HAL
open science

Rapport d'avancement sur la vérification formelle des algorithmes de ParcourSup

Benedikt Becker, Jean-Christophe Filliâtre, Claude Marché

To cite this version:

Benedikt Becker, Jean-Christophe Filliâtre, Claude Marché. Rapport d'avancement sur la vérification formelle des algorithmes de ParcourSup. [Rapport Technique] Université Paris-Saclay. 2020. hal-02447409

HAL Id: hal-02447409

https://inria.hal.science/hal-02447409v1

Submitted on 21 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Rapport d'avancement sur la vérification formelle des algorithmes de ParcourSup

Benedikt Becker Jean-Christophe Filliâtre
Claude Marché

Université Paris-Saclay
Laboratoire de Recherche en Informatique
UMR 8623 du CNRS et de l'Université Paris-Sud
Inria Saclay-Île-de-France

21 janvier 2020

Table des matières

1 Introduction, résumé des résultats obtenus **2**
1.1 Généralités sur l'approche par preuve formelle . . . 2
1.2 Résumé de la démarche et des travaux effectués . . . 3
1.3 Conclusions . . . 3
1.4 Perspectives . . . 4

2 Détails des travaux réalisés **6**
2.1 Preuve avec Why3 d'un algorithme simplifié à un seul taux . . . 6
2.2 Analysis of ParcourSup with two constraints using OpenJML . . . 7
2.2.1 Original Java implementation . . . 7
2.2.2 Adapted implementation . . . 7
2.2.3 OpenJML . . . 7
2.2.4 Array-based implementation . . . 8
2.2.5 Verification results . . . 8
2.2.6 Limitations of OpenJML for ParcourSup . . . 9
2.3 Jml_2Why^3: a prototype tool to convert Java/JML to Why3 . . . 9
2.3.1 Jml_2Why^3's conversion of Java/JML to Why3 . . . 10
2.3.2 Summary of limitations . . . 14
2.4 Verification of ParcourSup with two constraints using Jml_2Why^3 . . . 14
2.4.1 Arithmetic and null-pointer safety . . . 14
2.4.2 Permutation property . . . 14
2.5 Excursion to ownership and proofs in Rust . . . 16

3 Annexes **19**
3.1 Jml_2Why^3conversions . . . 19
3.1.1 Identifiers . . . 19
3.1.2 Types . . . 20
3.1.3 Expressions and terms . . . 20
3.1.4 Expressions . . . 21
3.1.5 Terms . . . 22
3.1.6 Statements . . . 23
3.1.7 Statement . . . 23
3.1.8 Exception declaration . . . 23
3.2 ParcourSup implementations . . . 24
3.2.1 Original implementation . . . 24
3.2.2 Adapted implementation . . . 27
3.2.3 Arrays-based implementation for OpenJML . . . 30
3.2.4 Permutation implementation . . . 33
3.2.5 Rust implementation . . . 38

Chapitre 1

Introduction, résumé des résultats obtenus

ParcourSup [14] est le système informatique national français utilisé pour l'orientation des nouveaux bacheliers dans les établissements d'enseignement supérieur. Ce système utilise des algorithmes spécifiques pour établir des classements des candidats pour chaque établissement auxquels ils postulent. Ces classements sont produits en fonction des vœux des candidats, du classement de leurs dossiers par les établissements, ceci en tenant compte de contraintes générales sur des taux de boursiers et des taux de non-résidents.

Les propriétés attendues des algorithmes en question sont documentées par un document en français [13]. Ces algorithmes sont implantés en langage Java, et leur code source [12] est disponible sous une licence libre et ouverte.

1.1 Généralités sur l'approche par preuve formelle

Entre mars et décembre 2019, nous nous sommes intéressés à étudier la possibilité de *prouver formellement* que le code Java respecte les spécifications.

Notre équipe de recherche [1] est spécialisée dans les techniques de preuves formelles de programmes, et en particulier développe et distribue l'environnement Why3 [5]. Cet environnement est conçu de manière générique, ou modulaire, de façon à tirer partie, en aval, d'un maximum d'outils de preuve automatique du monde entier, et, en amont, de servir de langage intermédiaire pour prouver des codes sources écrits en langage C, Java, Ada, via des environnements tiers comme Frama-C, Krakatoa et Ada/SPARK [16].

Le choix de l'environnement de preuve pour prouver les algorithmes de ParcourSup pourrait naturellement se tourner vers notre *front-end* Krakatoa [19, 18] de Why3, dédié à Java, mais celui-ci est assez ancien : il ne supporte que la version 1.4 de Java (le code Java de ParcoursSup nécessite la version 8), et n'est pas vraiment maintenu faute de suffisamment d'utilisateurs. D'autres environnements de preuve similaires existent dans le monde, comme KeY [2] et VeriFast [15]. Sauf erreur de notre part, ces outils ne supportent pas encore la version 8 de Java. Le candidat le plus prometteur est OpenJML [7], qui lui supporte la version 8 de Java.

Par ailleurs, notre expérience en preuve de programmes nous conduit à penser que pour prouver le code de ParcourSup, il est préférable de commencer à considérer une abstraction du code Java, écrite directement dans le langage intermédiaire de Why3, afin de se focaliser dès le départ sur l'écriture des spécifications formelles nécessaires, puis de la recherche des invariants à insérer dans le code.

1. au laboratoire LRI, sous la tutelle commune de l'Université Paris-Sud, du CNRS et de l'Inria

1.2 Résumé de la démarche et des travaux effectués

Notre démarche a consisté à poursuivre plusieurs pistes qui sont résumées ci-après. Chaque piste est ensuite décrite plus en détails dans les sections suivantes, plus techniques, de ce document. Pour des raisons pratiques, certaines de ces sections sont rédigées en anglais.

1. Dans un premier temps, nous avons écrit en Why3 une version simplifiée du code de ParcourSup, à un seul taux, c'est-à-dire en considérant uniquement le critère des boursiers mais pas le critère des non-résidents. Ce travail a fait l'objet d'un stage de Magistère (de Léo Andrès, mai-juillet 2019). Un large sous-ensemble des propriétés demandées dans la spécification ont été prouvées. Ce travail fait l'objet d'un rapport de stage spécifique [3]. Les résultats sont résumés dans la section 2.1 ci-après.
2. Dans un second temps, nous avons tenté d'analyser le code Java tel quel, c'est-à-dire sans spécifications formelles, avec OpenJML. Même sans spécifications formelles, il y a des obligations de preuve à prouver car il faut démontrer que le code ne peut pas faire *d'erreurs à l'exécution* : divisions par zéro, débordement d'opérations arithmétiques sur les entiers signés 32 bits de Java, débordement des bornes des tableaux, déréférenciation de pointeur nul. Cette analyse est détaillée en section 2.2. Nous présentons les résultats obtenus et les limitations identifiées.
3. Afin de contourner des contraintes et des limitations d'OpenJML, nous avons alors choisi de concevoir et d'implémenter une nouvelle chaîne d'outils. Afin de bénéficier du support à jour de Java 8 fourni par OpenJML, nous n'avons pas repris l'ancien Krakatoa mais nous avons :
   — conçu un patch à OpenJML pour lui demander de restituer le résultat de son analyse syntaxique et de son typage dans une structure de données enregistrée dans un fichier au format JSON ;
   — défini un nouveau logiciel Jml_2Why^3 qui lit le fichier JSON en question et traduit le code Java, ainsi que les spécification formelles, en un code Why3 équivalent ;
   — enfin nous réutilisons l'API de Why3 pour générer les obligations de preuve et interroger les prouveurs externes, ainsi que l'interface graphique de Why3 pour compléter les preuves de manière interactive quand c'est nécessaire.

