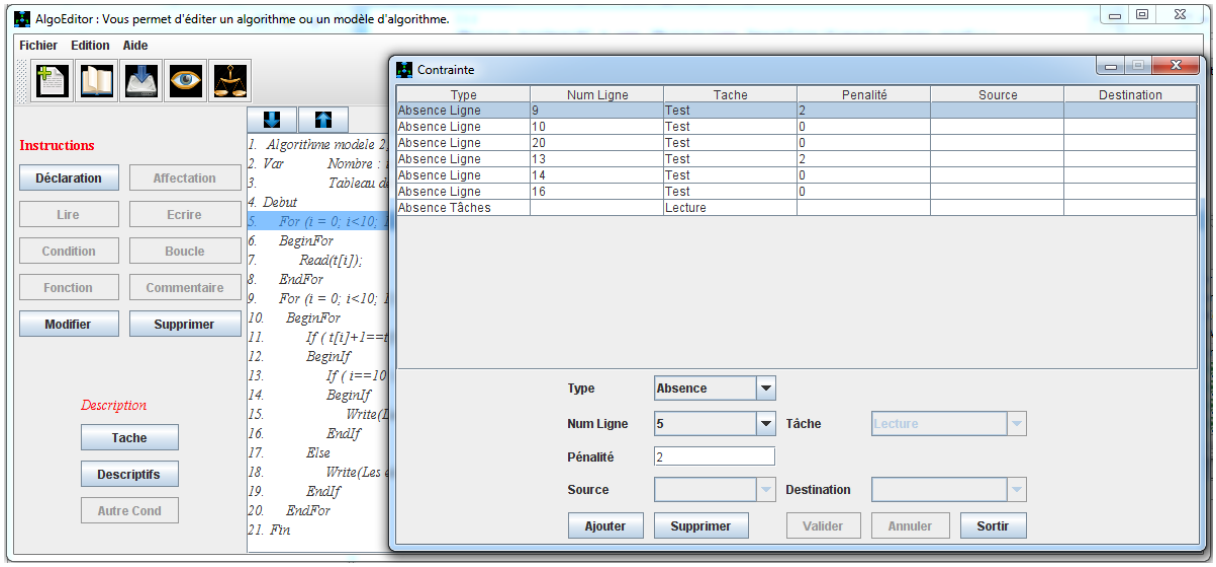

Figure 6.9 Descriptifs

La description des tâches et des descriptifs est le cœur du processus de compréhension. Les descriptifs permettent d'avoir plusieurs versions du même modèle et d'être tolérant lors de la compréhension. Cela donne plus de flexibilité à l'évaluation et permet d'apprécier les bonnes parties du code. Ainsi, dans sa proposition, un apprenant peut réussir pour certaines tâches et échouer pour d'autres.

Dans algoEditor, il existe un autre type de descriptif qui permet d'avoir plusieurs versions d'une même condition, voir .la figure 6.10.

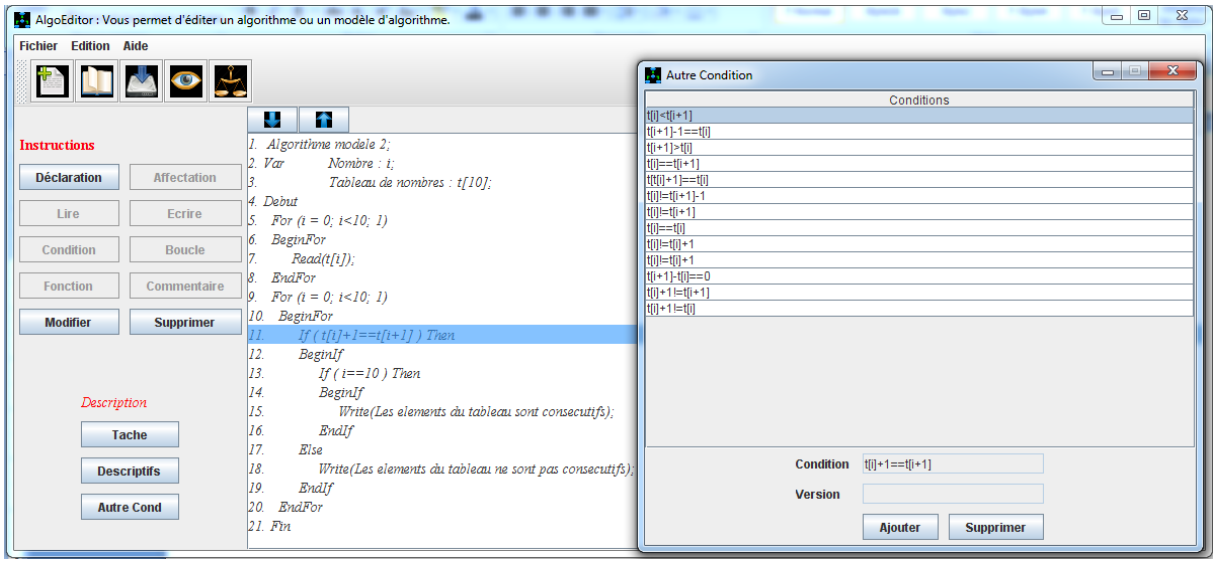

Figure 6.10 Descriptif versions d'une condition

En utilisant ce descriptif, une condition peut avoir plusieurs versions équivalentes. Cela nous permet d'éviter d'utiliser un modèle pour chaque version de la condition, et regrouper plusieurs modèles de propositions dans un seul algorithme. Les algorithmes (propositions ou modèles) sont présentés en pseudo code pour l'utilisateur, mais ils sont représentés en interne en utilisant le langage XML, comme présenté à la figure 6.11.

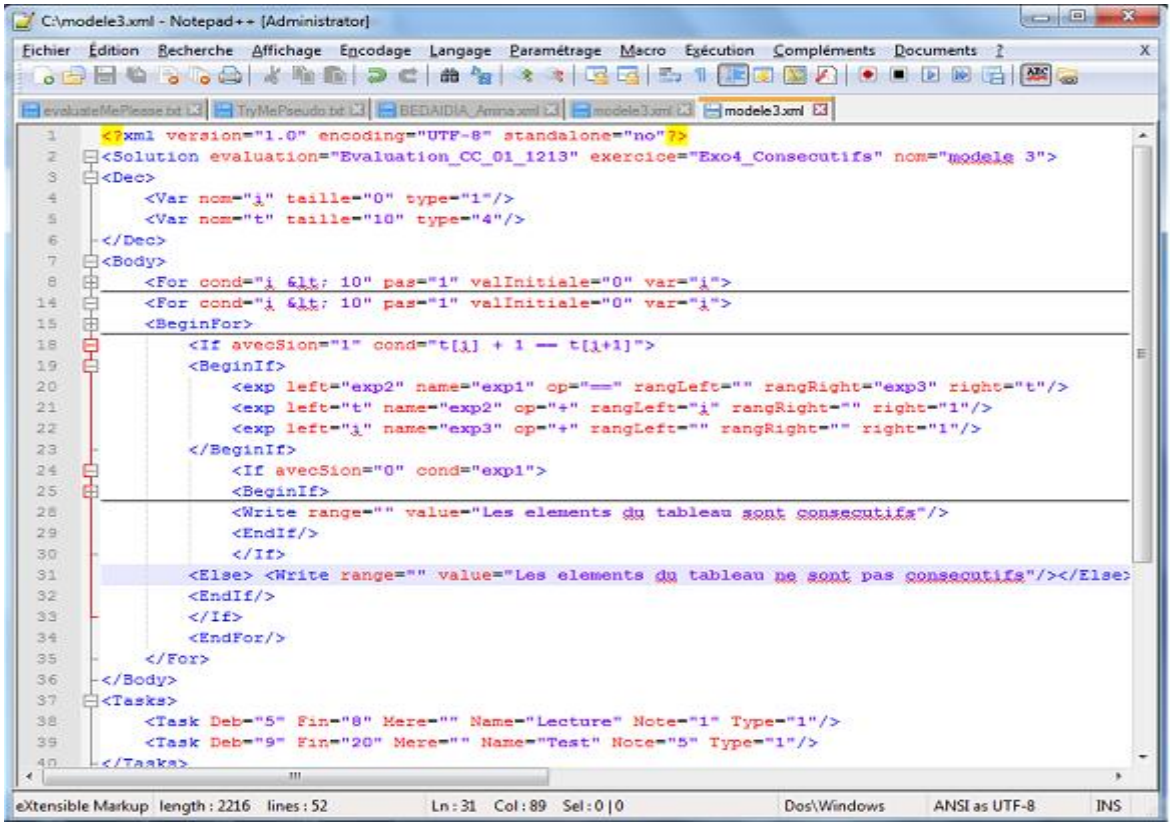

Figure 6.11 Représentation XML d'un algorithme

6.5. Compréhension des propositions

Comprendre un algorithme consiste à essayer d'identifier le modèle qui lui correspond. Nous ne cherchons pas à faire une simple comparaison des algorithmes, mais nous essayons d'extraire des informations sur ce que l'apprenant veut faire. Nous devons prendre en considération le fait que la même solution pourrait avoir des versions différentes en utilisant différents dénomination de variables. Dans cette section, nous expliquerons le processus de compréhension des propositions des apprenants. La figure 6.11 présente les étapes par lesquelles passe ce processus. Dans un premier temps T1, on fait une analyse du code, elle sert à localiser les lignes critiques de tous les modèles de propositions présentes dans l'algorithme proposition de l'apprenant. Chaque ligne critique indique la possibilité de présence d'une ou plusieurs tâches. Dans un deuxième temps T2, on localise les tâches présentes dans la proposition. Ainsi, nous avions un premier modèle candidat, celui qui contient le plus de tâches et de lignes critiques. Enfin, on passe à la reconnaissance de la proposition en vérifiant si elle correspond au modèle candidat ou pas.

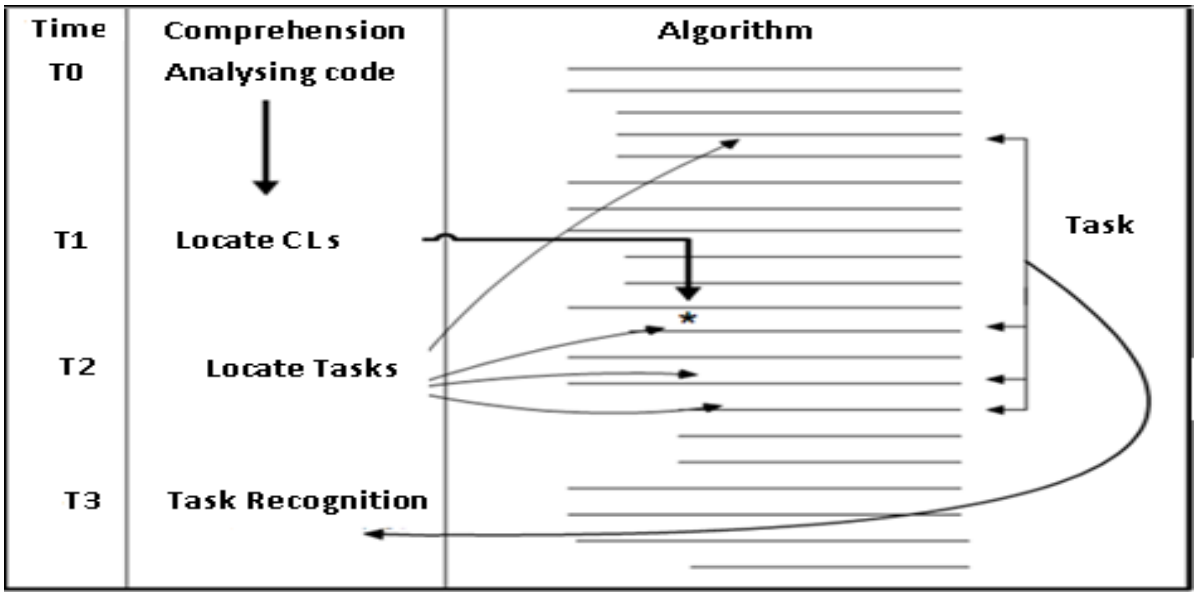

Figure 6.12 Etapes de la compréhension

La reconnaissance d'un algorithme proposition n'est pas assurée du premier coup, des Aller-retour entre les étapes de reconnaissance sont nécessaires, cela est justifié par le fait qu'une ligne critique peut indiquer la présence de plusieurs tâches, et une tâche peut appartenir à plusieurs solutions.

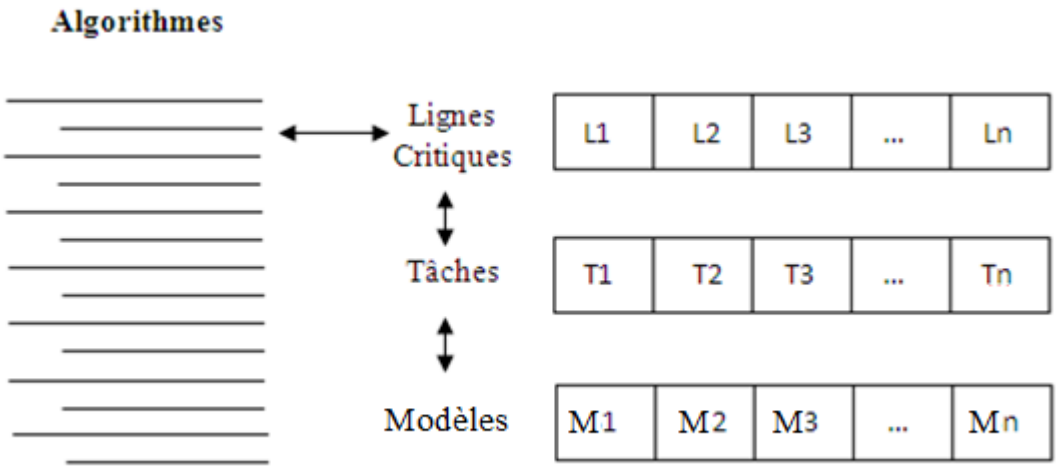

Figure 6.13 Aller-retour entre les étapes de compréhension

Il n'est pas possible de prévoir tous les modèles de propositions d'un problème donné, pour cela dans le cas où une proposition n'est pas reconnue nous proposons de suspendre l'évaluation, et demandons l'intervention de l'expert humain. De cette manière nous assurons une reconnaissance binaire des algorithmes et une évolutivité de la base de modèles de propositions (voir le chapitre 5).

