Construction d'un programme combinant exécution partielle et manipulation directe

Adam Michel, Daoud Moncef, Patrice Frison

Construction d'un programme combinant exécution partielle et manipulation directe

Michel Adam[1] Moncef Daoud[1]
Patrice Frison[2]
(1) Université Bretagne Sud, 56100 Lorient, France
(2) IRISA, Université Bretagne Sud, 56000 Vannes, France
michel.adam@univ-ubs.fr, moncef.daoud@univ-ubs.fr,
patrice.frison@univ-ubs.fr

RÉSUMÉ

La programmation est un art difficile à maîtriser. Dans le cas de la programmation impérative, la boucle est l'un des concepts majeurs à connaître. Pour les programmeurs novices, c'est aussi l'un des concepts les plus difficiles à comprendre et à mettre en oeuvre. Cette difficulté est d'autant plus grande quand un algorithme nécessite la mise en place d'une boucle à l'intérieur d'une autre boucle. On parle alors de boucles imbriquées. Dans cet article, nous proposons une méthode associée à un outil facilitant la conception et la programmation d'algorithmes avec boucles imbriquées. Cette méthode originale combine alternativement l'exécution partielle de la boucle externe du programme avec l'exécution manuelle par le programmeur de la boucle interne.

ABSTRACT

Construction of a program combining partial execution and direct manipulation

Programming is a difficult art to master. In the case of imperative programming, loop is one of the major concepts to know. For novice programmers, it is also one of the most difficult concepts to understand and implement. This difficulty is all the greater when an algorithm requires the implementation of a loop within another loop. We then speak of nested loops. In this article, we propose a method associated with a tool facilitating the design and programming of algorithms with nested loops. This original method alternately combines the partial execution of the outer loop of the program with the manual execution by the programmer of the inner loop.

MOTS-CLÉS : concept de boucles, boucles imbriquées, programmation par manipulation directe des données, visualisation d'algorithmes, environnements d'apprentissage interactifs.

KEYWORDS: loops concept, nested loops, programming by data direct manipulation, visualization of algorithms, interactive learning environnements.

1 Introduction

L'apprentissage de la programmation fait l'objet de nombreux travaux de recherche de par le monde. L'informatique étant désormais enseignée dès le primaire (Baron and Drot-Delange, 2016) dans la plupart des pays, de nouvelles approches ont été proposées pour adapter cet enseignement à un large public. La notion de pensée informatique (*computational thinking*) a été introduite par Wing (Wing, 2011). L'idée étant de montrer que de nombreux problèmes de la vie courante peuvent être traités

Atelier « Apprendre la Pensée Informatique de la Maternelle à l'Université », dans le cadre de la conférence Environnements Informatiques pour l'Apprentissage Humain (EIAH), Fribourg, Suisse, 07 juin 2021, pages 25–33,

de façon systématique et ainsi être programmés sur des ordinateurs. Dans ce contexte, le concept d'algorithme est défini ainsi que la notion fondamentale de boucle. Cependant, le concept de boucle reste difficile à mettre en pratique par les novices lorsqu'ils doivent programmer eux-mêmes un exercice. Dans cet article, nous nous intéressons plus particulièrement à l'apprentissage des boucles.

Le papier est organisé de la façon suivante. Dans la deuxième section, nous présentons le contexte de l'étude. La troisième section aborde la programmation par manipulation directe des données, illustrée par le système AlgoTouch. Puis, dans une quatrième section, nous montrerons comment construire des boucles imbriquées par une approche originale alternant exécution du programme et manipulation directe des données. Le principe est le suivant : la boucle externe s'exécute mais s'interrompt pour laisser à l'utilisateur le contrôle pour réaliser l'objectif de la boucle interne par manipulation directe des données. Ensuite, le programme de la boucle externe reprend et ainsi de suite. Nous conclurons par une analyse des avantages de l'approche proposée et ses perspectives.