   Le prototype Jml_2Why^3 est décrit en section 2.3. Les expérimentations avec cet outil sur le code de ParcourSup sont décrites dans la même section. Les résultats et les limitations de cette approche sont ensuite présentés.
4. Concernant les limitations de l'approche par Jml_2Why^3, sur les questions *d'ownership* et *d'emprunt*, une petite expérience a été conduite pour évaluer l'intérêt que pourrait apporter le langage de programmation Rust à la place de Java. Ceci est détaillé en section 2.5.

1.3 Conclusions

Les conclusions obtenues sont résumées par les points suivants :

— L'analyse avec Why3 d'un algorithme à un taux a permis de valider plusieurs des propriétés énoncées dans le document de spécification [13, page 7], à savoir, que l'ordre d'appel est bien une permutation du classement pédagogique, que le taux de boursiers est bien respecté (P1), qu'un candidat boursier n'est jamais doublé par personne (P2), qu'un candidat non boursier ne double jamais personne (P3), et que l'ordre d'appel est le minimum selon l'ordre lexicographique induit par les classements (P5).

— L'analyse du code Java (à deux taux, disponible en ligne [12]) avec OpenJML a identifié des risques d'erreurs à l'exécution dues à des dépassement de capacité dans les calculs arithmétiques. Nous avons déterminé que les erreurs ne sont plus possibles si on assure que le nombre de candidats est inférieur à 21.474.636. Plus de 20 millions est une limite suffisamment large en pratique, donc il n'y a pas de risque urgent à corriger. On note néanmoins que cette limite n'était pas identifiée auparavant.

L'analyse avec OpenJML s'est également montrée délicate sur une occurrence particulière d'une déréférenciation de pointeur (il faut montrer que celui-ci n'est pas le pointeur `null` à ce point de programme). La preuve de non-nullité a pu être obtenue formellement, avec un nombre significatif d'indications intermédiaires, qui n'ont néanmoins pas été une surprise car le code était commenté à ce sujet, la difficulté était connue. Notons que, afin de réussir cette analyse, nous avons dû modifier légèrement le code Java. Le code initial était correct, mais au-delà des capacités d'OpenJML.
La preuve du comportement fonctionnel du code Java (propriétés décrites dans le document de spécification en francais) n'a pas pu être faite à cause de limitations d'OpenJML décrites en section 2.2.

— Ensuite l'utilisation de notre prototype Jml_2Why^3 nous a permis d'aller plus loin dans la preuve de propriétés fonctionnelles du code Java. Non seulement nous avons pu obtenir de nouveau les preuves d'absence de débordement arithmétique et d'absence de déréférenciation de pointeur nul, mais nous avons pu établir formellement la propriété que l'ordre d'appel final est bien une permutation (donc une bijection) de la liste de vœux initiale : en particulier, aucun vœu n'est perdu en route par le code Java. Par manque de temps nous nous sommes arrêtés à la tentative de preuve de propriétés attendues de cette permutation, par exemple qu'un boursier non-résident n'est jamais doublé par personne.

1.4 Perspectives

Les travaux effectués, même si ceux-ci n'ont pas atteint l'objectif ultime d'une preuve formelle complète du code Java original, nous ont convaincu d'une manière très forte que, hormis le petit risque de débordement arithmétique dû à un nombre potentiellement trop élevé de candidats, le code Java de ParcourSup qui implémente les algorithmes d'interclassement s'exécutera toujours sans erreurs et que les propriétés informelles, mais mathématiquement très précises, du document de spécification [13] sont satisfaites.

À ce stade, il s'agit de s'interroger sur les leçons apprises par nos travaux et d'en déduire des recommandations s'il s'agit de continuer à faire des preuves formelles sur les logiciels utilisés par les organismes publics de la république française.

La recommandation principale concerne le choix du langage de programmation. Le choix du langage Java est sans doute justifié par des questions d'interopérabilité, par exemple avec les systèmes de gestion de bases de données. Ce choix s'avère par contre particulièrement inapproprié si l'on vise une preuve formelle du code implémentant les algorithmes : le langage Java est un langage qui autorise des mutations mémoires et du partage en mémoire de données mutables, traits de programmation qu'il est notoirement difficile de supporter par la preuve formelle. Même un langage plus récent comme Rust, qui apporte pourtant déjà beaucoup sur la sûreté des manipulations en place de la mémoire, reste encore de nos jours difficile à traiter par les approches et outils de l'état de l'art de la preuve formelle. Par ailleurs, le choix de JML comme langage de spécification apparaît comme peu satisfaisant. En effet, ce langage ne permet pas de définir facilement des concepts mathématiques avancés (une permutation, un ordre lexicographique, etc.). Pour cette raison, poursuivre le travail sur Jml_2Why^3 est peu prometteur.

Si l'on vise à obtenir des preuves formelles des implémentations des algorithmes, qui puissent être obtenues avec le minimum de moyens humains et qui soient relativement aisées de mettre à jour lorsque le code lui-même est mis à jour, il nous semble indispensable de choisir un langage de programmation, ou au minimum un style de programmation, qui :

1. au niveau du modèle d'exécution, imposerait des contraintes fortes sur les modifications en place de la mémoire ;
2. au niveau des spécifications, serait nativement équipé d'un langage formel pour ces spécifications.

On imagine plusieurs pistes possibles pour aller dans cette direction :

1. Utiliser un environnement dédié pour le développement de code formellement prouvé : Coq, Why3, F⋆, voire même l'atelier B qui est utilisé couramment dans le domaine du logiciel critique dans le ferroviaire. Ces environnements disposent typiquement d'un mécanisme *d'extraction* de code vers des langages comme OCaml, C ou Ada. C'est ainsi par exemple que le compilateur C prouvé CompCert [6] est développé : il est développé et prouvé en Coq, puis extrait vers OCaml ; ou bien le code critique de la ligne 14 du métro parisien, développé en B puis extrait vers Ada. Il faut remarquer par ailleurs que Why3 propose une extraction vers OCaml et C, et il serait d'un coût (humain) raisonnable de rajouter une extraction vers Java, comme ce qui a été fait pour le langage C [21].
2. Une piste similaire serait de coder en Ada/Spark, qui travaille directement sur du code Ada standard, car Ada/Spark est justement conçu pour limiter le langage Ada [1, 16] de manière à ce que les preuves soit raisonnablement faisables (dans le sens : avec un maximum d'automatisation). Ou bien, dans le même ordre d'idée d'utiliser un langage comme Dafny [17] qui est un langage qui se compile vers la JVM (mais qui par contre est peu contraignant sur les mutations mémoires). Par contre dans les deux cas les langages de spécification fournis ont des défauts similaires à ceux de JML (peu adapté à la définition de concepts très mathematiques).
3. On peut aussi imaginer d'ajouter à Why3 un « micro-front-end » pour Java, similaire aux front-ends micro-python et micro-C déjà existants. Il s'agirait alors de coder dans un fragment très restreint de Java, qui serait quasiment un fragment purement fonctionnel. Un autre manière de voir cette piste serait de proposer de développer le code en Why3 mais avec une syntaxe compatible avec Java. Pour le dire en une phrase, le langage micro-Java serait un Java où il serait interdit de mettre des objets mutables à l'intérieur d'un conteneur quelconque (y compris un tableau ou un autre objet). Par ailleurs, tous les objets seraient « non-nullable » (cf section 2.3) par défaut.
4. Concevoir et utiliser un langage spécifique au domaine, comme ce qui a été fait par exemple pour une implémentation du code des impôts [20], qui travaille sur le langage dédié M.