Pour mettre en œuvre ce processus de compréhension, nous avons proposé l'algorithme suivant :

1. Charger tous les modèles
2. Charger la solution
3. Initialiser l'état de tous les modèles avec la valeur « n'est pas vu »
4. Charger les variables utilisées dans la solution
5. Initialiser l'état de la solution avec la valeur « n'est pas reconnue »
6. **For** chaque modèle
7. **BeginFor1**
8. | Charger les variables clones du modèle // équivalentes dans la proposition
9. | Chercher les lignes critiques du modèle
10. **EndFor**
11. modCand = récupérerMeilleureCand(modèles) // modCand : Modèle candidat
12. **While** on a un candidat & la solution n'a pas été reconnue
13. **BeginWhile**
14. | EssayerModèle(cand)

15. | Mettre à jour l'état du modèle « à vu »
16. | Essayer un autre modèle : modCand = récupérerCandidatSuivant(modèles)
17. **EndWhile**
18. Ajouter la solution à la liste des solutions évaluées

Donc au départ nous chargeons tous les modèles en initialisant l'état de chacun à n'est pas vu. Puis nous chargeons la proposition en initialisant son état à n'est pas reconnue. Ensuite, nous cherchons pour chaque modèle les lignes critiques présentes dans la proposition de l'apprenant, après avoir chargé les variables du modèle. Le chargement des variables nous permet de faire une correspondance entre chacune des variables du modèles et toutes les variables équivalentes dans la proposition, d'où vient l'appellation clone (copie), voir la ligne 8 dans le code de l'algorithme. Ensuite, nous cherchons le meilleure candidat parmi les modèles de propositions du problème qu'on veut résoudre, et ce en appelant la fonction récupérerMeilleureCand, voir la figure 6.14.

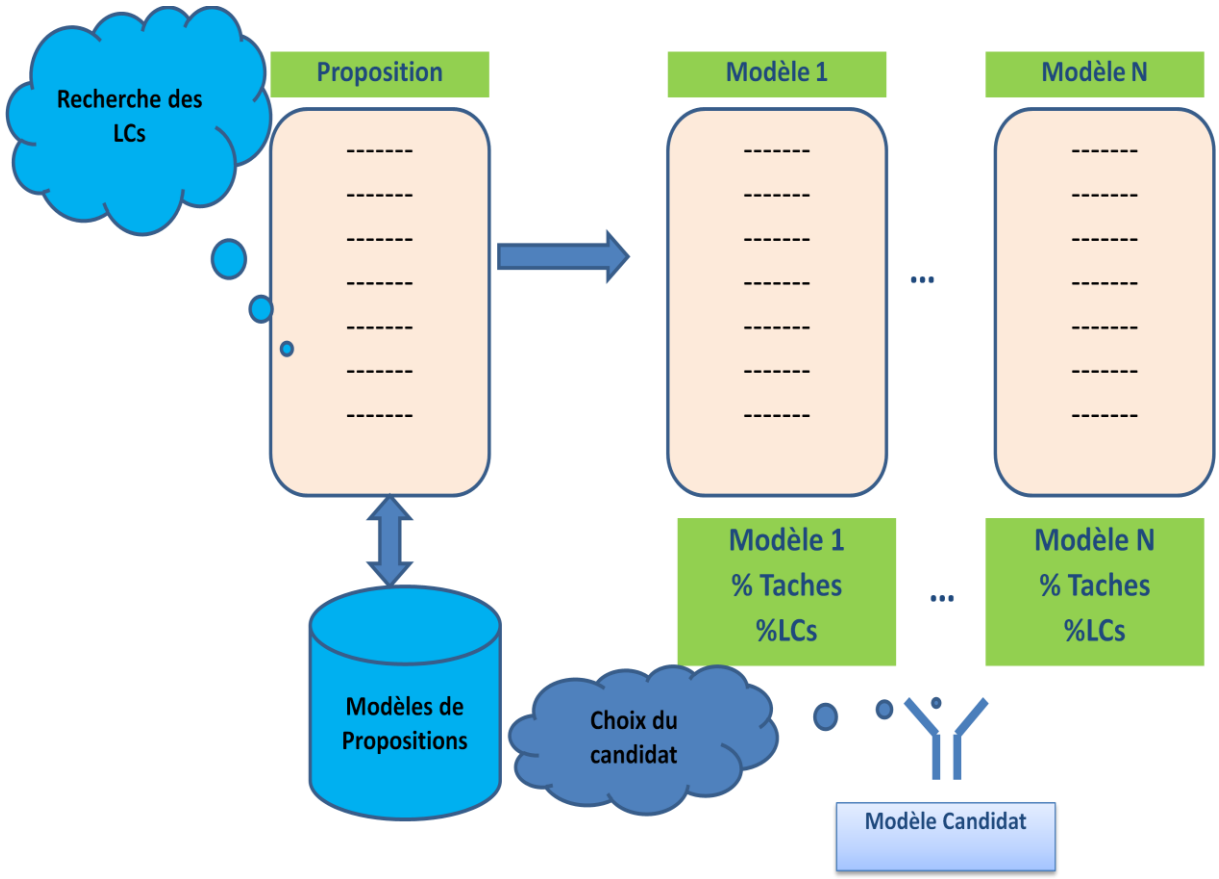

Figure 6.14 Processus de compréhension (1)

Pour choisir le meilleur candidat parmi les modèles de propositions, la fonction récupérerMeilleureCand(), calcule le pourcentage des tâches localisées, et le pourcentage des

lignes critiques localisées dans la proposition de l'apprenant en appliquant les formules suivantes :

*PrcTLs = (NbTâchesLocalisées * 100) / NbTâchesModèle ( ) ;*

*PrcLCLs = (NbreLCLs ( ) * 100) / NbtLCsModèle ( ) ;*

1. ***PrcTLs*** est le pourcentage de tâches localisées dans la proposition.

2. ***NbTâchesLocalisées*** est le nombre de tâches localisées.

3. ***NbTâchesModèle*** est le nombre de tâches total.

4. ***PrcLCLs*** est le pourcentage des lignes critiques localisées.

5. ***NbreLCLs*** est le nombre des lignes critiques localisées.

6. ***NbLCsModèle*** est le nombre des lignes critiques du modèle.

Nous appliquant un filtre sur les modèles par rapport au pourcentage des tâches localisées. Si nous avons le même pourcentage, nous appliquerons un deuxième filtre basé sur le pourcentage des lignes critiques localisées. Après avoir eu un modèle candidat, l'étape suivante consiste à vérifier la correspondance entre ce candidat et la proposition de l'apprenant, en appliquant l'algorithme suivant :

1. Initialiser l'état de toutes les instructions de la proposition avec la valeur « n'est pas vue »
2. Initialiser la liste des tâches désirées depuis le modèle
3. Initialiser la liste des tâches reconnues avec la valeur « vide »
4. **For** chaque Tâche du modèle
5. **BeginFor1**
6. récupérer Début, & Fin
7. Charger l'instruction de début de la tâche depuis le modèle
8. Charger les instructions similaires depuis la proposition
9. **For** chaque début possible
10. **beginFor2**
11. TâcheDésiree = Vrai
12. **If** (pasTraité (débutPossible) **&** son instruction n'est pas reconnue)

```
13.        BeginIf
14.              For chaque instruction de la tâche courante (commençant par début)
15.              BeginFor3
16.                    Comparer les instructions
17.              EndFor3
18.        Else1
19.        TâcheDésirée = faux
20.     EndIf
21.  EndFor2
22. EndFor1
23. DeciderSiPropositionEstReconnue
```

L'idée consiste à : pour chaque tâche du modèle candidat, nous récupérons les indices de lignes de début et de fin (qui sont des numéros). Puis, il faut charger l'instruction dont le numéro est égal à début. Ensuite, nous recherchons dans la proposition les instructions similaires à début. Ces instructions représentent des débuts possibles de la tâche désirée dans la proposition. Pour confirmer que l'un de ces débuts possibles est celui que nous souhaitons avoir, nous comparons toutes les instructions de la tâche souhaitée avec celles de la proposition et ce en commençant par les débuts possibles, voir la figure 6.15.

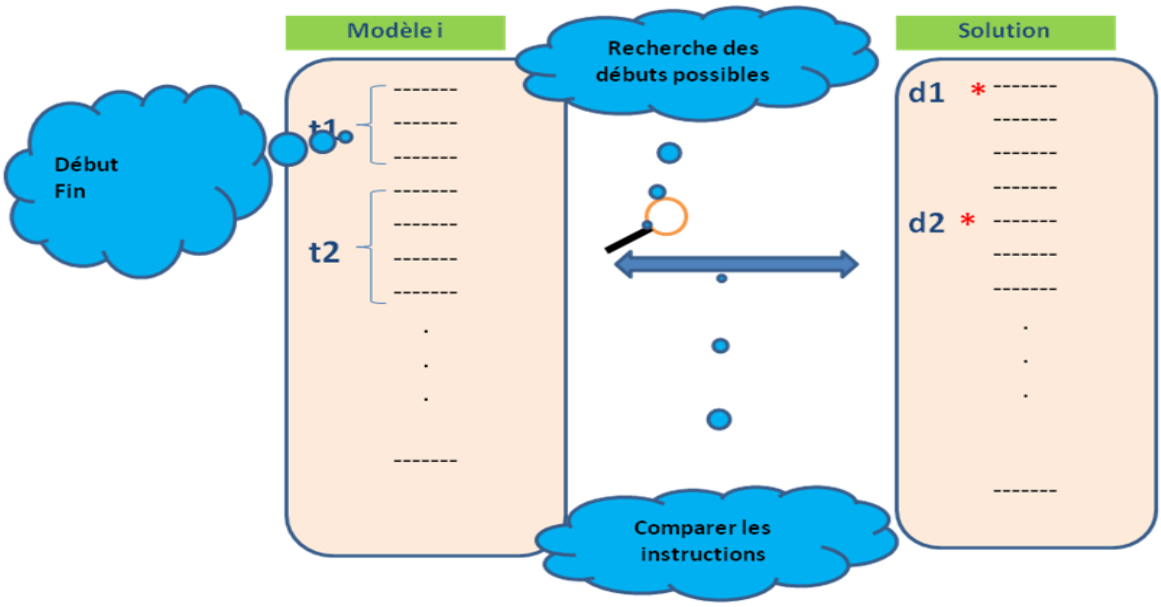

Figure 6.15 Processus de compréhension (2)

Après avoir traité toutes les tâches, nous devons vérifier la reconnaissance de la solution en comparant la liste des tâches reconnues avec la liste des tâches souhaitées. Nous nous basons

sur la description figurant dans le fichier XML de l'algorithme, qui contient les tâches ainsi que tous les descriptifs. Pour quelle soit reconnue, une proposition doit contenir toutes les tâches présentes dans la description XML. Seules les tâches ayant un descripteur (dans le modèle) qui leur permet d'être absentes peuvent s'absenter. On fait pareil pour les instructions d'une tâche. De cette manière si toutes les tâches et instructions sensées être présentes dans la proposition le sont, l'apprenant aura la note complète. Cette note, s'abaisse, dans le cas où il y a des tâches ou instructions absentes ou en désordre, avec bien sûr la présence des descripteurs décrivant cette absence ou ce désordre dans la description du modèle.

6.6.Evaluation basée compréhension

Pour évaluer des apprenants en algorithmique, notre approche consistait à extraire le maximum d'informations à partir de l'algorithme proposition, voir le chapitre précédent. Ces informations peuvent avoir plusieurs natures et doivent refléter le niveau de l'apprenant. Afin d'atteindre notre objectif, nous avons divisé le travail en deux parties, voir la figure 6.16.

