Vérification de l'algorithme de calcul des ordres d'appel dans Parcoursup

Pierre Castéran[1], Hugo Gimbert*[1], Claire Mathieu[2], and Gérald Point[1]

[1]LaBRI, CNRS, Université de Bordeaux, France
[2]IRIF, CNRS, France

Parcoursup est la plateforme nationale de préinscription en première année de l'enseignement supérieur en France. Chaque année, plus de 1 000 000 candidats toutes procédures confondues formulent des vœux sur Parcoursup, où plus de 19 500 formations de l'enseignement supérieur sont proposées.

La procédure principale de Parcoursup se décompose en plusieurs phases. Initialement, les candidats font des voeux pour les différentes formations proposées. Ensuite chaque formation classe une partie des candidats ayant effectué des voeux dans cette formation [1]. Enfin les candidats reçoivent des propositions.

La loi ORE[2] prévoit l'application de taux encadrant la proportion de candidats boursiers et de candidats hors-secteur dans les formations concernées [3].

La mise en oeuvre des taux minimum de candidats boursiers et des taux minimum de candidats résidents dans l'académie est traitée en partie de manière automatisée, à l'échelle nationale, par un algorithme mis en oeuvre par le Service à Compétence Nationale Parcoursup (SCN Parcoursup). Le contexte de cette automatisation et le fonctionnement de cet algorithme sont détaillés dans le "Document de présentation des algorithmes de Parcoursup" [9, Section 4] et les algorithmes et leur implémentation open-source sont consultables sur le dépôt de code dédié [10].

L'algorithme calcule un *ordre d'appel* des candidats, qui définit l'ordre dans lequel les propositions seront faites aux candidats. Si les taux minimum de boursiers et de résidents étaient de 0%, l'ordre d'appel serait tout simplement l'ordre du classement pédagogique des candidatures, tel que transmis par la formation. Dès lors qu'il convient de respecter ces taux, l'ordre d'appel est obtenu à partir du classement pédagogique en faisant remonter des boursiers et/ou des résidents dans le classement afin de garantir le respect des dits taux.

*Les auteurs de ces travaux ont pu bénéficier de plusieurs subventions du MESRI sur la période 2018-2021.

[1]Plus précisément chaque formation dans laquelle le nombre de candidature est supérieur au nombre de places disponibles.

[2]Loi du 8 mars 2018 relative à l'Orientation et à la Réussite des Étudiants

[3]cf sections V et VI de l'article L. 612-3 du code de l'éducation.

Dans ce document on présente une partie des travaux[4] de vérification logicielle effectués sur cet algorithme:

- spécification des algorithmes,
- preuve mathématique de la conformité des algorithmes à leur spécification,
- preuve formelle en `Why3` de la conformité d'une implémentation à la spécification.

Ce travail de preuve a débuté avec l'équipe Toccata du LRI [5], en collaboration avec le second auteur de cet article et avec un soutien financier du Ministère de l'Enseignement Supérieur, de la Recherche et de l'Innovation (MESRI). Les travaux effectués par le LRI ont permis de réaliser la preuve de certaines propriétés d'une implémentation `WhyML` de l'algorithme à un taux, ainsi qu'une preuve partielle d'une implémentation alternative en `Java` [1, 2].

Nous avons complété les travaux du LRI en deux temps. Tout d'abord nous nous sommes intéressés aux parties de la spécification qui n'avaient pas été prouvées. Lors de cette étape nous avons été confrontés à un lemme, présenté à la section 3, qui nous semblait difficile à prouver en `WhyML`; nous avons alors délégué cette preuve à `COQ` (par la suite, le lemme a été prouvé directement en `WhyML`).

Dans un second temps nous avons réalisé les preuves pour l'algorithme à deux taux (celui en production). Ne pouvant prouver directement le code `Java`, nous nous sommes attachés à traduire l'algorithme pratiquement ligne à ligne ainsi qu'à réduire au maximum les annotations `WhyML`. L'outil `COQ` n'a pas été utilisé lors de cette étape.

Développant les preuves dans un cadre d'intégration continue sur une forge logicielle, nous nous sommes astreints à obtenir une preuve entièrement automatique en limitant `Why3` à l'utilisation d'`Alt-Ergo`. Nous avons utilisé `Why3` de manière interactive mais sans faire appel à ses stratégies[6], en ajoutant progressivement des assertions et lemmes `WhyML`, jusqu'à obtenir une validation automatique par `Alt-Ergo` des obligations de preuves générées par par `Why3`.