2 Contexte de l'étude

Différentes approches ont été proposées pour aider à l'apprentissage de la programmation. L'approche débranchée (Drot-Delange, 2013) propose de décrire les actions à effectuer pour traiter un problème. La description doit être suffisamment précise pour qu'un opérateur humain puisse exécuter les actions décrites et ainsi atteindre l'objectif. L'informatique tangible (Papavlasopoulou et al., 2017) permet aux débutants d'interagir avec des objets physiques et transformer la logique du monde physique en logique du programme. La manipulation directe des données (Adam et al., 2019) permet de construire des programmes en réalisant directement sur un écran les opérations que doit effectuer une machine. Le logiciel traduit les actions de l'utilisateur en langage informatique. Ce programme peut ensuite être exécuté automatiquement.

Pour faciliter la programmation, des environnements éducatifs ont été proposés (Maloney et al., 2010) (Daly, 2013). Le programme est construit par assemblage de blocs qui s'emboitent comme des éléments de puzzle. Ceci simplifie l'apprentissage d'un langage de programmation. L'environnement d'exécution est très visuel, mettant en scène des avatars se déplaçant dans un univers virtuel. Pour les apprenants plus expérimentés, l'apprentissage s'effectue avec des outils plus standard du programmeur au sein d'un EDI (Environnement de Développement Intégré). L'outil au centre de l'EDI étant l'éditeur de texte permettant la saisie assistée du programme dans un langage particulier.

Pour aider à la compréhension et à la réalisation des boucles, diverses méthodes ont été proposées (Cetin et al., 2020). La lecture de code écrit par l'enseignant permet d'illustrer concrètement à travers d'exemples bien choisis comment programmer les boucles. Cet exercice peut être complété en demandant aux étudiants d'exécuter manuellement le programme. Il permet notamment à l'enseignant de vérifier que l'étudiant déroule correctement chaque pas de la boucle. Un autre exercice consiste à demander à un apprenant de compléter un programme constitué d'une boucle. Il manque, par exemple, la condition de sortie de boucle ou l'instruction d'évolution de la boucle (incrément d'un indice par exemple).

Des outils ont été mis au point pour suivre le comportement d'un programme. Le débogueur permet d'exécuter le programme instruction par instruction et de consulter le contenu des variables. Les logiciels de visualisation (SV pour Software Visualisation) ont été développés pour aider les apprenants à surmonter leurs difficultés cognitives dans l'apprentissage des concepts de programmation

(Sorva et al., 2013). Cependant ces systèmes de visualisation n'ont d'intérêt que si l'apprenant a un moyen d'intervenir dans le processus de construction (Hundhausen et al., 2002) : la visualisation du fonctionnement d'un algorithme ne suffit pas.

Si la construction de boucle est difficile pour les étudiants (Kordaki et al., 2008) (Teague and Lister, 2014) (Izu et al., 2016) (Mladenović et al., 2018), c'est d'autant plus vrai pour le cas des boucles imbriquées. Une approche possible consiste à remplacer une boucle interne par un appel de méthode (Caspersen and Kolling, 2009) : "*If you have a nested loop, move the inner loop into a separate method*". Le but de cette approche est d'isoler les parties de code des deux boucles, en s'intéressant à la mise en oeuvre de chacune d'elles séparément. Ce principe hérite d'un principe plus large que les auteurs ont nommé "The Mañana Principle" : "*When – during implementation of a method – you wish you had a certain support method, write your code as if you had it. Implement it later*". En fait, ce principe est souvent utilisé par les développeurs. Il consiste à implémenter une méthode non écrite avec un corps qui rend toujours le même résultat, compatible avec les spécifications. Dans cet article nous proposons également de remplacer une boucle interne par un appel de méthode. Mais contrairement à l'approche décrite précédemment dans laquelle la méthode nouvelle rend toujours le même résultat, nous proposons que l'utilisateur prenne le contrôle du programme pour réaliser lui-même les opérations de la méthode.

Ce dispositif a été mis en oeuvre au sein d'un logiciel appelé AlgoTouch que nous avons développé. Il fait partie des logiciels de manipulation directe et de visualisation. Il permet de manipuler visuellement et manuellement les données d'un programme (variables, tableaux, indices), de construire un programme par enregistrement de suites d'actions. Mais il permet aussi de visualiser l'exécution d'un programme déjà écrit. Un outil particulièrement utile d'une part à un enseignant pour montrer le principe de fonctionnement d'un algorithme, et d'autre part à un élève pour visualiser le déroulement de l'exécution de son programme (Adam et al., 2018). Une première expérience a montré que des étudiants avaient de meilleurs résultats sur la construction de boucles avec AlgoTouch, qu'en écrivant un programme en Python (Adam et al., 2019).