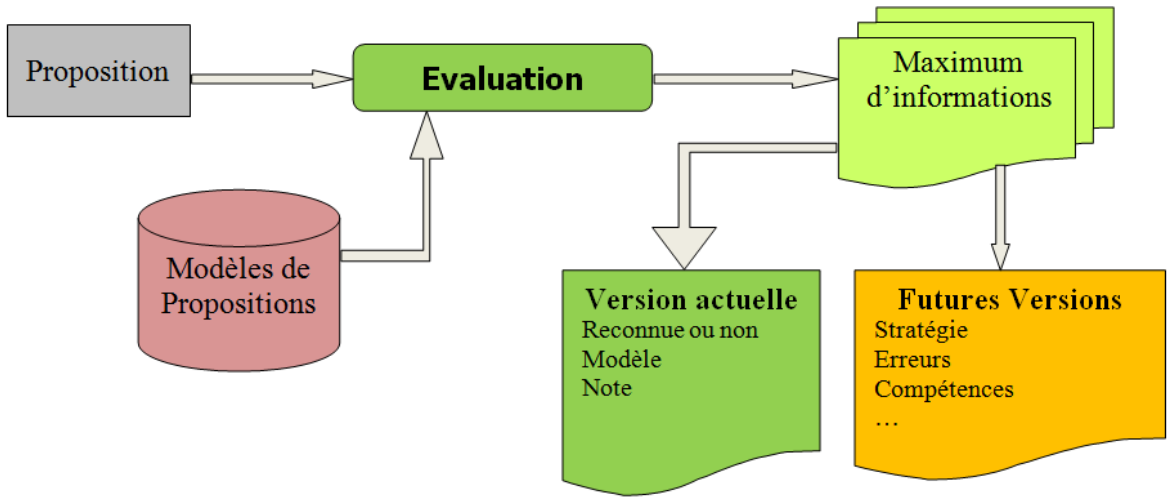

Figure 6.16 Evaluation basée compréhension

Selon la figure 6.16, l'évaluation d'un algorithme proposition signifiait chercher un modèle équivalent depuis une base de modèles en appliquant des techniques de compréhension. Dans une première version l'évaluation indiquera si la proposition a été reconnue ou pas, si oui quel est le modèle correspondant, et quelle est la note attribuée à cette proposition.

Après avoir eu un modèle d'une proposition, il serait possible d'enrichir cette évaluation en extrayant d'autres informations telles que : la stratégie, les erreurs, les compétences, etc.

6.7.Conclusion

Dans ce chapitre nous avons présenté en détails notre approche d'évaluation des algorithmes. Un outil d'édition des algorithmes a été présenté aussi et nous avons expliqué comment construire un modèle avec. L'interface de l'outil AlgoTest est basée sur la génération automatique du code, ce qui permet aux apprenants de ne pas commettre des erreurs syntaxique. L'enseignant a plus de travail, il doit construire les modèles des propositions.

Un algorithme de compréhension de propositions a été présenté et expliqué. Cet algorithme a été implémenté en java et xml. Dans le chapitre suivant nous essayerons de présenter notre étude expérimentale, et nous allons discuter les résultats obtenus.

Chapitre 7

Expérimentation et premiers résultats

« Une science doit être basée sur l'expérience, un jugement, sur la vérité »
'Pierre-Claude-Victor Boiste'

7.1.Introduction

Ce chapitre est consacré à la présentation de notre étude expérimentale, et à la discussion des résultats que nous avons obtenus. Nous allons essayer de présenter les résultats d'évaluation que nous avons eus en utilisant l'outil AlgorTest. Une comparaison entre la reconnaissance automatique et manuelle sera présentée, plus un retour sur les notes.

Avant de présenter les résultats, nous allons voir le contexte de notre expérimentation, et l'évaluation manuelle que nous avons faite.

7.2.Contexte de l'expérimentation

Pour tester notre approche, nous avons travaillé sur les algorithmes des étudiants de l'école préparatoire aux sciences et techniques Annaba, en Algérie. Cette école a pour but de prendre en charge, de former et d'encadrer les meilleurs bacheliers. En deux ans, une formation scientifique de haut niveau et de qualité leur est assurée pour préparer le concours d'entrée aux Grandes Ecoles de Technologie. Les étudiants avec lesquels nous avons travaillés sont du niveau deuxième année, ils avaient suivi des cours de programmation en première année. L'expérimentation est faite principalement sur l'exercice présenté dans le chapitre 5 dont l'énoncé est le suivant : écrire un algorithme qui permet de vérifier si les éléments d'un tableau (d'entiers) sont consécutifs ou non.

Exemple : les éléments 4, 5, 6, 7, 8 sont consécutifs tandis que les éléments : 1, 3, 4, 5, 6 ne le sont pas.

7.3.Construction des modèles

Après la correction manuelle d'un groupe composé de 22 personnes, nous avons pu extraire 7 modèles de proposition possibles (correctes et incorrectes) dont les codes de quelques uns sont présentés ci-dessous.

```
1.  Algorithme modele 1;
2.  Var          Booleen : Consec;
3.               Nombre : i;
4.               Tableau de nombres : t[10];
5.  Debut
6.      For (i = 0; i<10; 1)
7.      BeginFor
8.          Read(t[i]);
9.      EndFor
10.     Consec=Vrai;
11.     For (i = 0; i<9; 1)
12.     BeginFor
13.         If ( t[i]+1!=t[i+1] ) Then
14.         BeginIf
15.             Consec=Faux;
16.             i=11;
17.         EndIf
18.     EndFor
19.     If ( Consec==Vrai ) Then
20.     BeginIf
21.         Write(Les elements du tableau sont consecutifs);
22.     Else
23.         Write(Les elements du tableau ne sont pas consecutifs);
24.     EndIf
25. Fin
```

Figure 7.1 Modèle1

```
1.  Algorithme modele 2;
2.  Var          Nombre : i;
3.               Tableau de nombres : t[10];
4.  Debut
5.      For (i = 0; i<10; 1)
6.      BeginFor
7.          Read(t[i]);
8.      EndFor
9.      For (i = 0; i<10; 1)
10.     BeginFor
11.         If ( t[i]+1==t[i+1] ) Then
12.         BeginIf
13.             If ( i==10 ) Then
14.             BeginIf
15.                 Write(Les elements du tableau sont consecutifs);
16.             EndIf
17.         Else
18.             Write(Les elements du tableau ne sont pas consecutifs);
19.         EndIf
20.     EndFor
21. Fin
```

Figure 7.2 Modèle2

La première phase de notre recherche était de faire une évaluation manuelle, avec l'objectif d'établir manuellement la correspondance entre ces modèles et les propositions des étudiants.

7.4.Evaluation manuelle

Nous avons pris les copies de trois groupes (du même enseignant) d'environ 21 personnes, et après avoir saisi les algorithmes qu'elles contiennent, nous y avons appliqué les modèles présentés dans le tableau 7.1, voir ci-dessous.

En analysant ces résultats, nous pouvons constater que plus de la moitié des algorithmes proposés étaient reconnues. Fréquemment, les apprenants font les mêmes erreurs (syntaxe ou réflexion), et les propositions incorrectes sont plus nombreuses que celles qui sont correctes. Cela nous a encouragé, à poser les questions suivantes : est t-il possible de faire une évaluation automatique? Est-ce que nous pouvons dire que les étudiants de l'année prochaine adopteront, les mêmes propositions que nous avons récupérées de l'évaluation de cette année?

		Grp1	**Grp2**	**Grp3**	**Total**
		22	**19**	**20**	
Reconnue	Correct	3	2	2	7
	Incorrect	11	9	10	30
Non Reconnue	Correct	1	1	2	4
	Incorrect	7	7	6	20
Taux de reconnaissance		**63**	**57**	**60**	**61**

Tableau 7.1 Résultats de l'évaluation manuelle

Notre réponse est simple : nous allons refaire cette expérimentation (automatiquement) par un outil (AlgoTest). Si cet outil nous donne des résultats similaires à ceux obtenus manuellement, notre objectif ne se limitera pas en la compréhension des algorithmes, il sera

de fournir aux enseignants (et pourquoi pas aux étudiants) un outil qui réutilise l'expertise des années passées pour évaluer automatiquement des algorithmes.

7.5.Evaluation automatique

Pour confirmer notre idée, « les apprenants utilisent fréquemment les mêmes stratégies pour résoudre un problème », nous avons implémenté un deuxième outil appelé AlgoTest. Il permet de faire une évaluation automatique des propositions des apprenants, voir la figure 7.3.

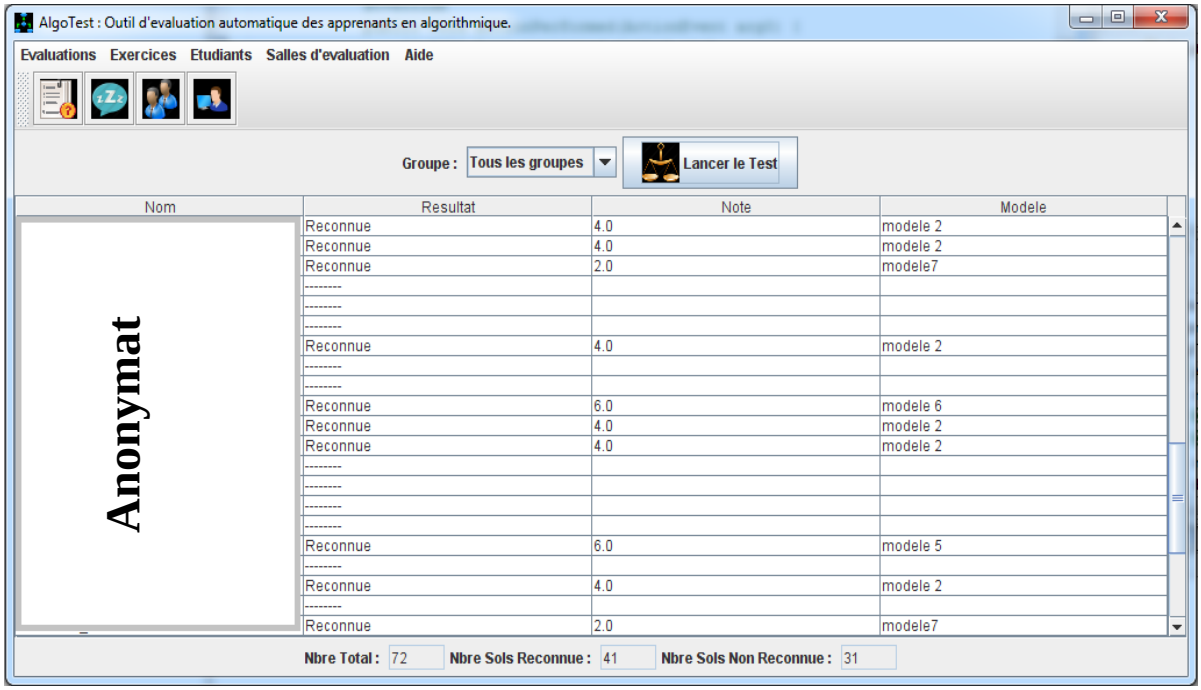

Figure 7.3 Outil AlgoTest

En fait, l'outil AlgoTest est la première version de notre projet, voir le chapitre précédent (section évaluation basée compréhension). Il permet de vérifier pour chaque proposition d'étudiant s'il y en a un modèle correspondant dans la base des modèles. Si oui, il renvoie le nom du modèle, et attribue une note à la proposition.

Dans une première expérimentation, nous avons pris quatre groupes correspondant à trois enseignants différents. Après l'évaluation des copies de ces groupes par leurs enseignants, nous avons saisi les algorithmes de chacun à l'aide de l'outil AlgoEditor. Les résultats obtenus sont présentés dans la figure 7.3 (ci-dessus), on les a détaillés dans le tableau 7.2.

En analysant ces résultats, nous pouvons constater que le taux de reconnaissance est plus de 50%. Ce pourcentage peut se varier en fonction de l'exercice. Nous avons également noté

que le nombre des propositions incorrectes est plus important que celui des propositions correctes. Les propositions incorrectes sont des propositions comportant des erreurs telle que la difficulté de faire toutes les tâches souhaitées (toutes les propositions sont correctes syntaxiquement).

		Grp1	**Grp2**	**Grp3**	**Grp 4**	**Total**
		19	**16**	**18**	**19**	
Reconnue	Correct	2	1	1	1	5
	Incorrect	10	8	9	9	36
Non Reconnue	Correct	1	2	1	1	5
	Incorrect	6	5	7	8	26
Taux de reconnaissance		**63**	**56**	**55**	**52**	**56**

Tableau 7.2 Evaluation automatique

Nous avons également constaté qu'en utilisant un seul modèle (modèle 2) nous avons obtenu 34 solutions reconnues. Ce qui représente 47% du taux de reconnaissance des propositions reconnues. Cela peut être justifié par le fait que ce modèle peut avoir 3 alternatives grâce à l'utilisation des descriptifs. En outre, parmi les propositions non reconnues, nous avons constaté que plusieurs propositions correspondaient à ce modèle, mais avec une légère différence. Par exemple certains étudiants remplacent l'instruction : if (t [i] + 1 ==t [i+1]) par deux instructions , qui sont : x = t [i +1] - t [i] ; et if (x == 1) . Par ailleurs nous avons constaté que les propositions non reconnues se ressemblent.

7.6.Retour sur les notes

Lors de la saisie des algorithmes proposition depuis les copies des étudiants, nous avons récupéré les notes attribuées par les enseignants. Nous avions l'intention de comparer ces notes, attribuées manuellement, avec les notes attribuées automatiquement par l'outil. Ci-dessous sont présentés les histogrammes contenant les notes de trois groupes. La couleur

bleue représente les notes attribuées automatiquement, et la couleur rouge représente celles attribuées manuellement (notons que la note est sur 6 points).