En conclusion, il nous semble que si l'on souhaite poursuivre un travail de vérification formelle du code de Parcoursup, il faut commencer par remettre en cause le choix de coder en Java d'une part et de vérifier le code a posteriori d'autre part. Il vaut mieux dès le départ écrire un code dans un langage ou un style adapté à la preuve formelle, et penser aux preuves qui devront être faites dès le choix des structures de données.

Chapitre 2

Détails des travaux réalisés

2.1 Preuve avec Why3 d'un algorithme simplifié à un seul taux

Dans le cadre d'un stage de magistère de M1 à l'Université Paris-Sud, Léo Andrès a travaillé sur la vérification d'un algorithme de Parcoursup avec l'outil Why3 entre le 1er mai et le 31 juillet 2019. Ce travail avait deux objectifs principaux : dégager les invariants nécessaires à une vérification déductive des algorithmes de Parcoursup ; étudier le degré d'automatisation d'une telle preuve. Le stage de Léo Andrès a été encadré par Jean-Christophe Filliâtre. Le travail de Léo Andrès a donné lieu a un rapport, en français, librement disponible en ligne [3].

Les quatre premières semaines du stage de Léo Andrès ont été consacrées à la maîtrise de l'outil Why3, une plate-forme de vérification déductive développée au LRI, notamment par Claude Marché et Jean-Christophe Filliâtre. Cet apprentissage s'est fait sur des programmes conceptuellement plus simples que les algorithmes de Parcoursup. Les huit semaines suivantes (juin–juillet) ont été consacrées à la vérification avec Why3 de l'algorithme le plus simple de Parcoursup, à savoir l'algorithme de calcul de l'ordre d'appel dans un groupe soumis au seul taux minimum boursiers [13, page 7].

La première tâche a consisté à implémenter l'algorithme dans le langage de l'outil Why3, qui s'apparente à un fragment du langage OCaml. Cette implémentation s'est en partie inspirée du code Java de Parcoursup [12], avec l'idée que les invariants puissent être repris plus tard pour une vérification du code Java (même si l'implémentation Java réalise un algorithme plus complexe, à deux taux). Une fois le code écrit, sa sûreté d'exécution a été prouvée : absence d'accès en dehors des bornes de tableaux, absence de tentative de retrait dans une file vide, terminaison des boucles, etc. Dans cette implémentation, l'arithmétique est de précision arbitraire. On n'a donc pas à vérifier l'absence de débordement arithmétique dans les calculs.

Dans un second temps, une preuve de correction fonctionnelle a été menée, visant à établir que le code vérifie bien les cinq propriétés énoncées dans le document public décrivant les algorithmes de Parcoursup pour l'algorithme à un seul taux [13, page 7, propriétés 1–5]. Une majorité de ces preuves ont pu être menées à terme pendant le stage. Plus précisément, il a été vérifié que l'ordre d'appel est bien une permutation du classement pédagogique, que le taux de boursiers est bien respecté (P1), qu'un candidat boursier n'est jamais doublé par personne (P2), qu'un candidat non boursier ne double jamais personne (P3), et que l'ordre d'appel est le minimum selon l'ordre lexicographique induit par les classements (P5). Seules la seconde partie de la propriété P3 (un candidat non-boursier aura au pire le rang $r \times (1 + q_b/(100 - q_b))$ et la propriété P4 (l'ordre d'appel est la permutation qui minimise le nombre d'inversions parmi celles qui vérifient P1) n'ont pu être vérifiées, par manque de temps. Les preuves se sont en effet avérées relativement difficiles, les démonstrateurs automatiques utilisés par l'outil Why3 étant souvent incapables de décharger les obligations de preuve. Il faut faire alors un travail de preuve interactive, long et fastidieux.

Pendant le stage, une implémentation de référence de l'algorithme à un taux a été obtenue par

traduction automatique du code Why3 vers le langage OCaml (un mécanisme fourni par l'outil Why3). Le code ainsi obtenu a pu être testé sur des données anonymisées de Parcoursup.

Une description plus détaillée du travail de Léo Andrès se trouve dans son rapport de stage [3].

2.2 Analysis of ParcourSup with two constraints using OpenJML

2.2.1 Original Java implementation

The main method of the Java implementation of ParcourSup is called `calculerOrdreAppel`. This method creates a permutation of a list of wishes (`voeuxClasses`), where each wish is characterised by its rank (`rang`) and its status as stipendiary and resident. The method returns a permutation of the input wishes considering their ranks and two constraints on the portions of wishes that represent stipendiaries and residents. The Java implementation of ParcourSup is available in a public git repository [12]. Version *Évolutions 2019* (Git commit `7be9a08`) of the code is shown in Appendix 3.2.1.

In the first part of method `calculerOrdreAppel`, the input wishes are ordered by rank and classified by their status as stipendiary and resident into one of four queues. As a result, the four queues contain wishes that are sorted by their rank and that are homogeneous according to their status (lines 45-72): stipendiary and resident, stipendiary and non-resident, non-stipendiary and resident, and non-stipendiary and non-resident. In the second part of the method, eligible candidates are selected in a loop as the heads of the queues, when their status respect the constraints on the portion of stipendiary and resident candidates (`eligibles`, lines 86-106). The best candidate is selected as the eligible candidate with the best rank (`meilleur`, lines 111-129). The best candidate is then removed from its queue, and appended to the result permutation.

2.2.2 Adapted implementation

In the original implementation, the four queues are stored in a hash map with four elements, mapping candidate statuses to the queues of wishes. This allows a concise selection of the right queue for insertion (lines 116-125) and removal (lines 150-152) of wishes. However, it hinders the static analysis or automated verification fundamentally due to a possible aliasing for the (mutable) queues in the hash map.

We adapted the Java implementation by representing the four homogeneous queues by four individual variables (`BR`, `BnR`, `nBR`, and `nBnR`, see Appendix 3.2.2). This requires a more explicit selection of queues for categorising the wishes (lines 63-78), for selecting the best candidate (lines 132-147), and for removing it (lines 152-163). It allows stating the homogeneity of the queues on the queues directly (lines 53-56 and 92-95), which is fundamental to proving the safety and other properties of the algorithm. The JML annotations in the adapted implementation are explained in Section 2.2.5.

2.2.3 OpenJML

OpenJML [7, 8] is a program verification tool for Java programs that allows for checking specifications of programs annotated in the Java Modelling Language (JML).

Installation of OpenJML A graphical user interface for OpenJML is available as a plugin for the Eclipse IDE. The installation instructions are available online [9]. After installing Eclipse using the standard installer *Oomph*, running the plugin as of version 0.8.41 may fail due to unresolved file paths. To use the plugin, Eclipse should be installed from a package in the `tar.gz` format available online [11].