3 Programmation par manipulation des données : exemple du système AlgoTouch

Les systèmes de programation par manipulation directe des données appartiennent à la catégorie des systèmes de programmation par démonstration (Programming By Demonstration : PBD) introduits par Cypher et al. (Cypher and Halbert, 1993). L'idée principale est que le système observe ce que fait l'utilisateur et qu'un programme soit produit de façon automatique ou contrôlée. De nombreuses variantes ont été proposées dans la littérature. Nous nous intéressons ici aux systèmes PBD concernant la manipulation directe des données du programme. Ces systèmes, dont le précurseur est Pygmalion (Cypher and Halbert, 1993), utilisent la métaphore du tableau noir, c'est-à-dire la manipulation directe sur l'écran des objets de la programmation : variables, tableaux, indices (figure 1). Une mise en oeuvre de cette méthode est réalisée avec l'outil AlgoTouch (Frison, 2015).

Les opérations de base sur les variables sont concrètes. Ainsi, l'utilisateur d'AlgoTouch agit directement sur les données et visualise immédiatement le résultat obtenu. A titre d'exemple, l'affectation est effectuée en glissant le contenu d'une variable et en le déposant sur une autre variable. Toutes les actions réalisées directement sur les données correspondent à des instructions d'un langage de

programmation spécifique à AlgoTouch et inspiré du langage Eiffel (Meyer, 2009). Par exemple, faire glisser le contenu d'une variable *y* dans une variable *x* provoquera la génération de l'instruction `x = y`. Par des gestes simples, l'utilisateur peut aussi augmenter les valeurs d'indices, lancer des comparaisons, effectuer les opérations de base (addition, soustraction, multiplication, division). Une séquence d'actions peut être enregistrée et ensuite rejouée. Une partie du programme est créée et produite au fur et à mesure que le programmeur « *joue* » avec les données de l'algorithme.

Le code genéré peut être nommé dans une structure de base d'AlgoTouch appelée macro-instruction ou simplement macro. Une macro est une manière d'abstraire une suite d'instructions. Aussi, une macro peut contenir un appel à une autre macro. Ainsi, il est possible de décomposer un programme en plusieurs macros de manière à resoudre des problèmes complexes. Une macro est différente d'une fonction ou d'une procédure. Dans un souci de simplification, les notions de paramètres et de variables locales n'existent pas dans AlgoTouch. En effet, toutes les variables, les données du programme, sont globales et donc utilisables dans toutes les macros.

AlgoTouch autorise uniquement deux types de macros : la macro simple, une séquence d'instructions et la macro boucle. Cette dernière est composée de 4 parties distinctes :

- la partie *From* permet de définir clairement les instructions nécessaires avant la boucle ;
- la partie *Until* définit les conditions de sortie de la boucle. Ces conditions sont placées en séquence sans opérateur. La première condition est évaluée, si elle est vraie, la boucle s'arrête. Sinon, la seconde condition est évaluée et ainsi de suite ;
- la partie *Loop* constitue le cœur de la boucle. Elle est exécutée quand toutes les conditions de sortie sont fausses ;
- enfin la partie *Terminate* permet de définir les instructions éventuelles à effectuer après la boucle.

La figure 1 illustre la construction d'un programme de recherche dichotomique.