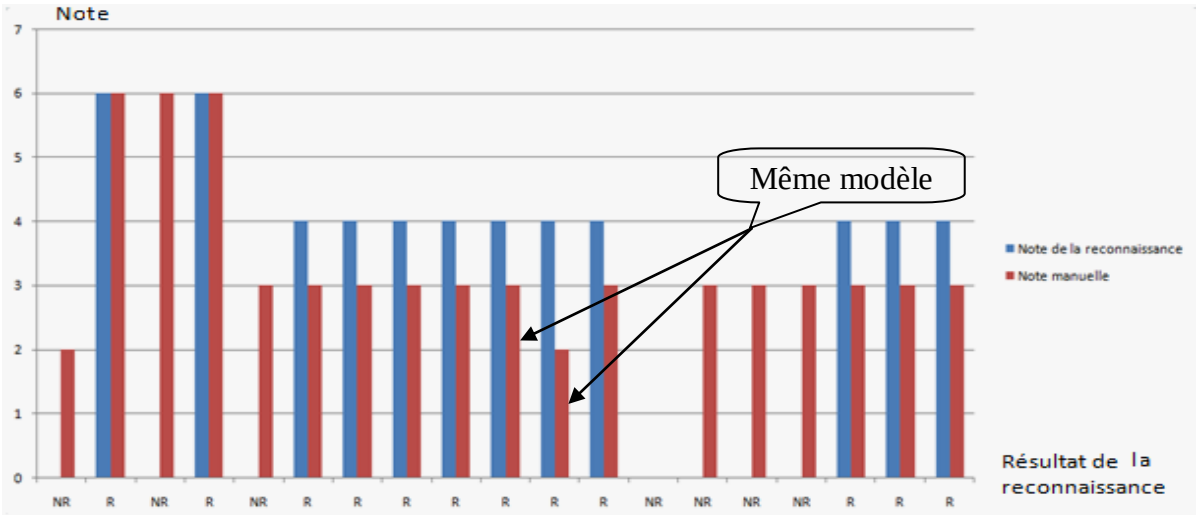

Figure 7.4 Notes : automatiques Vs manuelles (groupe1)

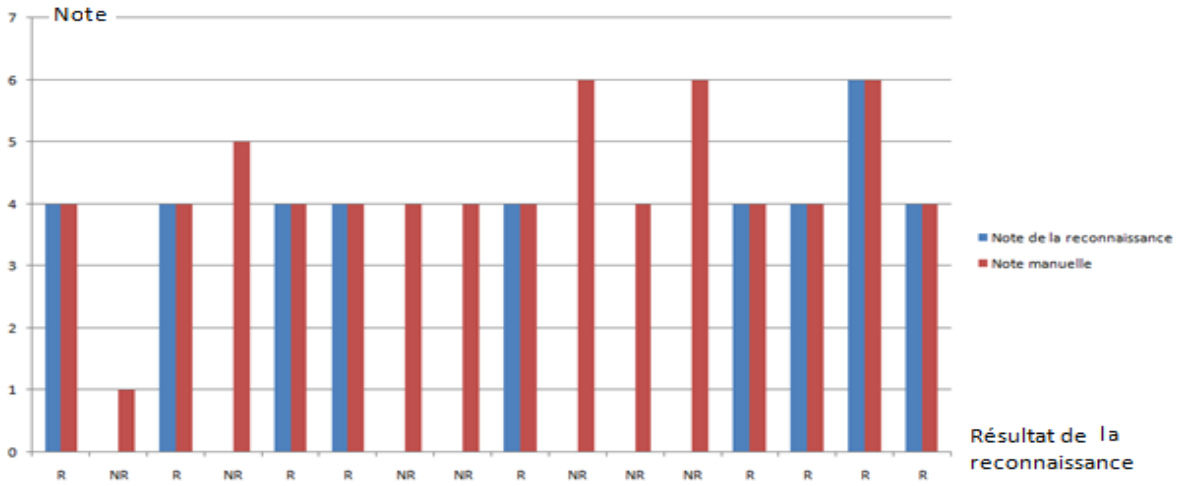

Figure 7.5 Notes : automatiques Vs manuelles (groupe2)

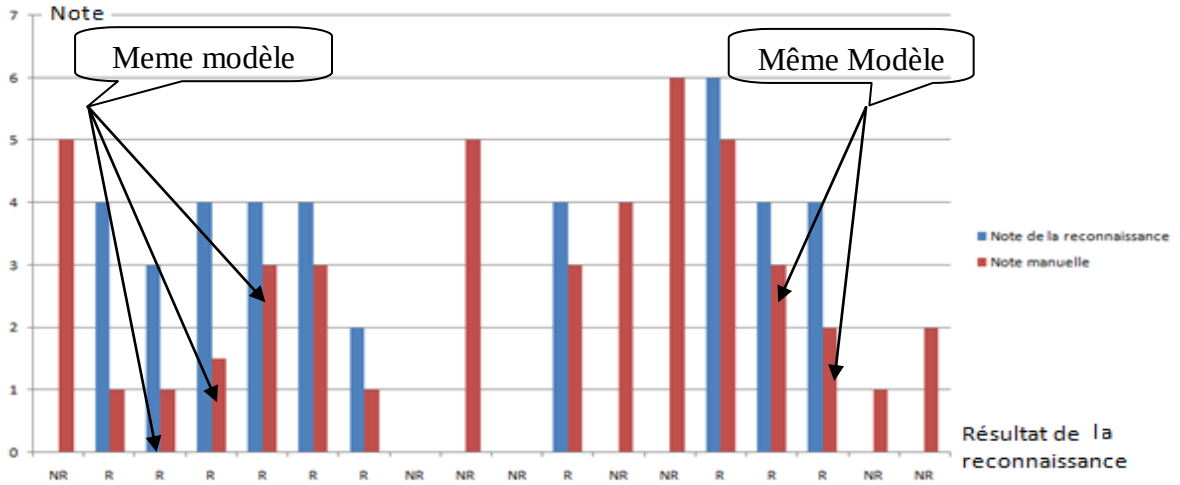

Figure 7.6 Notes : automatiques Vs manuelles (groupe3)

En analysant les figures présentées ci-dessus, nous constatons que les notes automatiques et manuelles sont les mêmes dans le groupe 2 pour les copies reconnues, et diffèrent légèrement dans le groupe 1 (d'un point). Cette différence devient évidente dans le groupe 3. Ce qui est remarquable aussi, est que dans le groupe 1, l'enseignant a attribués deux notes différentes à deux solutions ayant le même modèle, or l'outil leur a attribués la même note. Le même cas est apparu dans le groupe 3, mais cette fois ci la différence est considérable, elle est de deux points. Nous pouvons localiser cette différence entre les trois évaluateurs humains, qui ont attribué des notes différentes à des propositions similaires voir la figure ci-dessous.

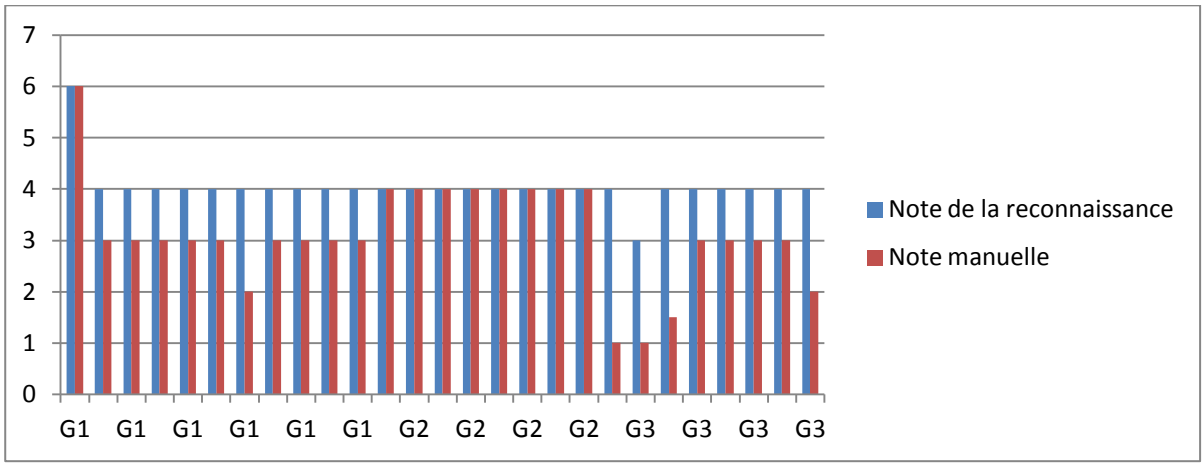

Figure 7.7 Différence des notes entre enseignants pour le modèles 2

La figure 7.7 présente les notes attribuées (couleur rouge) par les évaluateurs humains aux propositions reconnues, qui correspondaient au modèle 2. Ainsi que les notes attribuées automatiquement (en bleu). Nous constatons bien que l'enseignant du groupe 1 a attribué la note 2 à une seule proposition, et 3 à toues les autres. L'enseignant du groupe 2 a attribué 4 à toutes les propositions, et l'enseignant du troisième groupe a attribués des notes différentes. Avant d'aller plus loin dans la discussion de ces résultats, nous avons demandé à d'autres enseignants de nous rendre service.

7.7.Retour sur la reconnaissance

Nous avons présenté à des collègues d'autres universités l'exercice sur lequel nous avions travaillé, les modèles et les propositions des étudiants et nous leur avons demandé de vérifier s'il existe pour chaque proposition un modèle correspondant. La figure 7.8 présente les résultats de cette vérification, entre la reconnaissance manuelle et la reconnaissance automatique.

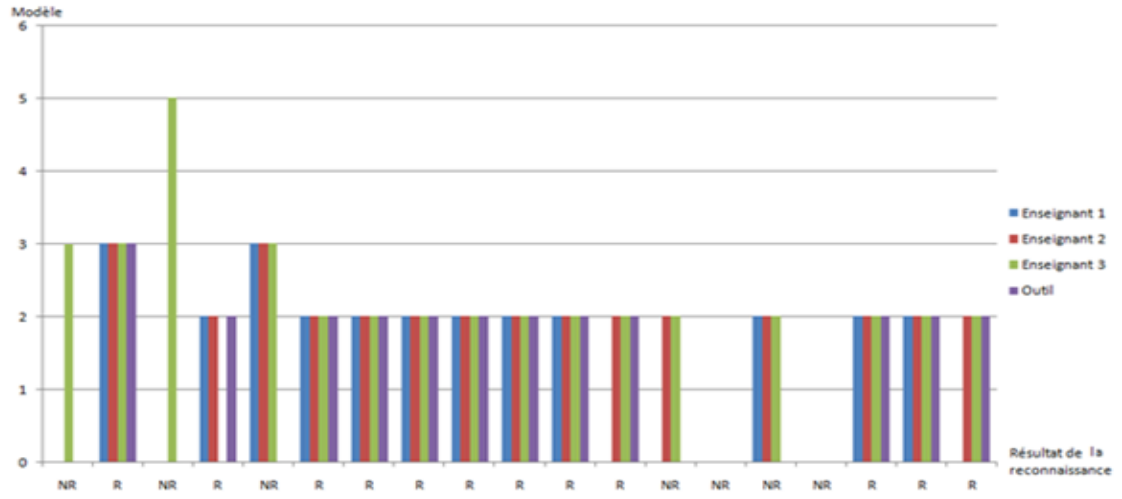

Figure 7.8 Reconnaissance : automatique Vs manuelle (groupe 1)

Selon cette figure, l'outil a reconnu 12 propositions parmi les 19 du groupe 1. En comparant cette reconnaissance automatique avec celle manuelle (de trois enseignants), nous constatons que parmi les 12 propositions reconnues, l'outil AlgoTest a le même résultat que les 3 évaluateurs pour 9 propositions. Pour les trois propositions restantes, un seul enseignant ne les a pas reconnues. Nous avons constaté aussi, que parmi les sept propositions non reconnues, deux propositions ont été reconnues par au moins deux enseignants, et deux ont on été reconnues par un seul enseignant.

Pour les propositions reconnues du groupe 2, la reconnaissance automatique donne les mêmes résultats que la reconnaissance manuelle (de deux enseignants). Un de ces deux enseignants, estiment que parmi les sept propositions non reconnues, 3 sont proches du modèle 1 et deux sont proches au modèle 2, voir la figure 7. 9.

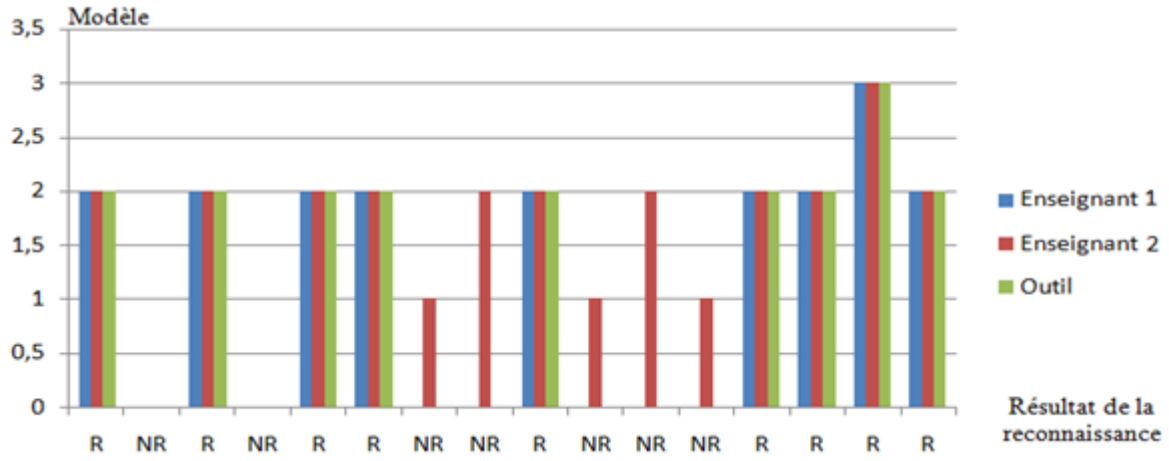

Figure 7.9 Reconnaissance : automatique Vs manuelle (groupe 2)

La figure 7.10, représente la comparaison entre les résultats de la reconnaissance manuelle (de deux enseignants) et l'outil AlgoTest, pour les propositions du groupe 3. Au moins, un des deux évaluateurs est pense que 2 propositions non reconnues doiventt correspondre au modèle 3. Pour les propositions reconnues, le résultat de la reconnaissance manuelle est le même pour 7 sur 10. Pour les 3 restantes, les deux enseignants pense qu'une proposition doit correspondre au modèle 7 au lieu du modèle 2, et un seul enseignant a pu reconnaitre les deux autres propositions.