2.2.4 Array-based implementation

The Java implementation of the ParcourSup algorithm uses standard containers such as `java.util.List` or `java.util.Queue`, which are supported by OpenJML. However, their use imposes a number of difficulties to the verification using OpenJML. First, the JML specification of the interfaces of the containers is not complete for Java 8, or the specification uses JML features that are not supported by OpenJML (e.g., the use of `\num_of` in the specification of `java.util.Arrays.sort()`). Second, we encountered difficulties to maintain class invariants concerning ownership over for-loops that modify multiple containers (see file `openjml/Test.java`). Third, the verification of programs using standard containers is more complex for OpenJML than equivalent programs using plain arrays. For these reasons, we replaced the containers in the adapted ParcourSup by arrays, without changing the program behaviour (shown in Appendix 3.2.3):

1. The input wishes and the resulting permutation of wishes are represented as an arrays of wishes.
2. The input wishes are required to be sorted by their rank (instead of sorting the input wishes in the method).
3. Each queue is represented by an array with start index and end index of valid values.

2.2.5 Verification results

Arithmetic safety All numbers in the original implementation (Appendix 3.2.1) are represented as standard Java integers, i.e. 32-bit signed integers which range from -2^{31} to $2^{31}-1$. We identified a possible arithmetic overflow in the (original, adapted, and array-based) ParcourSup implementation: To compute the rate constraint on stipendiaries (and residents), the number of assigned stipendiaries (and residents) is multiplied by 100 (lines 102-110). When the number of assigned wishes with a given status becomes larger than the maximum standard integer divided by hundred, the integer number exceets the valid integer range.

The arithmetic overflow can be prevented by limiting the number of input wishes `voeuClasse` to less than $(2^{31}-1)/100 = 21,474,836$, to ensure that the numbers in the calculation of the rate constraints stay in the valid range of 32-bit signed integers.

Null-pointer safety The verification of null-pointer safety in the ParcourSup algorithm is more complex: The second loop determines the resulting order of candidates, and the best candidate (`meilleur`) is defined as the best-ranking stipendiary that is not resident (lines 116-129 in the original implementation in Appendix 3.2.1). Verifying that there is actually a stipendiary left and that variable has a non-null value after the assignment requires comprehensive reasoning over the second loop in the implementation to deduce the following properties:

1. The candidates in the queues `BR`, `BnR`, `nBR`, and `nBnR` are homogeneous in their status as stipendiary and resident.
2. At least one wish is left in one of the queues.
3. The constraint on stipendiaries implies that either queue `BR` or queue `BnR` are non-empty.
4. The constraint on residents implies that either queue `BR` or queue `nBR` are non-empty.
5. After determining the best candidate, it is the head element of one of the four queues.

Properties 1. and 2. are established and maintained by invariants on both loops. Properties 3. and 4. are asserted after the definition of the constraints. Property 5. is asserted after determining the best candidate (line 149 in the original implementation).

The adapted and array-based implementations (Appendix 3.2.2 and Appendix 3.2.3) contain the JML annotations to ensure arithmetic safety and null-pointer safety. The annotations in the array-based implementation were verified using OpenJML. The verification was carried out using the command `java -jar openjml.jar -esc -progress GroupeClassement__everything_arrays.java`, and took one hour and 26

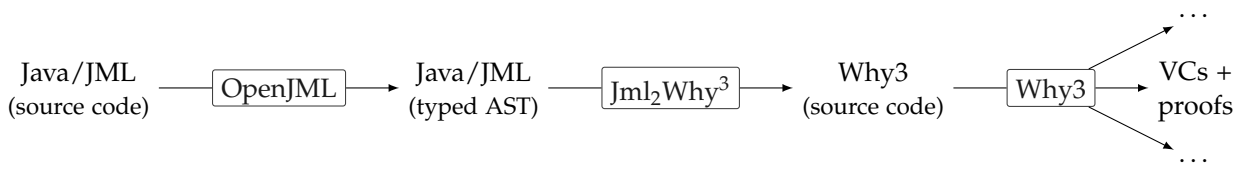

Figure 2.1 – Pipeline for the verification of Java/JML programs using Jml_2Why^3.

minutes on standard laptop hardware. Running OpenJML on the adapted implementation did not terminate within a day.

2.2.6 Limitations of OpenJML for ParcourSup

While proving the safety of the array-based implementation of ParcourSup we encountered a number of limitations that impede the use of OpenJML for verifying the functional properties of the ParcourSup implementation.

Places that are modified by loops in Java must be specified with JML annotations `\loop_modifies`. However, the correctness of these annotations are not checked by OpenJML, and their incompleteness may introduce inconsistencies, invalidating further proofs. By contrast, Why3 infers loop modifications generally, simplifying annotations and improving consistency.

The specifications of the standard API of Java uses JML features that are not supported by OpenJML, for example `\num_of`, preventing the verification of programs that use such API calls. The specification of other parts of the Java API are not well suited to discharging verification conditions to automated provers. By contrast, Why3's API, is designed with automatic verification in mind. For example, a queue is modelled by a sequences in Why3, which in turn is just characterised by a length and a function from indices to values.

Modifying multiple containers in a loop introduces verification conditions about loop invariants concerning the ownership of objects that are unclear to resolve. These loop invariants are easily verified in programs with loops that modify only a single mutable collection.

OpenJML verifies a program by generating verification conditions for each method and discharging them to a single automated prover, Z3. In Why3, by contrast, the verification conditions can be transformed using manual transformations (e.g., splitting, application of hypotheses, etc.), and verification conditions can be discharged to a number of different automated provers, including Z3, CVC4, Alt-Ergo, and Eprover. Manual transformations and the utilisation of different provers exploiting their different strengths can simplify the verification of complex properties considerably.

2.3 Jml_2Why^3: a prototype tool to convert Java/JML to Why3

Jml_2Why^3 is a prototype tool to convert Java programs with JML annotations to programs in the Why3 language, in order to use Why3's generation of verification conditions (VCs), its facility to manually apply transformations to the VCs, and its interaction with automatic provers to verify properties of the Java/JML program. The pipeline of Jml_2Why^3 is illustrated in Figure 2.1.

First, the source code of the Java/JML program is parsed and typed using OpenJML. We added an option to OpenJML to serialise the typed abstract syntax tree (AST) to a file in JSON format. The JSON formatting is performed using the Jackson library. However, typed AST is represented in OpenJML in a complex class hierarchy and cannot be serialised as JSON directly, due to potential cycles in the typing information. We used the Java library Jackson JSOG for the JSON serialisation of the potentially cyclic typed AST. Jackson JSOG assigns a unique identifier to every element, and references to identifiers are used to break loops in the representation. Our modification to OpenJML, which adds the new

command line flag `-tojson` to serialize the type AST as a JSON file, is available in a merge request to OpenJML [10].

Then the typed AST is reconstructed in Jml$_2$Why^3 and converted to a proof-equivalent Why3 program. Jml$_2$Why^3 is implemented in OCaml, and the typed AST is represented as an algebraic data type. To allow for representing of and simple pattern matching on the (potentially cyclic) typed AST, each element of type α is wrapped by type `α obj`, which contains the unique identifier of the element and defers the deserialisation of the actual element lazily (see Figure 2.2). Jml$_2$Why^3's conversion process creates the untyped AST of a Why3 program using Why3's internal API, `Why3.Ptree`. We contributed a pretty printer of the untyped AST to Why3 to serialize an untyped AST to Why3 source code. Details of the conversion process are given in Section 2.3.1.

Finally, Why3 is used to generate VSs for the program created by Jml$_2$Why^3, to simplify the VCs by manual transformations, and to apply automatic theorem provers to their verification.