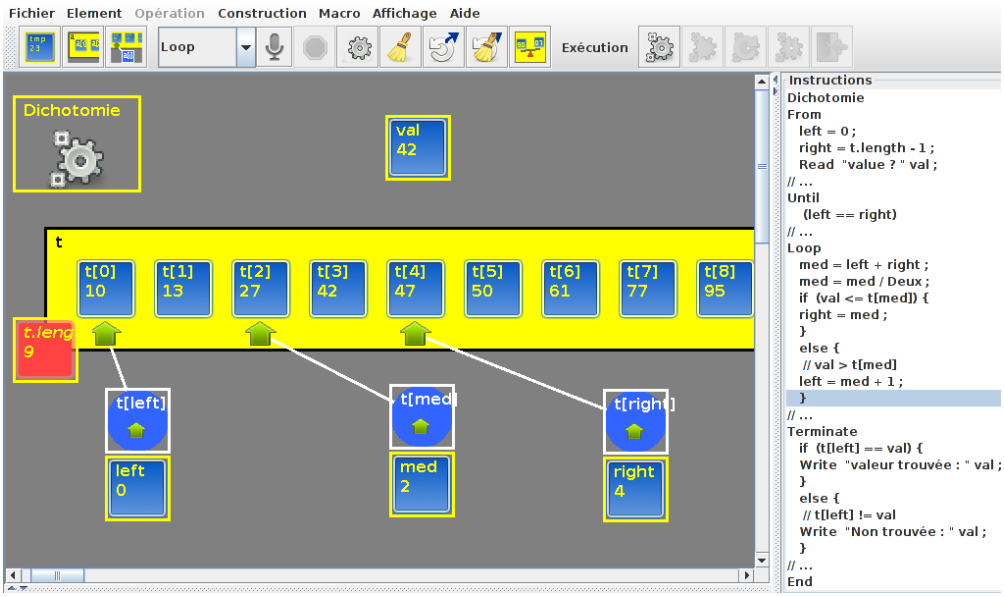

FIGURE 1 – Écran AlgoTouch, à gauche les données manipulées, à droite le code

4 Nouvelle approche de la construction de boucles imbriquées

La programmation de boucles imbriquées dans AlgoTouch est uniquement possible par appel de macro selon le "Mañana Principle" (Caspersen and Kolling, 2009). Une première macro permet de programmer une première boucle. Elle appelle une autre macro boucle dans son corps. L'appel d'une macro dans le corps de la boucle est équivalent à de l'imbrication. L'avantage est que le programmeur ne visualise à un instant donné qu'une seule boucle, qu'une seule macro. Il doit réaliser chacune des macros l'une après l'autre.

Dans cette partie, nous allons montrer qu'avec AlgoTouch, en utilisant la manipulation de données, il est possible de commencer par la boucle la plus externe. Le programmeur génère le code de la boucle externe. La macro représentant la boucle interne est appelée mais son code est vide. Le programmeur peut exécuter la macro externe. Quand la macro interne vide est appelée, l'exécution s'arrête et le programmeur reprend la main. Il peut alors manipuler directement les données de manière à produire le résultat attendu par l'exécution de la macro. Quand le résultat est produit, le programme peut reprendre son exécution. La boucle externe est ainsi testée.

Cette méthode est détaillée avec AlgoTouch à travers l'exemple du tri par insertion. Dans un premier temps, nous présentons le principe du tri, puis la programmation de la boucle externe et enfin celle de la boucle interne.

4.1 Exemple du tri par insertion

Le principe du tri par insertion étudié dans cette partie est de placer les valeurs au fur et à mesure de leur saisie à la bonne position dans le tableau. Ce tri est visuel : les premières valeurs du tableau sont toujours triées et il est facile de déterminer visuellement la place à attribuer à la dernière valeur saisie. La mise en oeuvre avec Algotouch est illustrée sur la figure 2.

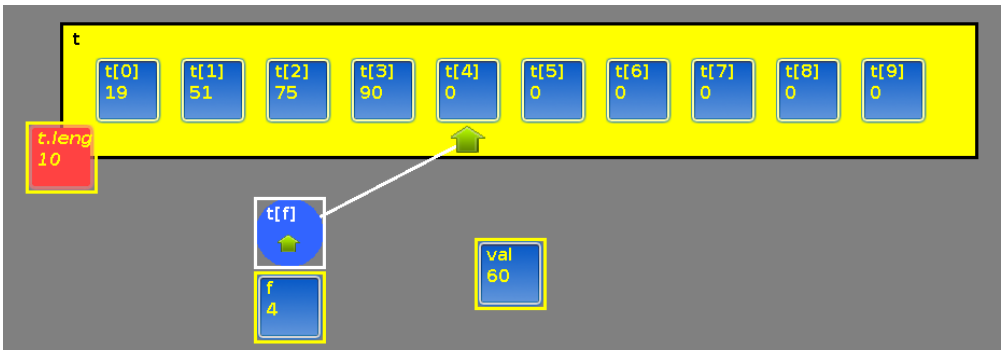

FIGURE 2 – Données du tri par insertion : *t* un tableau d'entiers, *val* valeur à insérer et *f* indice de la première case vide

L'algorithme répète la séquence suivante : saisir une valeur puis la placer dans le tableau déjà trié. Chaque étape de la séquence correspond à une boucle. La saisie s'effectue dans une boucle externe (macro *TriVolee*) et l'insertion dans une boucle interne (macro *Placer*). L'approche proposée étant descendante, nous commençons par concevoir et programmer la macro *TriVolee*, puis la macro *Placer*.