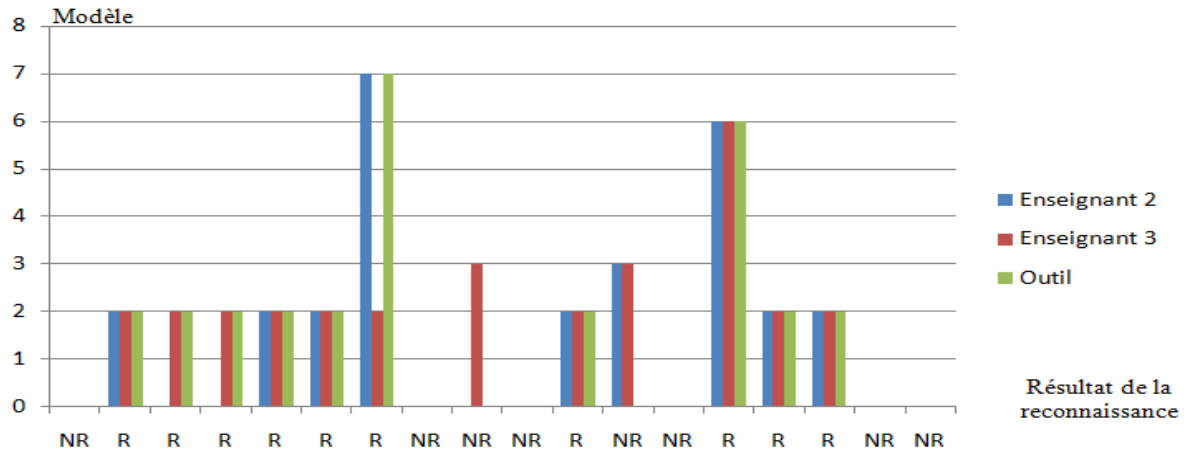

Figure 7.10 Reconnaissance : automatique Vs manuelle (groupe 3)

7.8.Discussion

Cette étude expérimentale, nous a permis de constater que le taux de reconnaissance de 56% est proche de celui obtenu manuellement qui était de 61%. Nous constatons aussi que le nombre de propositions reconnues est 41 sur 72. Parmi ces 41 propositions, 36 étaient incorrectes et seulement 5 étaient correctes. C'est-à-dire 87% étaient des propositions incorrectes. D'un autre coté, parmi les 36 incorrectes, il y avait 34 qui correspondaient au modèle 2, ce qui représente 94%. Le nombre total des propositions correctes était 10 sur 72. 5 ont été reconnues, et 5 n'ont pas été reconnues. Nous pensons que cela est dû à la nature difficile de la matière.

La différence des notes attribuées manuellement avec celle attribuées automatiquement, reflète la difficulté d'évaluation des disciplines telles que l'algorithmique, où il y a une attribution d'une note globale. Autrement dit, il est difficile d'avoir un barème détaillé pour l'évaluation. Cette différence d'attribution des notes, nous a permis de constater qu'un enseignant peut donner des notes différentes à des propositions similaires. Il est possible aussi qu'une même proposition évaluée par des enseignants différents puisse avoir des notes

différentes. La nature humaine peut aussi être la source de la subjectivité de l'évaluation manuelle (fatigue, concentration, nervosité, etc).

Nous avons constaté que les deux évaluations manuelle et automatique, affectent les propositions aux mêmes modèles, dans les trois groupes. Il y a seulement un désaccord entre les évaluateurs humain lors de la reconnaissance des propositions qui ne sont pas reconnues par l'outil, chacun a une estimation. Cependant, nous avons constaté qu'il y a un accord dans deux ou trois cas, que nous devons étudier (pourquoi ces propositions n'ont pas été reconnues).

7.9.Conclusion

Dans ce chapitre, nous avons présenté notre étude expérimentale. Nous avons commencé par présenter le contexte de notre expérimentation et la population avec laquelle nous avons travaillée. Nous avons fait une comparaison entre les notes attribuées manuellement et celles attribuées automatiquement. Avant de discuter tous les résultats nous avons fait une deuxième comparaison entre les résultats de la reconnaissance automatique et celles obtenues par des évaluateurs humains.

Après cette étude expérimentale, nous pouvons dire qu'il est possible de connaitre à quel modèle, une proposition peut correspondre. Cela permet de déduire la stratégie adoptée, ses points forts et ses faiblesses. Par conséquent, un feed-back détaillé, et pertinent pourra être offert aux enseignants. Ce qui permet de prendre les bonnes décisions pédagogiques.

Troisième partie

Conclusions et Perspectives

Chapitre 8

Conclusion générale

« *On ferait beaucoup plus de choses si l'on en croyait moins d'impossibles* »

‘Malesherbes’

8.1.Bilan

L'évaluation est une action essentielle dans une formation, Elle peut se faire de différentes manières et peut avoir plusieurs objectifs. Quelque soit la méthode adoptée et l'objectif fixé, nous pouvons dire que Evaluer sert à prendre une décision.
D'une part, les méthodes automatiques d'évaluation de programmes se basent sur l'utilisation des jeux de tests ou sur des mesures depuis le code d'un programme telle que sa complexité. Ces méthodes permettent d'avoir une idée globale sur le programme, elles ne distinguent pas les bons programmes des mauvais. En plus, elles ne permettent pas d'avoir une idée détaillée sur la façon dont les apprenants utilisent pour résoudre un problème. D'autre part, La compréhension de programmes joue un rôle important dans la phase de maintenance des applications existantes fréquemment développées par d'autres personnes. Elle consiste à analyser le code écrit par ces personnes pour en extraire des informations de haut niveau d'abstraction. Ces informations seront utiles lors de la mise à jour d'un code existant, sa documenter, etc.

Aussi, pour évaluer automatiquement des apprenants en algorithmique, nous avons proposé une approche basée sur la compréhension de programmes. Cette approche consiste à analyser les algorithmes écrits par des apprenants, pour en extraire le maximum des informations sur ce qu'ils veulent faire, leurs compétences et faiblesses. Pour faire cette analyse un outil d'évaluation a été développé. Il utilise une base de modèles, et applique des techniques de compréhension de programmes sur les propositions des apprenants pour reconnaitre à quel modèle chacune des propositions correspond.

L'approche proposée favorise la créativité des apprenants, et a comme objectif de renvoyer un feedback détaillé sur la solution proposée. En fait, l'apprenant peut exprimer ses idées et écrire n'importe quel algorithme. L'utilisation de la visualisation permet aux apprenants de se libérer de la syntaxe et de se concentrer sur le problème. Cela néglige l'aspect syntaxique, et favorise le raisonnement.

Les modèles de propositions abstraient les stratégies adoptées par les apprenants lors de la résolution des problèmes. Ainsi, nous avons une idée sur la façon dont les étudiants les (stratégies) utilisent pour atteindre un objectif. Les modèles de propositions incorrectes nous permettent de localiser la défaillance de raisonnement, et nous pouvons par conséquent y remédier.

La compréhension humaine de toutes les solutions est garantie. Cependant, elle peut être couteuse en temps et en ressources. D'un autre coté, elle est subjective.

La compréhension automatique peut réduire (voire libérer) les enseignants de la tâche d'évaluation, et produire un feed-back détaillé sur les apprenants. Cependant malgré sa rapidité, disponibilité et objectivité, elle est difficile à mettre en œuvre, et elle nécessite une bonne validation.

8.2. Perspectives

Il est possible d'affirmer que le travail mené dans le cadre de cette thèse a pour ambition de fournir aux enseignants un outil leur permettant d'évaluer finement des apprenants en algorithmique. Cependant, ce travail ne s'arrête pas là, et on peut dire que beaucoup de perspectives expérimentales et de recherche seront poursuivies à court et à long terme.

8.2.1. Perspectives expérimentales

Comme expliqué tout au long de ce manuscrit l'évaluation se base sur l'analyse statique des algorithmes sans les exécuter. Notre première perspective est d'offrir aux apprenants et enseignants la possibilité d'exécuter ces algorithmes. Cela permet aux apprenants de vérifier périodiquement leurs productions avant de les remettre. Elle permet aussi d'aider les enseignants à comprendre les solutions non reconnues.

Une deuxième perspective est de faire une expérimentation avec plus d'exercices et construire une base de problème assez riche. Cela nous permettra de valider notre approche et réutiliser l'expertise des évaluations précédentes.

Pour minimiser les interventions de l'expert humain lors de l'évaluation des propositions non reconnues. Ces propositions peuvent être comparées entre elles, puisque nous avons constaté qu'elles se ressemblent.

8.2.2. Perspectives de recherche

Ce travail de recherche nous a ouvert plusieurs perspectives de recherches dont les principales sont :

La première perspective consiste en la production d'un feedback après la reconnaissance d'une proposition. Ce feed-back permettra aux enseignants de prendre des décisions pédagogiques, il doit refléter le raisonnement des apprenants, leurs compétences et

faiblesses. Cependant, il faut penser à comment modéliser ces informations, les stocker, et les présenter (voire exploiter) par la suite.

La seconde perspective consiste en la construction d'un modèle de groupe d'apprenants. Comme les apprenants adoptent des stratégies similaires pour résoudre un problème. Il serait intéressant de construire des classes d'apprenants, et ce en comparant leurs propositions. D'où les questions : comment faire cette comparaison ? Quelle serait la nature d'un modèle de groupe ?

Annexes

A.1. Modèles de proposition

```
1. Algorithme modele 4;
2. Var        Nombre : i;
3.            Tableau de nombres : t[10];
4. Debut
5.    For (i = 0; i<10; 1)
6.    BeginFor
7.       Read(t[i]);
8.    EndFor
9.    For (i = 0; i<10; 1)
10.   BeginFor
11.      If ( t[i]+1==t[i+1] ) Then
12.      BeginIf
13.         If ( i+1=10 ) Then
14.         BeginIf
15.            Write(les elements du tableau sont consecutifs);
16.         Else
17.            i=i+1;
18.         EndIf
19.      Else
20.         Write(Les elements du tableau ne sont pas consecutifs);
21.      EndIf
22.   EndFor
23. Fin
```

```
1. Algorithme modele 5;
2. Var        Nombre : i;
3.            Tableau de nombres : t[10];
4.            Booleen : Consec;
5. Debut
6.    For (i = 0; i<10; 1)
7.    BeginFor
8.       Read(t[i]);
9.    EndFor
10.   i=0;
11.   Consec=Vrai;
12.   While ( i<9ANDConsec==Vrai )
13.   BeginWhile
14.      If ( t[i]+1!=t[i+1] ) Then
15.      BeginIf
16.         Consec=Faux;
17.      EndIf
18.   EndWhile
19.   If ( Consec==Vrai ) Then
20.   BeginIf
21.      Write(Les elements du tableau sont consecutifs);
22.   Else
23.      Write(Les elements du tableau ne sont pas consecutifs);
24.   EndIf
25. Fin
```

```
1. Algorithme modele 6;
2. Var        Nombre : i;
3.            Tableau de nombres : Tab[10];
4. Debut
5.    For (i = 0; i<10; 1)
6.    BeginFor
7.       Read(Tab[i]);
8.    EndFor
9.    For (i = 0; i<10; 1)
10.   BeginFor
11.      If ( Tab[i]+1!=Tab[i+1] ) Then
12.      BeginIf
13.         Write(Les elements du tableau ne sont pas consecutifs);
14.         i=11;
15.      EndIf
16.   EndFor
17.   If ( i==9 ) Then
18.   BeginIf
19.      Write(Les elements du tableau sont consecutifs);
20.   EndIf
21. Fin
```

A.2. Description XML d'un modèle de proposition (Modèle 2)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><Solution evaluation="Evaluation_CC_01_1213" exercice="Exo4_Consecutifs" nom="modele 2">
<Dec><Var nom="i" taille="0" type="1"/><Var nom="t" taille="10" type="4"/></Dec><Body>
    <For cond="i &lt; 10" pas="1" valInitiale="0" var="i">
        <BeginFor><exp left="i" name="exp1" op="&lt;" rangLeft="" rangRight="" right="10"/></BeginFor>
            <Read range="i" value="t"/>
        <EndFor/>
    </For>
    <For cond="i &lt; 10" pas="1" valInitiale="0" var="i">
        <BeginFor><exp left="i" name="exp1" op="&lt;" rangLeft="" rangRight="" right="10"/></BeginFor>
        <If avecSion="1" cond="t[i] + 1 == t[i+1]">
            <BeginIf>
                <exp left="exp2" name="exp1" op="==" rangLeft="" rangRight="exp3" right="t"/>
                <exp left="t" name="exp2" op="+" rangLeft="i" rangRight="" right="1"/>
                <exp left="i" name="exp3" op="+" rangLeft="" rangRight="" right="1"/>
            </BeginIf>
            <If avecSion="0" cond="exp1">
                <BeginIf>
                    <exp left="i" name="exp1" op="==" rangLeft="" rangRight="" right="10"/>
                </BeginIf>
                <Write range="" value="Les elements du tableau sont consecutifs"/>
                <EndIf/>
            </If>
            <Else>
                <Write range="" value="Les elements du tableau ne sont pas consecutifs"/>
            </Else>
            <EndIf/>
            </If>
            <EndFor/>
        </For>
    </Body>
```