Compilation of Jml$_2$Why^3 Jml$_2$Why^3 requires at least version 4.08 of OCaml and can be compiled using the following commands:

```
$ opam --version
2.0.2

$ opam switch 4.08.0  # at least 4.08
$ eval $(opam env)
$ opam install --deps-only .
$ make

# To run the proofs
$ why3 config --detect
At least Alt-Ergo (2.3.0), CVC4 (1.6), Z3 (4.6.0).
$ make test
```

2.3.1 Jml$_2$Why^3's conversion of Java/JML to Why3

Jml$_2$Why^3 supports a procedural subset of the Java language that can be mapped to Why3 fairly easily. However, a number of aspects of the conversion require special consideration, and are detailed in the following.

In the following, a *place* refers to a variable, parameter, or field. Values are accessible from places in the current scope by fields, method calls, array indices, which can be composed to *paths*. Java distinguishes between *primitive types* such as `int` and `boolean`, and *class types* with objects as values.

```
type α obj = α Lazy.t * (identifier * position option)

(* Subclasses of com.sun.tools.javac.tree.JCTree.JCExpression *)
and expr' =
  | Ident of {name: string; typ: typ; sym: symbol}
  | ArrayAccess of {indexed: expr; index: expr; typ: typ}
  | MethodInvocation of {typeargs: expr list; meth: expr option; args: expr list; typ: typ; kind:
      jml_clause_kind option}
  | Assign of {lhs: expr; rhs: expr; typ: typ}
  | ...
and expr = expr' obj
```

Figure 2.2 – Example of a algebraic data type to represent the typed AST of the Java/JML program, and the wrapping in `α obj` to deal with cycles in the type AST.

Arrays can hold either elements of primitive types or class types. Other *containers*, for example queues, can only hold elements of class types. Values of primitive types are converted to corresponding values of class types when needed, and vice versa, for example between `int` and `java.lang.Integer`.

We use the syntax $\langle x \rangle$ to designate the conversion of the identifier, type, expression, or term x in Java/JML to Why3. The operation is described in Section 3.1.

Classes Java/JML classes define fields, invariants, constructors, and methods. A definition of a class C is converted by Jml$_2$Why^3 into a number of Why3 definitions.

```
class C {
  final t f; ... // Class field

  // Class invariant i1
  class_invariant i1; ...

  // Method with preconditions p1 and postconditions p2, body ss, and return type t1
  requires p1; ...
  ensures p2; ...
  t1 m(t2 x, ...) { ss }

  // Pure model method with term t as body
  ensures \result = t;
  pure model t1 f(t2 x)

  // Pure model method with term t as body
  requires t1; ...
  ensures \result = t2;
  pure model boolean p(t3 x)
}
```

Currently, all class fields are required to be `final` for Jml$_2$Why^3. The class is represented in Why3 by a record type C with immutable fields corresponding to the Java class fields:

```
type ⟨C⟩ = {f: ⟨t⟩; ...}
```

Each class invariants i_1 is converted into a predicate $C'inv_1$ with an argument representing an instance of C. Notice that such an invariant cannot refer to anything "outside" the object `this`.

```
predicate C'inv1 (this: ⟨C⟩) = ⟨i1⟩
...
```

Jml$_2$Why^3 distinguishes normal class methods from pure model methods. A normal class method m possesses a method body and is not marked by JML keywords `pure` and `model`. A program method is converted into two Why3 definitions: A program value without implementation `val` $C'm$, and a value with implementation `let` $C'm$-*impl*. The two definitions have the same type signature and contract: The first argument is the instance on which the method was called and has type $\langle C \rangle$. The other arguments and the return type correspond to the arguments and return type of the method definition. The contracts are comprised of the JML pre-conditions and post-conditions, the invariants on `this` as pre-condition and post-condition, and the invariant on the result if its type is a user-defined class:

```
val C'm (this: ⟨C⟩) (x: ⟨t2⟩) ...: ⟨t1⟩
  requires { C'inv1(this) } ...
  requires { ⟨p1⟩ } ...
  ensures { ⟨p2⟩ } ...
  ensures { C'inv1(this) } ...
  ensures { t'1inv1(result) } // If t1 is custom class
```

The program definition $C'm\text{-}impl$ contains the method body together with some boiler-plate code to translate Java's return statement to Why3 exceptions and to bind non-final arguments to references to make them assignable (see Section 2.3.1). The method implementation is used to verify that the method body corresponds to its contract. The Why3 annotation `[vc:sp]` changes the generation of VCs to avoid an exponential explosion of the number of generated verification conditions.

```
let C'm-impl (this : C) (x: ⟨t2⟩) ... : ⟨t1⟩
  requires { C'inv1(this) } ...
  requires { p1 } ...
  ensures { p2 } ...
  ensures { C'inv1(this) } ...
  ensures { t'1inv1(result) } // If t1 is custom class
= [vc:sp]
  exception Return' t2 in
  let x = ref x in ... // Redefine non-final arguments as references
  try ⟨ss⟩ with Return' r →r end
```

Pure model methods are declared with JML keywords `pure model` and have no body. They are instead defined by a single postcondition that specifies the result by an equality with `\result`. Pure model methods of other types than `boolean` are converted to logical functions:

```
function C'f (this: ⟨C⟩) (x: ⟨t2⟩) : ⟨t1⟩ = ⟨t⟩
```

A pure model method p with with return type `boolean` is converted to a Why3 predicate. Preconditions of a boolean pure model method are considered as preconditions in the generated predicate.

```
predicate C'p (this: ⟨C⟩) (x: ⟨t3⟩) = ⟨t1⟩ →⟨t2⟩
```

The translation of pure model methods is included before the definition of invariants and methods to allow for using them in method contracts and invariants.

Class inheritance is currently unsupported.

Contracts and terms JML preconditions, postconditions, and loop invariants are translated directly to their counterparts in Why3. Mutation annotations in loops (`loop_modifies`) are ignored since they are inferred in Why3.

Nullability Places and methods of class types are *nullable*, when they are allowed to contain the value `null`. Places are not nullable by default in JML, and nullability is defined by the JML annotation `@nullable`. The nullability of Java expressions and places can be determined statically from the annotations on the involved definitions. Since the value `null` lies outside the domain of the Why3 record types representing the classes, nullability is represented by the type system using an option-like type in module `jml2why3.Nullable`:

```
type nullable α = Null | NonNull α
```

Consequently, the nullability of an expression has to be adjusted for every assignment to a place with mismatching nullability. Non-nullable values are wrapped by `NonNull : α → nullable α` when assigned to nullable assignment targets. Nullable values are asserted to be different than `Null` and their value is extracted using a partial, axiomatized function `Nullable.get_non_null`, when assigned in non-nullable places:

```
function get_non_null (n:nullable α ) : α

axiom get_non_null__spec1: forall n, x: α .
  n ≠ Null → get_non_null n = x → n = NonNull x
axiom get_non_null__spec2: forall n, x: α .
  n = NonNull x → get_non_null n = x
```

Mutability Java allows the mutation of any accessible non-final place (variables, parameters, fields, container contents). Mutability in Why3, however, is restricted to places with statically known paths (e.g., record fields), precluding for example the mutation of fields of array elements (e.g., `a[i].f ← x`).

In Jml$_2$Why^3, mutation is reduced to mutable variables and containers such as arrays and collections. For example, non-final Java variables are converted to Why3 references to allow for reassigning their values. Non-final method parameters are kept as-it in the function signatures but wrapped as references in the method implementation. This has two consequences: All fields of custom-defined classes must be final. And values with mutable content have to be stored in final places (e.g., arrays, queues, objects with mutable fields, and `this`).