4.2 Construction de la macro *TriVolee*

Le rôle de la macro *Trivolee* est de répéter les actions suivantes : saisir une valeur, l'insérer dans le tableau en appelant la macro *Placer* et d'incrémenter le nombre de valeurs insérées jusqu'à ce que le tableau soit entièrement rempli. Le code généré pour la macro *TriVolee* est visible sur la partie droite de la figure 3.

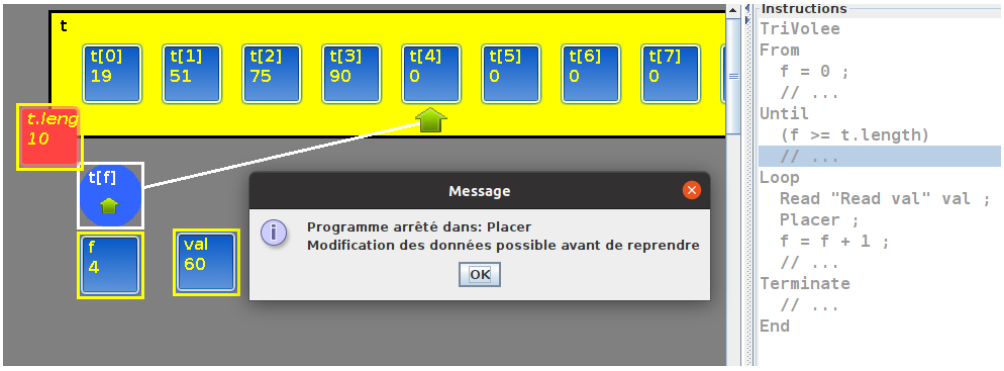

FIGURE 3 – Arrêt sur la macro *Placer* pendant l'exécution de la macro *TriVolee*

À ce niveau le code de la macro *Placer* n'est pas encore écrit. Pourtant, avec AlgoTouch, il est possible de tester la macro *TriVolee*. En effet, l'exécution d'une macro vide provoque l'arrêt de l'exécution. Sur la figure 3, on note l'arrêt sur la macro *Placer* pendant l'exécution de *TriVolee*. Le programmeur prend le contrôle car il a accès à toutes les variables du programme. Par manipulation directe des données, il insère la valeur au bon endroit dans le tableau et relance l'exécution au point d'arrêt. Sur l'exemple de la figure 3, l'utilisateur doit libérer la case t[2] et donc déplacer vers la droite les deux valeurs 75 et 90. En pratique, il doit d'abord déplacer la valeur 90 de t[3] dans t[4], puis la valeur 75 de t[2] dans t[3] et finalement placer la valeur 60 de val dans t[2].

Le programmeur s'est substitué à la macro *Placer*, et il a produit le résultat attendu à la fin de son exécution. À ce stade, il n'a pas eu besoin ni de la coder, ni même de connaitre l'algorithme à utiliser. Il a simplement constaté visuellement où placer la valeur.

Ainsi, la macro *TriVolee* peut être testée de manière unitaire. Le résultat produit ne dépend que de son code et des manipulations du programmeur. Évidemment, un test d'intégration ou fonctionnel sera à effectuer une fois le code de la macro *Placer* généré et testé.

Enfin, par les manipulations qu'il effectue, le programmeur peut déjà imaginer le comportement algorithmique de la macro *Placer*. Naturellement, il effectue des manipulations qui peuvent être répétitives et lui permettre de trouver plus facilement l'algorithme de la macro *Placer*. La génération du code de la macro doit en être facilitée comme illustré dans des travaux antérieurs (Adam et al., 2019).