```
<Tasks>
    <Task Deb="5" Fin="8" Mere="" Name="Lecture" Note="1" Type="1"/>
    <Task Deb="9" Fin="20" Mere="" Name="Test" Note="5" Type="1"/>
</Tasks>
<Constraints>
    <Constraint NumLig="7" Penalite="" Src="" TaskName="Lecture" Trg="" Type="1"/>
    <Constraint NumLig="9" Penalite="0" Src="" TaskName="Test" Trg="" Type="1"/>
    <Constraint NumLig="11" Penalite="0" Src="" TaskName="Test" Trg="" Type="1"/>
    <Constraint NumLig="9" Penalite="2" Src="" TaskName="Test" Trg="" Type="3"/>
    <Constraint NumLig="10" Penalite="0" Src="" TaskName="Test" Trg="" Type="3"/>
    <Constraint NumLig="20" Penalite="0" Src="" TaskName="Test" Trg="" Type="3"/>
    <Constraint NumLig="13" Penalite="2" Src="" TaskName="Test" Trg="" Type="3"/>
    <Constraint NumLig="14" Penalite="0" Src="" TaskName="Test" Trg="" Type="3"/>
    <Constraint NumLig="16" Penalite="0" Src="" TaskName="Test" Trg="" Type="3"/>
    <Constraint NumLig="" Penalite="" Src="" TaskName="Lecture" Trg="" Type="6"/>
    <Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8">
    <exp left="t" name="exp2" op="&lt;" rangLeft="i" rangRight="exp1" right="t"/>
    <exp left="i" name="exp1" op="+" rangLeft="" rangRight="" right="1"/></Constraint>
    <Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8"><exp left="exp2" name="exp3" op="==" rangLeft="" rangRight="i" right="t"/>
    <exp left="i" name="exp1" op="+" rangLeft="" rangRight="" right="1"/><exp left="t" name="exp2" op="-" rangLeft="exp1" rangRight="" right="1"/>
    </Constraint><Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8">
    <exp left="t" name="exp2" op="&gt;" rangLeft="exp1" rangRight="i" right="t"/><exp left="i" name="exp1" op="+" rangLeft="" rangRight="" right="1"/>
    </Constraint><Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8">
    <exp left="t" name="exp2" op="==" rangLeft="i" rangRight="exp1" right="t"/><exp left="i" name="exp1" op="+" rangLeft="" rangRight="" right="1"/>
    </Constraint><Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8">
    <exp left="t" name="exp2" op="==" rangLeft="exp1" rangRight="i" right="t"/><exp left="t" name="exp1" op="+" rangLeft="i" rangRight="" right="1"/>
    </Constraint><Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8">
    <exp left="t" name="exp3" op="!=" rangLeft="i" rangRight="" right="exp2"/><exp left="i" name="exp1" op="+" rangLeft="" rangRight="" right="1"/>
    <exp left="t" name="exp2" op="-" rangLeft="exp1" rangRight="" right="1"/></Constraint>
    <Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8"><exp left="t" name="exp2" op="!=" rangLeft="i" rangRight="exp1" right="t"/>
    <exp left="i" name="exp1" op="+" rangLeft="" rangRight="" right="1"/></Constraint>
    <Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8"><exp left="t" name="exp1" op="==" rangLeft="i" rangRight="i" right="t"/>
    </Constraint><Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8">
    <exp left="t" name="exp2" op="!=" rangLeft="i" rangRight="" right="exp1"/><exp left="t" name="exp1" op="+" rangLeft="i" rangRight="" right="1"/>
    </Constraint><Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8">
    <exp left="t" name="exp2" op="!=" rangLeft="i" rangRight="" right="exp1"/><exp left="t" name="exp1" op="+" rangLeft="i" rangRight="" right="1"/>
    </Constraint><Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8">
    <exp left="exp2" name="exp3" op="==" rangLeft="" rangRight="" right="0"/>
    <exp left="i" name="exp1" op="+" rangLeft="" rangRight="" right="1"/><exp left="t" name="exp2" op="-" rangLeft="exp1" rangRight="i" right="t"/>
    </Constraint><Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8">
    <exp left="exp2" name="exp3" op="!=" rangLeft="" rangRight="exp1" right="t"/><exp left="i" name="exp1" op="+" rangLeft="" rangRight="" right="1"/>
    <exp left="t" name="exp2" op="+" rangLeft="i" rangRight="" right="1"/></Constraint>
    <Constraint NumLig="11" Penalite="" Src="" TaskName="" Trg="" Type="8"><exp left="exp2" name="exp1" op="!=" rangLeft="" rangRight="i" right="t"/>
    <exp left="t" name="exp2" op="+" rangLeft="i" rangRight="" right="1"/></Constraint>
    </Constraints>
</Solution>
```

A.3. Code Source de la méthode de compréhension des algorithmes

```
public void evaluateMePlease(String pSolutionPath)
    {
        reinitModelsStat(models);
        SolutionEvaluation curEval = new SolutionEvaluation();
        curEval.setSourcePath(pSolutionPath);
        curEval.setXmlPath(pSolutionPath);
        currentSolFile = new File(curEval.getXmlPath());
        try {
            currentSolDoc = builder.parse(currentSolFile);
        } catch (SAXException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        curEval.setVariables(getVarNames(curEval.getXmlPath()));
        curEval.setEtat(false);
        curEval.setNote(0);
        curEval.setName(getModelName(curEval.getXmlPath()));
        curEval.setRootSol(currentDoc.getDocumentElement());
        entierCouter nbreInst = new entierCouter();
        getInstrctionNb(curEval.getRootSol(),nbreInst);
        curEval.setNbreInst(nbreInst.getVal());
        chargeSolInstructions(curEval);
        Iterator<Model> mods = models.iterator();
        while(mods.hasNext())
        {
            Model curMdl = (Model) mods.next();
            Iterator <Variable> it = curMdl.getVariables().iterator();
            while (it.hasNext())
                {
                    Variable item = (Variable) it.next();
                    item.setVarClone(getVarClones(item,curEval));
                }
                isThereCLPlease(curMdl, curEval);
            }
           Model cand = wichOneShoudWeTry(models);
           while(cand !=null && !curEval.isEtat())
           {
               tryThisModel(cand, curEval);
               cand.setSeen(true);
               cand = wichOneShoudWeTry(models);
           }
           evaluations.add(curEval);
      }
```

A.4. Code Source de la méthode de vérification de la correspondance Modèle/ proposition

```
public boolean tryThisModel(Model pModel, SolutionEvaluation pEvaluation)
{
    reinitSolInstStat(pEvaluation.getListeInstruction());
    ArrayList<Task> taches = pModel.getTasks();
    ArrayList<Constraint> contraintes = pModel.getConstraints();
    pEvaluation.setNote(0);
    currentFile = new File(pModel.getPath());
    try { currentDoc = builder.parse(currentFile);
        }
        catch (SAXException e) {e.printStackTrace();}
        catch (IOException e)  {e.printStackTrace();}
    currentSolFile = new File(pEvaluation.getXmlPath());
    try { currentSolDoc = builder.parse(currentSolFile);
        }
        catch (SAXException e) {e.printStackTrace();}
        catch (IOException e) {e.printStackTrace();}
    Element root = currentDoc.getDocumentElement(); // modele
    Element rootSol = currentSolDoc.getDocumentElement(); // solution
    ArrayList<Task> ListOfRecTasks = new ArrayList<Task>();
    ArrayList<Task> ListOfDesiredTasks = pModel.getTasks();
    int possibleDeb = 0;
    int possibleDebClone = 0;
    int fin;
    boolean wanted = false;
    Task recTask = null;
    entierCouter nbInst = new entierCouter();
    nbInst.setVal(0);
    getInstrctionNb(rootSol, nbInst);
    // S'il n'y a aucune instruction
    if(nbInst.getVal()==0)
    {
        // Le modèle correspondant est le modele vide
        pEvaluation.setEtat(true);
        pEvaluation.setModel("Modele Vide.");
    }
    else
    {
        Iterator itTasks = taches.iterator(); // tant qu'il y a une tache
        while (itTasks.hasNext())
        {
            //Charger la  tache suivante
            Task curTask = (Task) itTasks.next();
            String debTask = curTask.getDeb();
            String finTask = curTask.getFin();
            int curInst = Integer.parseInt(debTask); // on va commencer la reconnaissance à partir d'ici
                Instruction inst = new Instruction(); // dans laquelle on va charger l'instruction de debut
                ArrayList<Instruction> samInst = new ArrayList<Instruction>(); // elle va contenir les instructions similaire
                                                                               //   Ces instructions representeront des débuts possible
                entierCouter numLine = new entierCouter();
                inst.setNum(1);
                getInsctruction(root, inst, curInst);
                inst.setNum(curInst);numLine.setVal(1);
                // Charger les debuts possibles (dans la solution)
                // Poue eviter la rechargement de tache
                // exemple : Pour : lecture, on charge les deb possib : deux pour (lecture et test)
                //           Pour : test, on recharger les deb possibles  : deux pour (lecture et test)
                if(!alreadyImportedDebPossible(curInst, samInst))
                    getAllInsctructionWithNumLine(rootSol, samInst,inst.getNoeud().getNodeName(),numLine);
                boolean res = true;
                boolean desordre ;
                float note = 0;
                ArrayList<String> listeDesordre = new ArrayList<String>();
                ArrayList<String> listeMissed = new ArrayList<String>();
                ArrayList<String> listeReconus = new ArrayList<String>();
                ArrayList<String> dejaVus = new ArrayList<String>();
                Iterator<Instruction> itPossibleInst = samInst.iterator();
                while(itPossibleInst.hasNext())
                {
                    // on essaye avec ce debut
                    Instruction possibleDebInst = (Instruction) itPossibleInst.next();
                    possibleDeb = possibleDebInst.getNum();
                    possibleDebClone = possibleDeb;
                    fin = Integer.parseInt(finTask);
                    curInst = Integer.parseInt(debTask);
                    int MaxTaskSol ;
                    wanted = true; // on suppose que la tache voulue est ici
                    desordre = false;
                    Instruction instMod = new Instruction();
                    instMod.setNum(1);
                    inst.setNum(curInst);
                    // Si ce deb possible n'a pas été traité, et n'a pas été reconuu
                    if(!alreadyTreated(possibleDeb, ListOfRecTasks)&&!pEvaluation.getListeInstruction().get(possibleDeb-1).isReconue())
                    {
                        // Verifier toutes les instructions de la taches
                        while (curInst <= fin && wanted)
                        {
                            instMod = new Instruction();
                            instMod.setNum(1);
```

```
                    Instruction instSol = new Instruction();
                    instSol.setNum(1);
                    getInsctruction(root, instMod, curInst);
                    instMod.setNum(curInst);
                    getInsctruction(rootSol, instSol, possibleDeb);
                    instSol.setNum(possibleDeb);
                    if (areTheseSameInstruction(instMod, instSol,pModel,pEvaluation))
                    {
                        // Les deux instrcutions sont les memes
                        dejaVus.add(Integer.toString(possibleDeb)); // marquer comme déja vue ()
                        listeReconus.add(Integer.toString(possibleDeb)); // maruqer comme reconnue
                        possibleDeb ++;
                        curInst++;
                        continue;
                    }
                    else
                    // les deux instructOs ne sont pas les memes
                    // Verifier l'existence d'une inst similaire
                    if ( ! isItMissed(instMod,pModel,pEvaluation,Integer.parseInt(curTask.getDeb()),possibleDebInst.getNum(),dejaVus)),
                    {
                        // elle est présente
                        dejaVus.add(Integer.toString(possibleDeb));
                        listeReconus.add(Integer.toString(possibleDeb));
                        // mentionner le desordre
                        if(!desordre)
                        {
                            desordre = true;
                            listeDesordre.add(Integer.toString(instMod.getNum()));
                        }
                        possibleDeb ++;
                        curInst++;
                        continue;
                    }
                    // si elle est absente
                    else
                    {
                        // les deux instructions sont differentes
                        // je dois verifier qu'il n y a pas de contrainte d'absence sur cette inst
                        // verification des contraintes d'absence
                        if(instMod.getNoeud().getNodeName().equals("If")&&instSol.getNoeud().getNodeName().equals("If"))
                        {
                            if(isThereCondLike(instSol,pModel.getConstraints(),pModel.getVariables(),pEvaluation.getVariables()))
                            {
                                // elle est présente
                                dejaVus.add(Integer.toString(possibleDeb));
                                listeReconus.add(Integer.toString(possibleDeb));
                                possibleDeb ++;
                                curInst++;
                                continue;
                            }
                        }
                        if (canIbeMissed(instMod.getNum(),pModel.getConstraints()))//-----------------------------------------
                        {
                            listeMissed.add(Integer.toString(curInst));
                            curInst++;
                            continue;
                        }
                           else
                           {
                               wanted = false;
                               break;
                           }
                       }
                   }
               }
               else
                   wanted = false;
               if (wanted) // la tache a été reconnue
               {
                   recTask = new Task();
                   recTask.setName(curTask.getName());
                   recTask.setDeb(Integer.toString(possibleDebClone));
                   recTask.setFin(Integer.toString(possibleDeb - 1));
                   setRecognizedInstructions(pEvaluation, listeReconus); // signaler ces inst comme reconnues
                   recTask.setType(curTask.getType());
                   note = Float.parseFloat(curTask.getNote());
                   if(desordre)
                       note = note - getPenDesordre(pModel.getConstraints(),listeDesordre);
                   if(listeMissed.size() != 0)
                       note = note - getPenDesordre(pModel.getConstraints(),listeMissed);
                   recTask.setNote(Float.toString(note));
                   ListOfRecTasks.add(recTask);
                   pModel.setNbOfRecTasks(pModel.getNbOfRecTasks()+1);
                   //MAJ de la note
                   pEvaluation.setNote(pEvaluation.getNote()+note);
               }
           }
       }
```