This prevents the sharing of mutable places via aliasing. For example, the following Java program is invalid for Jml$_2$Why^3 because the value `ll.get(0)` has the alias `ll.get(1)`.

```
List<List<String>> l = new LinkedList≠();
l.add(new LinkedList≠());
l.add(ll.get(0));
l.get(0).add("Hello");
assert l.get(0).get(0).equals("Hello") && l.get(1).get(0).equals("Hello");
```

Physical equality Tests for physical equality are required to specify that the result of the method is a permutation of `voeuxClasses` in the adapted implementation (Section 2.2.2). The conversion of physical (in-)equality in Java (by operator `==` on values of class types) to Why3 is not possible directly, since there is no notion of physical equality in Why3. On top of that, when comparing values at different points in the program (for example using `\old(x)`), physical equality does not imply logical equality. Physical equality is instead implemented in Jml$_2$Why^3 by adding to each class record a field that contains the address of the object as an abstract type. Tests for physical (in-)equality between objects of user-defined classes are converted to tests of (in-)equality between the values of the address fields.

Currently, physical equality cannot be used together with the allocation of new objects: The address of a newly created object is supplied by the Why3 function `jml2why3.Address.fresh`. However, the *freshness* of the resulting address, i.e., that the result is different than *any other* currently existing address, cannot be easily specified since it has to refer to a global state of addresses that exist so far.

Integers Standard Java integers are signed 32-bit and range from -2^{31} to $2^{31}-1$. They are converted to the corresponding type `mach.int.Int32.int32` in Why3. The absence of arithmetic overflows in operations on this type is ensured by the preconditions of the operands.

Arrays Arrays in the Java program are converted to the Why3's corresponding 32-bit array type `mach.array.Array32.array`, which is comprised of an immutable, 32-bit length, and a mutable ghost mapping of indices to array elements. An array with elements of primitive type t is directly converted to an array `Array32.array ⟨t⟩` since their default values are included in the type (e.g., `0` for type `int`). The values of an array of elements with an object type C are nullable in Java, and such an array is converted to an array with nullable elements in Why3 `Array32.array (nullable ⟨C⟩)`.

Queues Linked lists (`java.util.LinkedList`) are a common implementation of Java queues (`java.util.Queue`). This specific usage of linked lists is converted to Why3's queue (`queue.Queue`). The queue methods are converted to wrapper functions of Why3's queue functions that operate on nullable values. For example, a call to the Java's queue method

```
boolean java.util.Queue.add(E e) // throws NullPointerException, if the specified element is null
```

is converted to a call to the following wrapper function,

```
let nullable_push (v: nullable α) (q: t α) : writes { q.seq }
  requires { v ≠ Null }
```

```
  ensures { match v with NonNull x → q = snoc (old q) x | Null → false end }
= match v with
  | NonNull x → Queue.push x q
  | Null → absurd
  end
```

and `Queue.push` is defined in the Why3 standard library as:

```
  val push (x: α) (q: t α) : unit writes {q}
    ensures { q.elts = old q.elts ++ Cons x Nil }
```

2.3.2 Summary of limitations

The following Java/JML features are not supported. Some of these limitations are *unchecked* by the current prototype, hence a misuse may lead to unsound proofs.

— Class inheritance (checked)
— Mutable fields in user-defined classes (checked)
— Clauses `modifies` and `loop_modifies` are silently ignored (inferred by Why3)
— Tests for physical equality exclude allocation of objects, and vice verse (unchecked)
— Sharing of mutable places via aliasing is unsupported and usually undetected. See example in section 2.3.1 above and the discussion about ownership in Section 2.5.

A technical limitation of Jml$_2$Why^3 is that JML annotations in the Java program cannot be used to define auxiliaries (e.g., lemmas or functions) about the models used in the translation, because there is generally no notion of these models in JML. For example, Java's queues are translated to Why3's queues and the latter are represented by sequences in the logical parts of the program and in the proofs. There is no notion of Why3's sequences in JML, which prohibits the definitions of auxiliaries on sequences in the Java program. In some cases, the auxiliary can be swapped out to a Why3 library (e.g., the Jml$_2$Why^3 library `jml2why3.mlw`). However, when the auxiliary depends on values in the current proof context, this may be also cumbersome.

2.4 Verification of ParcourSup with two constraints using Jml$_2$Why^3

We used the Jml$_2$Why^3 tool to verify the safety and the permutation property of the adapted Java implementation of ParcourSup with two constraints. In both cases, we first implemented the algorithm in Why3 to develop the required loop invariants and lemmas.

2.4.1 Arithmetic and null-pointer safety

The arithmetic safety of the adapted ParcourSup algorithm (Appendix 3.2.2) was verified using Jml$_2$Why^3's pipeline. Why3 generated 83 verification conditions (VCs) for the translation of method `GroupeClassement.calculerOrdreAppel`. Almost all VCs were handled fully automatically using Why3's strategy `Auto level 2` and the automatic theorem provers Alt-Ergo 2.3.0 and CVC4 1.6. The application of manual transformations was necessary in only in five VCs, to convert logical integers from the current verification context to 32-bit integers, to trigger the application of (JML-defined) preconditions quantified over 32-bit integers. Details on the verification of the final 145 sub-tasks are given in Section 2.1. In all VCs, only the result of the fastest prover was retained. The verification of all sub-tasks took 21 seconds using three parallel processes.

2.4.2 Permutation property

The adapted ParcourSup implementation (Section 2.2.2) combines two functionalities:

1. It creates a permutation of input list of wishes respecting ranks and constraints, and

Table 2.1 – The use of automatic theorem provers in the verification conditions (VC) as generated using Jml$_2$Why^{3}for the arithmetic and null-pointer safety of the ParcourSup algorithm (Appendix 3.2.2).

Prover	VCs	Fastest	Slowest	Average
CVC4 1.6	102	0.03	5.37	0.50
Alt-Ergo 2.3.0	13	0.02	0.97	0.26

Table 2.2 – The use of different automatic theorem provers in the verification conditions (VC) as generated using Jml$_2$Why^{3}for safety and the permutation property in the ParcourSup algorithm (Appendix 3.2.4).

Prover	VCs	Fastest	Slowest	Average
CVC4 1.6	192	0.05	5.66	0.64
Alt-Ergo 2.3.0	46	0.14	3.40	0.62
Z3 4.6.0	13	0.06	0.50	0.26

```
class Permutation {
    public final int[] map, inv;

    /*@ public invariant map.length == inv.length;
        public invariant (\forall int i; 0 ≤ i && i < map.length;
            0 ≤ map[i] && map[i] < map.length && inv[map[i]] == i);
        public invariant (\forall int i; 0 ≤ i && i < map.length;
            0 ≤ inv[i] && inv[i] < map.length && map[inv[i]] == i); @*/
}
```

Figure 2.3 – Fields and invariants of class `Permutation`

2. re-arranges the wishes in the resulting list of wishes.

This increases the complexity of the implementation, the complexity of the specification of the permutation property, and its the verification.