D'un point de vue quantitatif le projet représente un peu plus de 8000 lignes de `WhyML`. L'algorithme annoté représente uniquement, environ, 400 lignes de code. La preuve du lemme de la section 3 a demandé environ 1300 lignes de `COQ` (sans le contexte produit par `Why3`).

Travaux connexes. Les méthodes formelles sont également appliquées à d'autres algorithmes de décision publiques, notamment la sécurisation du calcul des impôts et des allocations sociales grâce au langage CATALA [8] ou les systèmes de régulation des heures de conduite des chauffeurs routiers [6]. Plus globalement, une

[4] Le détail de ces travaux est consultable dans le rapport technique [7] et disponible en libre accès dans le dépôt du projet[12].

[5] https://toccata.gitlabpages.inria.fr/toccata/

[6] Les stratégies de `Why3` ressemblent aux tactiques de `COQ`.

Q1) Pour tout $k \in 1 \dots |C|$, $d_1, \dots, d_k$ contient au moins $\lceil q_B * k \rceil$ candidats boursiers ; ou sinon, aucun candidat parmi $d_{k+1} \dots d_{|C|}$ n'est boursier.

Q3) Un candidat résident boursier qui a le rang r dans le classement pédagogique n'est jamais doublé par personne et a un rang inférieur ou égal à r dans l'ordre d'appel.

P4) Supposons $q_R = 0$. Comparé au classement pédagogique, l'ordre d'appel minimise le nombre d'inversions (distance de Kendall-tau), parmi ceux qui garantissent la propriété Q1.

Figure 1: Une partie de la spécification du calcul de l'ordre d'appel.

réflexion internationale appelée *Rules as Code* interroge les conditions de mise en oeuvre et d'élaboration des lois dont l'application est partiellement ou totalement automatisée [11].

1 Spécification et calcul des ordres d'appel

Le document de présentation des algorithmes de Parcoursup [9, Section 4] précise un certain nombre de propriétés que l'algorithme est tenu de respecter. Pour des raisons de lisibilité, la spécification disponible dans [9, Section 4] est exprimée de manière semi-formelle.

L'ordre d'appel est représenté par une permutation $d_1, \dots, d_{|C|}$ de l'ensemble C des candidats apparaissant dans le classement pédagogique. Certaines des propriétés de la spécification sont présentées dans la Figure 1. La propriété Q1 garantit le respect du taux boursier. La propriété P4 spécifie dans quelles proportions un candidat peut perdre des places lors du calcul de l'ordre d'appel.

Dans une formation soumise à deux taux q_B (boursiers) et q_R (résidents), l'algorithme consiste à énumérer les candidats à partir du candidat mieux classé, en les intégrant un à un dans l'ordre d'appel. Ce faisant, on surveille le respect des taux. L'algorithme est présenté figure 2, dans le cas particulier où le taux minimum de résidents q_R est fixé à 0%. On note q_B le taux minimum de boursiers, avec $0 \leq q_B \leq 1$.

Illustrons cet algorithme avec un exemple. Dans cette formation, le taux minimum de résidents est fixé à $q_R = 0$, le taux minimum de boursiers est de $q_B = 20\%$, et la formation a classé ses candidats dans l'ordre

$$(C_1, C_2, B_3, C_4, C_5, C_6, C_7, C_8, C_9, B_{10}, C_{11})$$

où seuls deux candidats B_3 et B_{10} sont boursiers. Alors l'ordre d'appel sera $(B_3, C_1, C_2, C_4, C_5, B_{10}, C_6, C_7, C_8, C_9, C_{11})$. En effet, d'une part le taux résident n'est jamais contraignant. D'autre part le taux boursier est contraignant lors du choix des candidats d'indices 1 et 6, ce qui permet à B_3 et B_{10} de "remonter" dans

1. Notons n le nombre de candidats apparaissant dans le classement pédagogique de la formation.
2. Pour chaque entier k de 1 à n, dans cet ordre, le candidat C_k de rang k dans l'ordre d'appel est calculé de la manière suivante. On a déjà sélectionné les candidats $C_1, \ldots, C_{k-1}$ dans l'ordre d'appel, et parmi eux il y a b boursiers. On dit que le taux minimum boursiers est contraignant si $b/k < q_B$. On considère tous les autres candidats, pris dans l'ordre pédagogique. Pour choisir C_k, parmi ceux-là:
   - Si le taux n'est pas contraignant, on prend le premier candidat.
   - Si le taux est contraignant, on prend le premier candidat boursier s'il y en a, le premier candidat sinon.