```
177             // Verifier le desodre de taches.-----------------------------------------------------------------------------------
178             pEvaluation.setTaches(ListOfRecTasks);
179             boolean SolReconnue = true;
180             Iterator<Task> itDesTasks = ListOfDesiredTasks.iterator();
181             int i = 0;
182             while(itDesTasks.hasNext())
183             {
184                 Task itemDesTask = (Task) itDesTasks.next();
185                 if(!isTaskPresent(itemDesTask.getName(), ListOfRecTasks))
186                 {
187                     ListOfDesiredTasks.get(i).setReconue(false);
188                     if(!canThisTaskBeMissed(itemDesTask.getName(), pModel.getConstraints()))
189                         SolReconnue = false;
190                     else
191                         ListOfDesiredTasks.get(i).setCanBeMissed(true);
192                 }
193                 else
194                     ListOfDesiredTasks.get(i).setReconue(true);
195                 i++;
196             }
197             if (SolReconnue)
198             {
199                 pEvaluation.setEtat(true);
200                 pEvaluation.setModel(pModel.getName());
201             }
202             else
203                 pEvaluation.setNote(0);
204         }
205         return pEvaluation.isEtat();
206     }
```

Bibliographie

(Alkhatib, 1992) Alkhatib, G., (1992) 'The maintenance problem of application software: An empirical analysis'. Journal of Software Maintenance: Research and Practice, 1:83-104.

(Allal, 1991) Allal, L., (1991) 'Vers une pratique de l'évaluation formative, matériel de formation continue des enseignants', Bruxelles, De Boeck.

(Allal, 2005) Allal, L. (2005), 'L'évaluation formative de l'apprentissage : revue de publication en langue française', L'évaluation formative- Pour un meilleur apprentissage dans les classes secondaires.

(Antoniol et al, 2001) Antoniol, G., Casazza, G., Di Penta, M., & Fiuten, R., (2001) 'Object-oriented design patterns recovery'.Journal of Systems and Software, 59, 181 – 196.

(Arnold, 1994) Arnold, R.S., (1994) 'Software Reengineering: a quick history'. Communication sof the ACM, 37(5):13-14, May1994.

(Arnow et Barshay, 1999) Arnow, D., & Barshay, O., (1999) 'WebToTeach: An interactive focused programming exercise system'. Proceedings of the 29th ASEE/IEEE Frontiers in Education Conference, 12A9/39 – 12A9/44.

(Basili et Mills, 1982) Basili, VR., & Mills, H.D., (1982) 'Understanding and documenting programs'. IEEE Transactions on Software Engineering, 8 : 270-83.

(Becker, 2003) Becker, K., (2003) 'Grading programming assignments using rubrics'. In Proceedings of the 8th annual conference on Innovation and technology in computer science education, Greece, 253.

(Ben-Ari, 1998) Ben-Ari, M., (1998) 'Constructivism in Computer Science Education'. 29th ACM SIGCSE Technical Symposium on Computer Science Education, Atlanta Georgia: ACM press.

(Bettini et al, 2004) Bettini, L., Crescenzi, P., Innocenti, G., Loreti, M., & Cecchi, L., (2004) 'An Environment for SelfAssessing Java Programming Skills in Undergraduate First Programming Courses'. In Proceedings of the 4th IEEE International Conference on Advanced Learning Technologies, Finland, 161 – 165.

(Bloom, 1956) Bloom, B.S. (Ed.) (1956) 'Taxonomy of Educational Objectives: The Classification of Educational Goals'. Handbook I. Cognitive Domain. New York: David McKay Company, Inc.

(Blum, 1991) Blum, B.I., (1991) 'The software process for medical applications'. In T.Timmers and B. I. Blum, editors, Software Engineering in Medical Informatics, pages3-25, North-Holland. Elsevier Science Publishers B.V.

(Boehm, 1981) Boehm, B. W., (1981) 'Prentice-Hall', Inc., New Jersey. Chapter 30 of this book shows how the COCOMO model can be used to estimate the cost of maintenance work using different cost drivers.

(Boehm, 1983) Boehm, B.W., (1983) 'The economics of software maintenance'. In R. S. Arnold, editor, Proceedings, Workshop on Software Maintenance , pages9-37, Silver Spring, MD. IEEE Computer Society Press, 1983.

(Bouacha et Bensebaa) Bouacha, I. & Bensebaa, T. (2013) 'Evaluation des apprenants en algorithmique', EIAH 2013, Vol. 29, pp.7–8.

(Bouhineau et Puitg, 2011) Bouhineau, D. & Puitg, F., (2011) 'Évaluations de solutions d'exercices d'algorithmique, « à la main » versus « automatiques par jeux d'essai »', Conférence Environnements Informatiques pour l'Apprentissage Humain, EIAH 2011, Mons, Belgique, ISBN: 978 2 87325 061 4, 5/2011

(Bratitsis et Dimitracopoulou, 05) Bratitsis, T., & Dimitracopoulou, A. (2005), « Data recording and usage interaction analysis in asynchronous discussions : The D.I.A.S. System », AIED Workshops

(Brooks, 1983) Brooks, R., (1983) 'Towards a Theory of the Comprehension of Computer Programs'. International Journal of Man-Machine Studies, Vol. 18, pp 543-554.

(Brown, 1991) Brown, P., (19910) 'Integrated hypertext and program understanding tools'. IBM Systems Journal, 30 (3) : 363-92.

(Bull, 1999) Bull J., 1999 'Computer-Assisted Assessment : Impact on Higher Education Institutions', Journal of Educational Technology and Society, vol. 2(3), p. 123-126.

(Carter, 2003) Carter, J., English, J., Ala-Mutka, K., Dick, M., Fone, W., Fuller, U., & Sheard, J., (2003) 'How shall we assess this? ' ACM SIGCSE Bulletin, 35(4), 107 – 123.

(Charle, 1990) Charle, H., (1990) 'Evaluation, les règles du jeu'. ESF.

(Charle, 2000) Charle, H., (2000) 'Evaluer, règles du jeu', E.S.F. éditeur, Paris, 2000 6ème édition.

(Cheang et al, 2003) Cheang, B., Kurnia, A., Lim, A., & Oon, W.-C. (2003) 'On automated grading of Programming Assignments in an academic institution'. *Computers & Education , 41, 121 – 131.*

(Checkstyle) Checkstyle. [Computer Software] Retrieved November 15, 2004, from http://checkstyle.sourceforge.net/.

(Chen, 2004) Chen, P. (2004) 'An Automated Feedback System for Computer Organization Projects'. *IEEE Transactions on Education , 47 , 232 – 240.*

(Chen, 2004) Chen, P., (2004) 'An Automated Feedback System for Computer Organization Projects'. *IEEE Transactions on Education , 47 , 232 – 240.*

(Corbi, 1989) Corbi, T.A., (1989) 'Program understanding:Challengeforthe1990s'. IBM Systems Journal,28 (2): 294-306, 1989. Discusses the importance of program

comprehension in maintenance and makes a case for developing tools that assist programmers in understanding existing code.

(Cox et Clark, 1998) Cox, K., & Clark, D. (1998) 'The Use of Formative Quizzes for Deep Learning'. *Computers & Education, 30, 157 – 167.*

(David, 2003) David J.-P., (2003) 'Modélisation et production d'objets pédagogiques', Sciences et Techniques Éducatives, avril.

(Darwin, 1990) Darwin, I., (1990) 'Checking C Programs with Lint'. New York: OReilly & Associates.

(Dean et McCune, 1983) Dean, J.S., & McCune, B.P., (1983) 'An informal study of maintenance problems'. In R. S. Arnold, editor, Proceedings, Workshop on Software Maintenance, pages 137-9, Silver Spring, MD. IEEE Computer Society Press.

(Delozanne et Grugeon, 04) Delozanne, E., Grugeon, B. (2004) 'Pépites et lingots : des logiciels pour faciliter la régulation par les enseignants des apprentissages en algèbre', Cahiers Éducation et Devenir, vol. Hors série, « Les TIC à l'école : miracle ou mirage ? », p. 82-92, septembre.

(Dintilhac et RAK, 2005) Dintilhac, J.P. & Rak, I., (2005) 'Evaluation de la technologie en collège, académie de montpellier'. Technical report, juillet 2005.

(Dromey, 1995) Dromey, R.G., (1995) 'A model for software product quality'. IEEE Transactions on Software Engineering, 21, 146 – 162.

(Edwards, 2004) Edwards, S. (2004) 'Improving Student Performance by Evaluating How Well Students Test Their Own Programs'. ACM Journal of Educational Resources in Computing, 3(3), 1 – 24.

(Ellsworth et al, 2004) Ellsworth, C., Fenwick, J., & Kurtz, B., (2004) 'The Quiver System'. InProceedings of the 35th SIGCSE technical symposium on Computer Science Education, US, 205 – 209.

(Ellsworth et al, 2004) Ellsworth, C., Fenwick, J., & Kurtz, B., (2004) 'The Quiver System'. InProceedings of the 35th SIGCSE technical symposium on Computer Science Education, US, 205 – 209.

(English, 2004) English, J., (2004) 'Automatic Assessment of GUI Programs using JEWL'. In ²Proceedings of 9th annual conference on Innovation and technology in computer science education, UK, 137 – 141.

(Foxley, 1999) Foxley, E. (1999) 'Ceilidh Documentation on the World Wide Web'. Retrieved November 15, 2004, from http://www.cs.nott.ac.uk/~ ceilidh/papers.html.

(Gallagher, 1990) Gallagher, K., (1990) 'Surgeon's Assistant limits side effect'. IEEE Software, 7 : 64, May, 1990.

(GCC) GCC compiler. [Computer software] Retrieved November 15, 2004, from http://gcc.gnu.org/.

(Geneviève, 1995) Geneviève, MEYER., (1995) 'Evaluer : pourquoi? Comment?', Hachette, 1995.

(George, 2001) George, S., (2001) 'SPLACH : un environnement informatique support d'une pédagogie par projet', thèse de doctorat, Université du Maine, apprentissage collectif à distance.

(Gilbert, 1995) Gilbert, L. (1980), 'Evaluation continue et examen', Précis de Docimologie. Edition LABOR.

(Guibert et al, 2005) Guibert, N., Guittet, L., & Girard, P., (2005) 'Initiation à la Programmation « par l'exemple »: concepts, environnement, et étude d'utilité'. Acte de colloque EIAH'05, Montpellier: 25-27 Mai, 461-466.

(Green et al, 1984) Green, B., Bock, R., Humphreys, L., Linn R., & Reckase M., (1984) 'Technical guidelines for assessing computerized adaptive tests', Journal of Educational Measurement, vol. 21, session 4, p. 347-360.

(Halstead, 1977) Halstead, M.H., (1977) 'Elements of Software Science'. New York: Elsevier North-Holland.
(Halstead, 1977) Halstead, M.H., (1977) 'Elements of Software Science'. New York: Elsevier North-Holland.

(Hansen et Ruuska,2003) Hansen, H., & Ruuska, M., (2003) 'Assessing time-efficiency in a course on data structures and algorithms'. In Proceedings of the 3rd Annual Finnish/Baltic Sea Conference on Computer Science Education, Finland.

(Higgins et al, 2002) Higgins, C., Symeonidis, P., & Tsinsifas, A., (2002) 'Diagram-based CBA using DATsys and CourseMaster'. *In Proceedings of the International Conference on Computers in Education , 367 – 372.*

(Higgins et al, 2003) Higgins, C., Hergazy, T., Symeonidis, P., & Tsinsifas, A., (2003) 'The CourseMarker CBA system: Improvements over Ceilidh'. *Education and Information Technologies , 8 , 287 – 304.*

(Hoc et al, 1990) Hoc, J.-M., Green, T.R.G., Samurcay, R.,Gilmore, D.J., (1990) 'Psychology of Programming'. Academic Press, London.

(Hoffnagle et Beregi, 1985) Hoffnagle, G.F., & Beregi, WE. 1985 'Automating the software development process'. IBM Systems Journal, 24(2):102-20, 1985.

(Howles, 2003) Howles, T. (2003) 'Fostering the growth of a software quality culture'. *ACM SIGCSE Bulletin ,35(2), 45 – 47.*

(Hung et al, 1993) Hung, S.-L., Kwok, L.-F., & Chan, R., (1993) 'Automatic programming assessment'. Computers & Education, 20, 183 – 190.

(Jackson et Usher, 1997) Jackson, D., & Usher, M. (1997) 'Grading Student Programs using ASSYST'. *Proceedings of the 28th SIGCSE technical symposium on Computer science education, USA, 335 – 339.*

(Johnson-Laird, 1983) Johnson-Laird, P.N., (1983) 'Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness'. Cambridge University Press, Cambridge.

(Juwah, 2003) Juwah, C., 2003 'Using Peer Assessment to Develop Skills and Capabilities', United States Distance Learning Association, vol. janvier 2003, p. 39-50.

(Kay et al, 1994) Kay, D., Scott, T., Isaacson, P., & Reek, K., (1994) 'Automated grading assistance for student programs'. In Proceedings of the 25th SIGCSE technical symposium on Computer science education, USA, 381 – 382.
(Korhonen, et Malmi 2000) Korhonen, A., & Malmi, L. (2000) 'Algorithm simulation with automatic assessment'. *Proceedings of 5th annual conference on Innovation and technology in computer science education , Finland, 160 – 163.*