We verified the permutation property instead in a modified implementation that produces an explicit permutation (Appendix 3.2.4). This explicit permutation enables a simpler formalisation of the correctness properties of the ParcourSup algorithm [13]. A permutation of a Java array is defined a class with two fields: An array `map` that maps indices of the permutation to indices of the input array of wishes, and an array `inv` that maps indices in the input array of wishes to indices in the resulting permutation. The invariants of the class ensure that `map` and `inv` describe a permutation (Figure 2.3). For example

```
input:       [a,b,c,d,e]     map: [2,1,3,4,0]
permutated:  [c,b,d,e,a]     inv: [4,1,0,2,3]
```

For the verification of arithmetic safety and null-pointer safety for method `GroupeClassement.calculerOrdreAppel` in the permutation-based implementation of the ParcourSup algorithm, Why3 generated 164 VCs for the Why3 translation. The verification of these conditions using automatic provers required a substantial amount of manually applied transformations. The resulting 319 sub-tasks were proven in 59 seconds using three parallel processes using CVC4 1.6, Alt-Ergo 2.3.0, and Z3 4.6.0 (see Section 2.2).

2.5 Excursion to ownership and proofs in Rust

An experiment shows that the ParcourSup algorithm can be implemented and specified without aliasing using Rust-style ownership (see Appendix 3.2.5). In the first loop (line 43 ff.), the ownership of each of the input wishes is transferred from argument `voeux_classes` to one of the four queues. The best candidate `meilleur` is defined as an mutable option of an immutable reference, in order to leave the ownership of the head element at the queues while determining the best candidate (lines 77-92). Only when removing the best candidate from its associated queue, the ownership is transferred with the actual modifications from the queue to the result order of wishes, `ordre_appel` (lines 94-110).

Under the affine semantics as in Rust, it may be impossible to specify the result of a function in terms of its arguments with normal post-conditions: Ownership of the arguments is often moved to the result, which makes the arguments unavailable at the end of the function.

An extension of normal contracts for specifying programs with affine semantics has been proposed by the Prusti verifier [4]. Prusti implements the verification of normal JML-style contracts for Rust, and adds *pledges* to relate values whose ownership is lost at the end of a method (e.g., argument x) with currently accessible values (e.g., result y). For example, a pledge `after_expiry<`y`>(... `x` ...before_expiry (`y`) ...)` is a logical statement that is checked at the border of validity of the function result x, when ownership returns to argument x.

In the Rust implementation of the ParcourSup algorithm (Appendix 3.2.5), we use pledges to specify the permutation property in a post-condition (lines 28-31). The record field `rang_appel` of the input wishes is used as index in the resulting ordering of wishes, and the record field `rang` is used as the index of a wish in the resulting order in the array of input wishes. Our attempts to use Prusti to verify the specifications of the Rust program, however, were thwarted by Prusti's missing support for the Rust libraries used in the program.

Bibliography

[1] AdaCore and Thales. Implementation guidance for the adoption of SPARK. https://www.adacore.com/books/implementation-guidance-spark, 2018.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[3] Léo Andrès. Vérification par preuve formelle de propriétés fonctionnelles d'algorithme de classification. Rapport de stage de M1, Université Paris Sud, August 2019.

[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA):147:1–147:30, October 2019.

[5] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. See also http://toccata.lri.fr/gallery/fm2012comp.en.html.

[6] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified Compilation of Floating-Point Computations. *Journal of Automated Reasoning*, 54(2):135–163, February 2015.

[7] David R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment*, volume 149 of *EPTCS*, pages 79–92, 2014.

[8] OpenJML contributors. OpenJML. http://www.openjml.org/.

[9] OpenJML contributors. OpenJML installation instructions. http://www.openjml.org/documentation/installation.shtml#plugininstallation.

[10] OpenJML contributors. OpenJML pull request 691. https://github.com/OpenJML/OpenJML/issues/691.

[11] Eclipse contributors. Eclipse installation packages. https://www.eclipse.org/downloads/packages/.

[12] Hugo Gimbert. Code source en Java de ParcourSup. https://framagit.org/parcoursup/algorithmes-de-parcoursup.

[13] Hugo Gimbert. Document de présentation des algorithmes de ParcourSup. https://framagit.org/parcoursup/algorithmes-de-parcoursup/blob/master/doc/presentation_algorithmes_parcoursup_2019.pdf.

[14] Hugo Gimbert. Site principal de ParcourSup. https://www.parcoursup.fr/.

[15] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[16] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In Tiziana Margaria and Bernhard Steffen, editors, *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 9952 of *Lecture Notes in Computer Science*, pages 461–478, Corfu, Greece, October 2016. Springer.

[17] K. Rustan M. Leino and Valentin Wüstholz. The Dafny integrated development environment. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, volume 149 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–15, 2014.

[18] Claude Marché. The Krakatoa tool for deductive verification of Java programs. Winter School on Object-Oriented Verification, Viinistu, Estonia, January 2009. http://krakatoa.lri.fr/ws/.

[19] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. http://krakatoa.lri.fr.

[20] Denis Merigoux, Raphaël Monat, and Christophe Gaie. Étude formelle de l'implémentation du code des impôts. In Zaynah Dargaye and Yann Régis-Gianas, editors, *Trente-et-unièmes Journées Francophones des Langages Applicatifs*, Gruissan, France, January 2020.

[21] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. How to get an efficient yet verified arbitrary-precision integer library. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, pages 84–101, Heidelberg, Germany, July 2017.

Chapter 3

Annexes

3.1 Jml$_2$Why^{3}conversions

In this section, we describe in detail the conversion expressions, types, and terms in Java/JML to Why3. We use the syntax ⟨x⟩ for the conversion of a Java element `x` to Why3. Two macros are used in the conversion to deal with nullable values: `x`|*`nullable`* forces `x` to be nullable, i.e. it expands to `get_non_null x` if `x` is not nullable, and `x` otherwise. Similary, `x`|*`nonnull`* forces `x` to be non-null, and expands for nullable expressions to `NonNull x`.

The conversions are described in the following form:

Description of syntactic element:

`Java pattern` — *Condition on Java pattern (optional)*

— `Why3 pattern 1` — *Conditions for pattern* 1

— `Why3 pattern 2` — *Conditions for pattern* 2

— `...`

The Why3 program resulting from the conversion makes use of the Why3 library `jml2why3` (`jml2why3.mlw`). The library provides a number of modules:

— `Nullable`: an option-like type for nullable Java values, and accompanying definitions

```
type nullable α = Null | NonNull α
```

— `Queue`: redefinition of Why3's queue functions to be closer to `java.util.Queue`

— `LinkedList`: wrapper of Why3 sequence with an interface closer to java.util.LinkedList

```
type linked_list α = { mutable seq : seq (nullable α) }
```

— `Address`: an address is an abstract type and the modules defines equality tests to model physical equality

— `Utls`: utility definitions on existing modules

3.1.1 Identifiers

A mapping of Java identifiers of custom classes, methods, variables, and fields and to Why3 identifiers and related information is stored in a global environment. This environment is populated by analysing the Java definitions, and constitutues the basis for translating Java identifiers to Why3.