4.3 Construction de la macro *Placer*

Le rôle de la macro *Placer* est d'insérer la valeur saisie dans le tableau. Évidemment, il existe plusieurs algorithmes pour réaliser la macro *Placer*. Nous avons donc effectué un choix parmi les différentes méthodes de placement existantes. L'algorithme décale les valeurs d'une position vers la

droite et s'arrête s'il trouve une valeur plus petite que celle à insérer ou s'il atteint le début du tableau. Le code généré pour la macro *Placer* est présenté sur la figure 4.

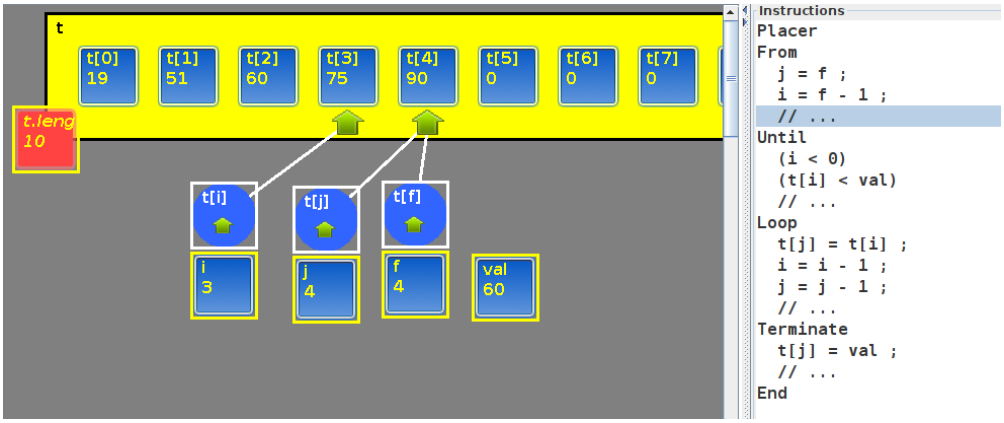

FIGURE 4 – Code la macro *Placer*

Une fois le code généré, il est possible de tester la macro *Placer* de manière unitaire. Ensuite, le test d'intégration peut être effectué en appelant la méthode *TriVolee*. Ce test est dit d'intégration car il teste le bon fonctionnement des deux macros. Dans la partie précédente, le test était unitaire, seul le code de *TriVolee* était complet, seule la macro était vérifiée.

4.4 Analyse

Dans cette partie, nous analysons *a priori* les apports de l'approche descendante avec AlgoTouch présentée dans la partie précédente. Toutes ces hypothèses devront être validées par des expérimentations.

Conception et "écriture" du code de manière concomitante
Avec des langages de programmation classique, la conception est effectuée de manière descendante et l'écriture du code ascendante. Les macros ou méthodes les plus internes sont écrites et testées en premier. Avec l'approche proposée, la conception, la programmation et les tests sont effectués de manière descendante.

Conception et programmation
Le novice ne se concentre que sur un seul niveau de boucle. Quand l'imbrication est sur plusieurs niveaux, le programmeur n'a pas besoin de déterminer à l'avance le nombre de macros qui seront nécessaires, ni toutes les variables. Il les crée en fonction de ses besoins. En générant d'abord la macro la plus externe, le programmeur définit le rôle de la macro interne.

Différentiation forte du comportement et du code
Lors de la construction de la macro externe *TriVolee*, pour notre exemple, seul le comportement de la macro interne *Placer* est utile. Il n'est pas nécessaire de connaître son code. Ainsi, le débutant peut aborder, par l'exemple, la différence entre le rôle de la macro, son algorithme et son code.

Aide à la conception de la macro interne
Quand le novice en programmation génère et exécute la macro externe *TriVolee*, il doit manipuler directement les données pour produire le résultat attendu par l'exécution de la macro interne *Placer*. Par ces manipulations, il pourrait progressivement identifier l'algorithme de la macro imbriquée. Son

codage devrait en être facilité.

Identification des variables nécessaires à chaque étape
Comme les variables ne sont créées par le programmeur qu'au fur et à mesure, le débutant en programmation découvre une première approche de la portée des variables. Une variable utilisée uniquement dans une macro interne peut être vue comme une variable propre à cette macro. Ceci permettrait d'introduire la notion de variables locales présente dans la plupart des langages de programmation.