Figure 2: Algorithme de calcul de l'ordre d'appel dans les formations soumises au seul taux boursier.

l'ordre d'appel. Remarquons que le choix du candidat d'indice 11 est également contraignant, mais à ce stade du calcul de l'ordre d'appel tous les boursiers ont déjà été sélectionnés.

2 Vérification de l'algorithme

Les travaux de vérification logicielle effectués sur cet algorithme (voir [7, 12]) comprennent la spécification des algorithmes, la preuve mathématique de la conformité des algorithmes à leur spécification, la preuve formelle de la conformité d'une implémentation à la spécification, la vérification à l'exécution de la spécification.

La preuve formelle s'applique à une implémentation de l'algorithme dans le langage `WhyML`, en s'appuyant sur la plateforme `Why3` pour la preuve de programme [3, 4]. `Why3` est une plateforme pour la preuve de programme, qui s'appuie sur des prouveurs externes automatiques ou interactifs [3]. Elle permet de prouver des programmes écrits en `WhyML`, un langage de spécification et de programmation. Une fois prouvé, le programme peut être automatiquement traduit en un programme Caml correct par construction.

L'implémentation en `WhyML` de l'algorithme n'est pas l'implémentation `Java` effectivement exécutée par le SCN Parcoursup. Cette mesure garantit donc l'absence de bugs dans l'implémentation `WhyML` mais pas dans l'implémentation `Java`. Le langage `WhyML` se prête lui même peu à une exploitation par le SCN Parcoursup. Actuellement c'est l'implémentation `Java` qui est exploitée, avec des mesures de vérification à l'exécution, dont une vérification en double-aveugle contre une autre implémentation en PL/SQL.

Des mesures de *vérification à l'exécution* intégrées dans l'implémentation `Java`

va permettent de certifier que chaque exécution du logiciel est conforme à sa spécification, tant qu'elle ne lève pas une exception. Cette mesure est complémentaire au travail de preuve formelle effectuée sur l'implémentation WhyML. A première vue, le niveau de confiance donné par les mesures de *vérification à l'exécution* est moindre que celui d'une preuve formelle, car cette mesure de vérification à l'exécution écarte la possibilité d'une erreur dans les exécutions passées du programme, mais ne garantit rien sur les exécutions futures avec de nouvelles données d'entrée. Toutefois, la vérification à l'exécution s'applique à l'implémentation Java effectivement exploitée par le SCN Parcoursup, et elle a pu être mise en place rapidement dès la première année, très en avance de phase par rapport au développement des preuves formelles.

3 Preuve formelle de l'implémentation WhyML

Dans cette section on donne un aperçu de l'implémentation WhyML et des preuves formelles associées, qui sont disponibles sur le dépôt [12].

L'implémentation de l'algorithme à deux taux en WhyML est une traduction assez proche de l'implémentation Java, consultable dans le module algo.mlw.

L'essentiel de la preuve de l'algorithme consiste à prouver que les invariants et propriétés de l'algorithme restent satisfaits à chaque itération. La preuve est décomposée en plusieurs fichiers indépendants, correspondants aux différents invariants. Cette modularité permet de conserver une définition de l'algorithme la plus lisible possible (cf. algo.mlw). De plus cette isolation des différentes parties de la preuve améliore également la performance des solveurs automatiques en isolant les différents contextes dans lesquels les différents invariants sont prouvés.

Traitement modulaire des invariants. Nous avons essayé autant que possible de séparer en plusieurs modules et fichiers l'implémentation WhyML proprement dite, la définition des différentes propriétés de la spécification, la définition des différentes invariants, et les lemmes permettant de prouver les invariants. Prenons pour exemple la fonction selectionnerMeilleurVoeu[7] dont le contrat spécifie que (voir figure 1):

- si avant l'appel à cette fonction les invariants étaient vérifiés par g, oak (c-à-d. l'ordre courant) et filesAttente (lignes 75-76);
- alors (lignes 77-78) le candidat choisi, result, est tel que les invariants restent satisfaits pour g, oak augmenté de result et filesAttente de laquelle result a été retiré. Pour les variables modifiables (c-à-d. annotées mutable en WhyML), la post-condition fait référence à la valeur des variables à la fin de la fonction.