(Layzell et Macaulay, 1990) Layzell, P.J., & Macaulay,L., (1990) 'An investigation into software maintenance : Perception and practices'. In Conference on Software Maintenance, pages 130-140, LosAlamitos, California. IEEEComputer Society.

(Lehman, 1989) Lehman, M.M., (1989) 'Uncertainty in computer application and its control through the engineering of software'. Journal of Software Maintenance : Researchand Practice, 1 (l) : 3-27.

(Letovsky, 1986) Letovsky, S. (1986) 'Cognitive processes in program comprehension'. In E. Soloway and S. Iyengar, editors, Empirical Studies of Programmers, Chapter5, pages 58-79. Ablex Publishing Corporation, NewJersey.

(Lientz et Swanson, 1980) Lientz, P. & Swanson, E. B., (1980) 'Software Maintenance Management'. Addison- Wesley Publishing Company, Reading, Massachusetts.

(Lindsay et Norman, 1977) Lindsay, P.H., & Norman, D.A., (1977) 'Human Information Processing : An Introduction to Psychology'. Academic Press, NewYork.

(Luck et Joy, 1999) Luck, M., & Joy, M. (1999) 'A Secure On-line Submission System'. *Software – Practice and Experience , 29 , 721 – 740.*

(MacNish, 2000) MacNish, C., (2000) 'Java facilities for automating analysis, feedback and assessment of laboratory work'. Computer Science Education, 10, 147 – 163.

(Martin et McClure, 1982) Martin, J., McClure, C., (1982) 'Maintenance of Computer Programming' ,volumes I and II. SavantResearchStudies, Carnforth, Lancashire.

(MAZ et Milani ,05) Mazza, R., Milani, C., (2005) 'Exploring usage analysis in learning systems : gaining insights from visualisations', AIED Workshops (AIED'05), juillet.

(McCabe, 1976) McCabe, T.J., (1976) 'A Complexity Measure'. IEEE Transactions on Software Engineering, SE-2, 308 – 320.

(McCracken et al, 2001) McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., & Morris, D. (2003) 'Automatic Grading of Student's Programming Assignments: An Interactive Process and Suite of Programs'. *In Proceedings of the 33rd ASEE/IEEE Frontiers in Education Conference , S3F-1 – S3F-5.*

(Mengel et Ulans, 1999) Mengel, S.A., & Ulans, J.V., (1999) 'A case study of the analysis of the quality of novice students programs'. In Proceedings of the 12th Conference on Software Engineering Education and Training, 40 – 49.

(MER et Acef. ,04) Merceron, A., Acef, K. (2004), 'Train, store, analyse for more adaptive teaching', Technologies de l'Information et de la Connaissance dans l'Enseignement supérieur et l'Industrie, vol. octobre 2004, p. 52-58, 2004.

(Michaelson, 1996) Michaelson, G., (1996) 'Automatic Analysis of Functional Program Style'. InProceedings of Australian Software Engineering Conference, 38 – 46.
(Michaelson, 1996) Michaelson, G. (1996). 'Automatic Analysis of Functional Program Style'. In Proceedings of Australian Software Engineering Conference, 38 – 46.

(Morris, 2003) Morris, D., (2003) 'Automatic Grading of Student's Programming Assignments: An Interactive Process and Suite of Programs'. In Proceedings of the 33rd ASEE/IEEE Frontiers in Conference, S3F-1 – S3F-5. Education Conference, S3F-1 – S3F-5.

(MOSS) MOSS. [Computer software] Retrieved November 15, 2004, from http://www.cs.berkeley.edu/*aiken/moss.html.

(Olson, 1988) Olson, D.M., (1988) 'The reliability of analytic and holistic methods in rating students' computer programs'. In Proceedings of the 19th SIGCSE technical symposium on Computer science education,

(Oman et Cook, 1990) Oman, P.W., & Cook, C.R., (1990) 'A taxonomy for programming style'. In Proceedings of the 1990 ACM Annual Conference on Cooperation, 244 – 250.

(Oman, 1990) Oman, P., (1990) 'Maintenance Tools'. IEEE Software, 7 : 59, May.

(Osborne et Chikofsky, 1990) Osborne, WM., & Chikofsky, E.J., (1990) 'Fitting pieces to the maintenance puzzle'. IEEE Software, 7 : 11-12, January.

(Padula, 1993) Padula, A., (1993) 'Use of a program understanding taxonomy at Hewlett-Packard'. In Proceedings, 2nd Workshop on Program Comprehension, pages 66-70, Los Alamitos. IEEEComputer Society.

(Parikh,et Zvegintzov, 1983) Parikh, G., & Zvegintzov, N., (1983) 'Tutorial on Software Maintenance'. IEEE Computer Society Press, Silver Spring, Maryland.

(Pennington et Grabowski, 1990) Pennington, N., Grabowski, B., (1990) 'Psychology of programming'. In J.-M.Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore, editors, Psychology of Programming, Chapter 1.3, pages 45-62. Academic Press, London.

(Pennington,, 1987) Pennington, N., (1987) 'Stimulus structures and mental representations in expert comprehension of programs'. Cognitive Psychology, 19 (3) : 295-341, July.

(Postman, 1979) Postman, N., (1979) 'Teaching as a Consering Activity'. Delacorte.

(Prechelt et al, 2002) Prechelt, L., Malpohl, G., & Philippsen, M., (2002) 'Finding plagiarisms among a set of programs with Jplag'. Journal of Universal Computer Science, 8, 1016 – 1038.

(Purao et Vaishnavi, 2003) Purao, S., & Vaishnavi, V., (2003). 'Product metrics for object-oriented systems'. ACM Computing Surveys, 35, 191 – 221.

(Purao, et Vaishnavi, 2003) Purao, S., & Vaishnavi, V., (2003) 'Product metrics for object-oriented systems'. ACM Computing Surveys, 35, 191 – 221.

(Rajlich, 1990) Rajlich, V., (1990) 'Vifor transforms code skeletons to graphs'. IEEE Software, 7 : 60, May.

(Reek, 1989) Reek, K.A. (1989) 'The TRY system -or- how to avoid testing student programs'. *In Proceedings of the 20th SIGCSE technical symposium on Computer science education , USA, 112 – 116.*

(Rees, 1982) Rees, M.J., (1982) 'Automatic assessment aid for pascal programs'. SIGPLAN Notices, 17, 33 – 42.

(Rintala, 2002) Rintala, M., (2002) 'Tutnew memory management library'. Retrieved November 15, 2004, from http://www.cs.tut.fi/*bitti/tutnew/english/.

(Robins et al, 2003) Robins, A., Rountree, J., & Rountree, N. (2003) 'Learning and teaching programming: A review and discussion'. *Computer Science Education , 13 , 137 – 172.*

(Rolland et al, 2000) Rolland, M.C., Arenilla, L., Roussel, M.P., & Gossot, B., (2000) 'Dictionnaire de pédagogie'. Bordas.

(Ruehr et Orr, 2002) Ruehr, F., & Orr, G. (2002) 'Interactive program demonstration as a form of student program assessment'. Journal of Computing in Small Colleges, 18, 65 – 78.

(Saikkonen et al, 2001) Saikkonen, R., Malmi, L., & Korhonen, A., (2001) 'Fully automatic assessment of programming exercises'. Proceedings of 6th annual conference on Innovation and technology in computer science education, UK, 133 – 136.

(Saikkonen et al, 2001) Saikkonen, R., Malmi, L., & Korhonen, A., (2001) 'Fully automatic assessment of programming exercises'. Proceedings of 6th annual conference on Innovation and technology in computer science education, UK, 133 – 136.

(Schleime et al, 2003) Schleimer, S., Wilkerson, D., & Aiken, A., (2003) 'Winnowing: Local algorithms for document fingerprinting'. Proceedings of the 2003 ACM SIGMOD international conference on Management of data, USA, 76 – 85.

(Schorsch, 1995) Schorsch, T., (1995) 'CAP: an automated self-assessment tool to check Pascal programs for syntax', logic and style errors. In Proceedings of the 26th SIGCSE technical symposium on Computer science education, 168 – 172.

(Schorsch, 1995) Schorsch, T. (1995). 'CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors'. In Proceedings of the 26th SIGCSE technical symposium on Computer science education, 168 – 172.

(Shneiderman et Mayer, 1979) Shneiderman, B., & Mayer, R., (1979) 'Syntactic semantic interactions in programming behaviour : a model'. International Journal of Computer and Information Science, 8 : 219-38.

(Shneiderman, 1980) Shneiderman, B., (1980) 'Software Psychology'. Winthrop, Cambridge, Massachusetts.

(Sitthiworachart et Joy, 2003) Sitthiworachart, J., & Joy, M., (2003) 'Effective Peer Assessment for Learning Computer Programming'.Proceedings of the 8th annual conference on Innovation and technology in computer science education, UK, 122 – 126.

(Sommerville et Thomson, 1989) Sommerville, I., & Thomson,R., (1989) 'An Approach To The Support Of Software evolution'. The Computer Journal, 32(8): 386-98, 1989.

(Tamai et Torimitsu, 1992) Tamai,T., & Torimitsu,Y., (1992) 'Software life time and its evolution process over generations'. In Proceedings, Conference on Software Maintenance, 8th Conference, pages63-69, Orlando, Florida, IEEE Computer Society Press, November.

(Truong et al, 2003) Truong, N., Roe, P., & Bancroft, P., (2003) 'A Web Based Environment for Learning to Program'. In Proceedings of the 25th Australasian Computer Science Conference, Australia, 255 – 264.

(Truong et al, 2004) Truong, N., Roe, P., & Bancroft, P., (2004) 'Static Analysis of Students' Java Programs'. In Proceedings of the sixth Australian Computing Education Conference, New Zealand, 317 – 325.

(Vanek et Davis, 1990) Vanek, L., & Davis, L., (1990) 'Expert dataflow and static analysis tool'. IEEE Software, 7 : 63, May.

(Verco et Wise, 1996) Verco, K., & Wise. M., (1996) 'A comparison of automated systems for detecting suspected plagiarism'. The Computer Journal, 39, 741 – 750.

(VonMayrhauser, 1994) VonMayrhauser, A., (1994) 'Maintenance and evolution of software products'. In M. C. Yovits, editor, Advances in Computers, Chapter1, pages 1-49. Academic Press, Boston.

(Waters et Chikofsky, 1994) Waters, R.C., Chikofsky, E., (1994) 'Reverse engineering : progress along many dimensions'. Communications of the ACM,37(5):23-4,May 1994.

(Weiser, 1984) Weiser, M., (1984) 'Program slicing'. IEEE Transactions on Software Engineering, SE-10 (4): 352-57, July.

(Wiedenbeck, 1986) Wiedenbeck, S., (1986) 'Processes in computer program comprehension'. In E. Soloway and S. lyengar, editors, Empirical Studies of Programmers, Chapter4, pages 48-57. Ablex Publishing Corporation, NewJersey.
(Wilde, 1990) Wilde, N., (1990) 'Dependency analysis tool set prototype'. IEEE Software, 7: 65, May.

(Woit et Mason, 2003) Woit, D., & Mason, D., (2003) 'Effectiveness of online assessment'. In Proceedings of the 34thSIGCSE technical symposium on Computer science education, USA, 137 – 141.

(Xie et Engler, 2002) Xie, Y., & Engler, D., (2002) 'Using Redundancies to Find Errors'. In Proceedings of SIGSOFT 2002/ FSE-10, USA, 51 – 60.

(Zeller, 2000) Zeller, A., (2000) 'Making students read and review code'. In Proceedings of 5th annual conference on Innovation and technology in computer science education, Finland, 89 – 92.

(JPLAG) JPLAG. [Computer software] Retrieved November 15, 2004, from http://www.jplag.de/.

(Thorburn et Rowe, 1997) Thorburn, G., & Rowe, G., (1997) 'PASS: An automated system for program assessment'. Computers & Education, 29, 195 – 206.

Biographie

Expérience professionnelle

2008-2010 : Ingénieur développeur dans la boite de développement BIG Informatique.

2008-2012 : Enseignant vacataire à l'université Badji Mokhtar Annaba.

2012-Jusqu'à présent : Enseignant à l'école préparatoire aux sciences et techniques Annaba.

Stages

2013 (Septembre-Décembre) : Chercheur Junior dans le projet de coopération ROSE : Raisonnements Ontologiques pour les Systèmes d'enseignement (projet de coopération accord CMEP, entre EIAH de l'ESI Alger, EIAO & learning d'Annaba et le METAH de Grenoble).

2014 (Septembre-Novembre) : Chercheur Junior dans le projet de coopération ROSE.

Communications et publication

Bouacha. I, & Seridi. H, (2010) 'Development of context-aware web services using the MDA approach'. The Third International Conference on Web & Information Technologies, 16-19 juin 2010.

Seridi. H, Bouacha. I, & Benselim. M, (2012) 'Development of context-aware web services using the MDA approach'. IJWS 1(3): 224-241.

Bouacha, I. and Bensebaa, T. (2013) 'Evaluation des apprenants en algorithmique', EIAH 2013, Vol. 29, pp.7–8.

Bouacha, I. and Bensebaa, T. 'Automatic Comprehension of Algorithms for Algorithmic assessment,

http://www.inderscience.com/info/ingeneral/forthcoming.php?jcode=ijil