5 Conclusion

Pour les novices, la programmation des boucles reste difficile même après plusieurs semaines de formation (Teague and Lister, 2014). Dans cet article, nous avons proposé une méthode permettant de faciliter la programmation de boucles imbriquées. La boucle interne est remplacée par un appel d'une macro vide. L'originalité de la méthode réside dans le fait qu'à l'exécution de la boucle, lors de l'appel de la macro vide, le contrôle du programme est rendu au programmeur. Par manipulation directe des données, il peut effectuer les actions sur les données du programme pour satisfaire l'objectif de la macro. Puis il rend le contrôle au programme. L'exécution reprend et s'arrête à nouveau dans la boucle. Le programmeur manipule à nouveau les données et peut progressivement identifier l'algorithme de la macro qu'il simule. Il peut donc tester la boucle principale puis s'intéresser à la construction de la macro interne. Dans l'approche par "*stubs*" proposée dans (Caspersen and Kolling, 2009), la macro interne quand elle est vide produit toujours un résultat identique. Avec notre approche, le programmeur manipule les données pour produire le résultat attendu, qui peut être différent à chaque tour de boucle.

Ce dispositif a été mis en place dans le logiciel de manipulation directe AlgoTouch. A l'heure actuelle, pour des raisons liées à la pandémie, il a été seulement utilisé par les auteurs sur plusieurs exemples. Nous envisageons des expérimentations avec des enseignants et des apprenants. Pour les enseignants, ce système permet de montrer concrètement comment construire progressivement un programme contenant une boucle imbriquée. Pour un apprenant, le système doit permettre de simplifier la construction et le test de tels programmes.

Références

Adam, M., Daoud, M., and Frison, P. (2018). Teaching and learning how to program without writing code. In *International Conference Europe Middle East & North Africa Information Systems and Technologies to Support Learning*, pages 106–117. Springer.

Adam, M., Daoud, M., and Frison, P. (2019). Direct manipulation versus text-based programming : An experiment report. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 353–359.

Baron, G.-L. and Drot-Delange, B. (2016). L'éducation à l'informatique à l'école primaire. *1024 : Bulletin de la Société Informatique de France*, 8 :73–79.

Caspersen, M. E. and Kolling, M. (2009). Stream : A first programming process. *ACM Trans. Comput. Educ.*, 9(1).

Cetin, I. et al. (2020). Teaching loops concept through visualization construction. *Informatics in Education-An International Journal*, 19(4) :589–609.

Cypher, A. and Halbert, D. C. (1993). *Watch what I do : programming by demonstration*. MIT press.

Daly, T. (2013). *Influence of alice 3 : reducing the hurdles to success in a cs1 programming course*. University of North Texas.

Drot-Delange, B. (2013). Enseigner l'informatique débranchée : analyse didactique d'activités. In *AREF*, pages 1–13.

Frison, P. (2015). A teaching assistant for algorithm construction. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pages 9–14, New York, NY, USA. ACM.

Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3) :259–290.

Izu, C., Weerasinghe, A., and Pope, C. (2016). A study of code design skills in novice programmers using the solo taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 251–259.

Kordaki, M., Miatidis, M., and Kapsampelis, G. (2008). A computer environment for beginners' learning of sorting algorithms : Design and pilot evaluation. *Computers & Education*, 51(2) :708–723.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4) :1–15.

Meyer, B. (2009). *Touch of Class : learning to program well with objects and contracts*. Springer.

Mladenović, M., Boljat, I., and Žanko, Ž. (2018). Comparing loops misconceptions in block-based and text-based programming languages at the k-12 level. *Education and Information Technologies*, 23(4) :1483–1500.

Papavlasopoulou, S., Giannakos, M. N., and Jaccheri, L. (2017). Reviewing the affordances of tangible programming languages : Implications for design and practice. In *2017 IEEE Global Engineering Education Conference (EDUCON)*, pages 1811–1816.

Sorva, J., Karavirta, V., and Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Educ.*, 13(4).

Teague, D. and Lister, R. (2014). Programming : reading, writing and reversing. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 285–290.

Wing, J. (2011). Research notebook : Computational thinking—what and why ? the link magazine, spring. *Carnegie Mellon University, Pittsburgh. Retrieved*, 1 :2019.