[7] Le code de la fonction selectionnerMeilleurVoeu est donné dans le module algo_selection_meilleur_voeu.mlw

```
69    let selectionnerMeilleurVoeu
70        (contrainteTauxBoursier : bool)
71        (contrainteTauxResident : bool)
72        (g: groupe_classement)
73        (oak : Q.t voeu)
74        (filesAttente :  enummap) : voeu
75      requires {[@expl:InvAlgo2]
76            invariants_algo2 g oak filesAttente }
77      ensures {[@expl:InvAlgo2(oak.res)]
78           invariants_algo2 g (snoc oak result) filesAttente }
```

Figure 1: Contrat de la fonction de sélection du meilleur candidat.

La preuve de la post-condition du contrat par `Why3` nécessite l'utilisation du lemme `invariants_algo_snoc` déclaré dans le module algo_invariants.mlw. Ce lemme est un simple corollaire des lemmes qui prouvent que chaque propriété ou invariant est satisfait après l'appel à `selectionnerMeilleurVoeu`. Ces lemmes sont prouvés dans des modules séparés portant le nom de la propriété ou de l'invariant concerné; par exemple q1.mlw ou iq4_3.mlw.

Traduction de types `Java`. L'implémentation `Java` commence par le partitionnement du classement pédagogique en quatre files (`java.util.Queue`) de candidats stockées dans une table (`java.util.EnumMap`) indexée par les quatre types de candidats:

```
EnumMap<VoeuClasse.TypeCandidat, Queue<VoeuClasse>> filesAttente
```

Nous avons modélisé cette structure de donnée avec un type `WhyML` *ad hoc* contenant explicitement les quatre files. Cette structure est déclarée dans le module enummap.mlw de la preuve.

```
type enummap =
{
  bds : Q.t voeu;
  bhs : Q.t voeu;
  nbds : Q.t voeu;
  nbhs : Q.t voeu;
}
```

Par exemple la file stockée dans le champ `bds` contient les boursiers du secteur, l'équivalent de la file `Java`

```
filesAttente[BOURSIER_DU_SECTEUR]
```

Nous avons adapté des méthodes de la classe `java.util.EnumMap` à notre structure comme par exemple les méthodes `get` et `put`.

Preuve d'un lemme clé. La propriété (P4) de l'algorithme à un taux s'appuie sur un lemme clé concernant le nombre d'inversions dans un vecteur.

Le *nombre d'inversions* ou indice de Kendall-Tau est défini pour toute permutation $(e_k)_{k\in 1\ldots n}$ de $\{1,\ldots,n\}$ par

$$\begin{aligned}\text{Inversions}((e_k)_{k\in 1\ldots n}) =\mid \{&i \in 1\ldots n, j \in 1\ldots n,\\ &(i<j \wedge \text{rang}(e_i) > \text{rang}(e_j)) \vee (i>j \wedge \text{rang}(e_i) < \text{rang}(e_j))\} \mid\end{aligned}$$

La preuve de (P4) passe par le lemme suivant.

Lemma 1. *Soit $(e_k)_{k\in 1\ldots n}$ une permutation de $1\ldots n$ et $0 \leq i_0 < j_0 \leq n$ une inversion de ce vecteur. Soit e' le vecteur obtenu en échangeant e_{i_0} et e_{j_0}. Alors le nombre d'inversions de e' est inférieur à celui de e.*

La preuve de ce lemme a été réalisée en COQ et en Why3. En Why3, le lemme s'énonce comme suit

```
let lemma  inversions_dec (s : seq voeu) (p : pair)
  requires { 0 <= p.i && 0 <= p.j }
  requires { inversion s p }
  ensures { nb_inversions (swap s p) < nb_inversions s }
= ()
```

COQ est un système de gestion des preuves formelles [5]. Il fournit un langage formel permettant d'écrire des définitions mathématiques, des algorithmes exécutables et des théorèmes, et fournit également un environnement de preuves semi-interactive vérifiées par ordinateur. Les applications typiques sont la certification des propriétés des langages de programmation (e.g. le projet de certification du compilateur CompCert, la "Verified Software Toolchain for verification of C programs", ou le framework Iris pour la logique de séparation concurrente), la formalisation des mathématiques (e.g. la formalisation complète du théorème de Feit-Thompson theorem), et l'enseignement.

L'utilisation des deux approches basées sur COQ d'une part et Why3 d'autre part a permis de les comparer. L'utilisation de COQ est adaptée à la preuve des propriétés abstraites, en s'exploitant sur les riches librairies du langage. Les modules Why3 générant les obligations de preuve COQ doivent être réduits à l'essentiel, sous peine de produire de nombreux axiomes inutiles pour le théorème visé. Une preuve COQ fournit un très haut niveau de confiance, sans dépendance à des solveurs externes. La preuve COQ utilise un angle différent de la preuve en Why3. L'utilisation de Why3 offre un bon degré d'automatisation du traitement des cas et sous-cas. Dans les deux approches, le traitement des sommes partielles sur des sous-ensembles de $\mathbb{Z}$ nécessite de forger quelques outils *ad-hoc*.

Conclusion

Un autre algorithme crucial pour le fonctionnement de Parcoursup est l'algorithme qui calcule quotidiennement quelles propositions doivent être envoyées aux candidats. Tout comme le calcul des ordres d'appel, cet algorithme est documenté dans le "Document de présentation des algorithmes de Parcoursup" [9, Sections 3,5,6] et

l'implémentation `Java` est publiée sur le dépôt public du code de Parcoursup [10]. Des mesures de vérification à l'exécution ont été implémentées en `Java` pour garantir la sécurité du code de l'algorithme des propositions mais cet algorithme n'a pas encore été prouvé formellement, notamment l'unicité du résultat pour les formations avec internat, une propriété importante qui implique l'indépendance du résultat à l'ordre d'énumération des données d'entrée.

La preuve formelle de l'implémentation `Java` a été partiellement effectuée [2] mais pas encore finalisée. Pour avancer dans cette direction, il y a au moins deux possibilités. Premièrement contourner les difficultés inhérentes à l'analyse du code `Java` en générant des implémentations de référence en `Java` à partir des implémentations `WhyML`, en commençant par le code effectuant la vérification à l'exécution. C'est un travail en cours, dans le cadre d'une collaboration entre le LRI et le LaBRI. Deuxièmement définir un fragment syntaxique de `Java` qui se prête à une interprétation en `WhyML` avec une extension de l'outil `jmltowhy3` développé au LRI [2].

References

[1] Léo Andrès. Vérification par preuve formelle de propriétés fonctionnelles d'algorithme de classification. Research report, Université Paris Sud (Paris 11) - Université Paris Saclay, https://hal.inria.fr/hal-02421484/file/internship-report-parcoursup.pdf, August 2019.

[2] Benedikt Becker, Jean-Christophe Filliâtre, and Claude Marché. Rapport d'avancement sur la vérification formelle des algorithmes de ParcourSup. Technical report, Université Paris-Saclay, https://hal.inria.fr/hal-02447409/file/main.pdf, January 2020.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3. http://why3.lri.fr.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's Verify This with Why3. *International Journal on Software Tools for Technology Transfer*, 17(6):709–727, 2015.

[5] COQ. Site web de COQ. https://coq.inria.fr.

[6] formalvindications. Site web de formalvindications. https://formalvindications.com.

[7] Hugo Gimbert, Pierre Castéran, Claire Mathieu, and Gérald Point. Vérification de l'algorithme de calcul des ordres d'appel dans Parcoursup. Research report, CNRS ; Université de Bordeaux ; LaBRI - Laboratoire Bordelais de Recherche en Informatique ; IRIF, October 2021.

[8] Liane Huttner and Denis Merigoux. Traduire la loi en code grâce au langage de programmation Catala. In *Intelligence artificielle et finances publiques*, Nice, France, October 2020.

[9] MESRI. Document de présentation des algorithmes de parcoursup. https://framagit.org/parcoursup/algorithmes-de-parcoursup/-/blob/master/doc/presentation_algorithmes_parcoursup_2021.pdf, 2021.

[10] MESRI. Dépôt public du code de parcoursup. https://framagit.org/parcoursup/algorithmes-de-parcoursup/, 2021.

[11] OCDE. Rapport RaC de l'OCDE. https://oecd-opsi.org/projects/rulesascode/.

[12] Hugo Gimbert, Pierre Castéran, Claire Mathieu, Gérald Point. Dépôt des preuves des algoritihmes de Parcoursup Why3 et COQ. https://gitub.u-bordeaux.fr/parcoursup/why3.