3.1.2 Types

— `int`
  — `int.Int32`
— `java.lang.Integer`
  — `int.Int32`
— `bool`
  — `bool.Bool`
— `java.util.Queue`
  — `Queue.t`
— *user-defined class:*
  `C`
  — `info(C).ident` — *in mutable context*
  — `info(C).ident_imm` — *in immutable context*
— *primitive array:*
  `t[]`
  — `PrimitiveArray.array ‹t›`
— *object array:*
  `C[]`
  — `ObjectArray.array ‹C›`

3.1.3 Expressions and terms

— *Null:*
  `null` — *zero*
  — `Nullable.Null` — *Nada* — *zielch*
— *Literal of type `int`:*
  `l`
  — `(l:int32)`
— *Literal of type `boolean`:*
  `l`
  — `True or False`
— *This:*
  `this`
  — `this`
— *Variable or parameter:*
  `x`
  — `‹x›` — *Identifier ‹x› from variable info*
— *Field:*
  `f`
  — `this.‹f›` — *identifier ‹f› from field info*
— *Field access:*
  `e.f`
  — `‹e`|*nonnull*`›.‹f›` — *identifier ‹f› from field info*
— *Binary operation for operators `"&&"` and `"||"`:*
  `e1 op e2`
  — `‹e1› ‹op› ‹e2›`
— *Binary operation between integer operants (`int`/`Integer`):*
  `e1 op e2`
  — `not (‹e1› = ‹e2›)` — *op is !=*
  — `‹e1› ‹op› ‹e2›` — *op is = or op*

— *Equality/inequality between boolean operants:*
```
e1 == e2
— Utils.biff ‹e1›!  ‹e2›!                    — if op is "=="
— Bool.bxor ‹e1›!  ‹e2›!                     — if op is "!="
```
— *Test null/nonnull:*
```
null op x                                    — op is "==" or "!="
— Nullable.is_null ‹x›                       — negate if op is !=
```
— *Test null/nonnull:*
```
x op null                                    — op is "==" or "!="
— Nullable.is_null ‹x›                       — negate if op is !=
```
— *Physical equality between obects of user-defined classes:*
```
e1 op e2                                     — op is == or !=
— Address.same ‹e1› ‹e2›                     — if e1, e2 non-nullable
                                             — negate if op is !=
— Address.nullable_same_address ‹e1›!‹e2›    — if e1 or e2 nullable
                                             — negate if op is !=
```
— *Negation unary operator:*
```
!x
— not ‹x›
```
— *Array length:*
```
x.length                                     — x has type array
— PrimitiveArray.length ‹x›                  — x is array of primitive types
— ObjectArray.length ‹x›                     — x is array of object types
```
— *Maximum integer constant:*
```
Integer.MAX_VALUE
— Int32.max_int32
```
— *Peek from queue:*
```
x.peek()                                     — x has type java.util.Queue
— (Queue.nullable_peek ‹e›)|mut
```
— *Size of queue:*
```
x.size()                                     — x has type java.util.Queue
— Queue.length ‹e›                           — if target is expr
— Queue.length ‹e› |> peano_to_int32         — if target is term
```
— *Add to queue:*
```
e1.add(e2)                                   — e1 has type java.util.Queue
— Queue.nullable_push (‹e1›|nonnull) ‹e2›|nullable|immutable
```

3.1.4 Expressions

— *New array:*
```
new t[N]
— PrimitiveArray.(make ‹default› N : array ‹t›)   — if t primitive type
— ObjectArray.(make null N : array ‹t›)           — if t object type
```
— *Constructor call:*
```
new C(args)
— let this = {‹fields = defaults›} in ‹C› this ‹args›   — ‹C› is the identifier of the constructor
```
— *Method call:*
```
e.m(args)
— ‹m› ‹e› ‹args›
```
— *Method call:*
```
m(args)
— ‹m› this ‹args›
```

— *Test for empty queue:*
`e.isEmpty()` — *e has type* `java.util.Queue`
— *Poll from queue:*
`e.poll()` — *e has type* `java.util.Queue`
— *Array access:*
`e1[e2]`
— `PrimitiveArray.([])` `‹e1›`|*nonnull* `‹e2›`|*nonnull* — *e1 is array of primitive type elements*
— `ObjectArray.([])` `‹e1›`|*nonnull* `‹e2›`|*nonnull* — *e1 is array of object elements*
— *Assignment to variable:*
`‹x› = ‹e›` — *‹x› is non-final*
— `‹x›.contents ← ‹e›`
— *Assignment to field:*
`‹x› = ‹e›` — *‹f› is non-final*
— `this.x ← ‹e›`
— *Assignment to array element:*
`e[i] = rhs`
— `‹e›`|*nonnull*`[‹i›] ← ‹rhs›`|*imm*|*nullable*
— *Assignment to field of array element:*
`a[i].f = e`
— `let x = ‹a›[‹i›]`|*nonnull*|*mut* `in x.f ← ‹e›; ‹a›[‹i›] ← v`|*imm*|*nullable*
— *Assignment to (non-array) field access:*
`e1.f = e2`
— `‹e1›.f ← ‹e2›` — *Force rhs same nullability as f*

3.1.5 Terms

— `\result`
— `result`
— *op is* `||`, *&&*, *==>*, *<==>*:
`e1 op e2`
— `‹e1› ‹op› ‹e2›`
— `e.content.theSize`
— `Seq.length ‹e›`
— *Array access:*
`a[i]`
— `Seq.([]) ‹a.values› ‹i›`
— *e has type java.util.Queue:*
`e.isEmpty()`
— `‹e› = Seq.empty`
— *e has type java.util.Queue:*
`e._get(i)`
— `(Seq.get ‹e› ‹i›)`|*mut*
— *m refers to a boolean pure model method:*
`m(args)`
— `sym this ‹args›`
— `\old(e)`
— `old e`
— `\old(e, l)`
— `(e at l)`
— *e is an array of objects:*
`\nonnullelements(e)`

```
    — ObjectArray.non_null_elements ‹e›
— \forall ‹decls› ‹range› ‹value›
    — forall ‹decls›.  ‹range› → ‹value›
— Special case of \num_of as occ:
    \num_of int v; l <= v && v < u; a[v] == x
    — occ x a l u
```

3.1.6 Statements

```
— Special case:
    Queue x = new LinkedList<>(); ...
    — let x = Queue.create() in ...
— Variable declaration:
    t x = e; ...
    — let x = e in ...
— Call to super inserted by OpenJDK, ignored in conversion:
    super(); ...
    — ...
```

3.1.7 Statement

```
— l:  { body }
    — label ‹l› in ‹body›
— Expression:
    expr
    — ‹expr›
— Block:
    { stats }
    — ‹stats›
— if (cond) {thenpart} {elsepart?}
    — if ‹cond› ‹thenpart› ‹elsepart›                                  — When elsepart defined
    — if ‹cond› ‹thenpart› ()                                          — When no elsepart
— ‹invariants› for (int i=e1; i < e2; i++) ‹stats›
    — let i = ref ‹e1› in while !i < ‹e2› do variant { ‹e2› - !i } invariant { ‹e1› <= i <=
      ‹e2› } ‹invariants› ‹stats›; i := !i+1 done
— try { body } except (E e) {stats} ...
    — match ‹body› with exception E → ‹stats› ...
— except (E e) { stats }
— return e
    — raise (Return' ‹e›)
— E is user-defined exception:
    raise E
    — raise E
— assert e
    — assert ‹e›
```

3.1.8 Exception declaration

```
— class E extends Exception {}
    — exception e
```

3.2 ParcourSup implementations

3.2.1 Original implementation

As of commit `7be9a08e` in the Git repository of ParcourSup[12].

```
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Queue;

public class GroupeClassement {

    /* le code identifiant le groupe de classement dans la base de données Remarque: un
        même groupe de classement peut être commun à plusieurs formations
     */
    public final int C_GP_COD;

    /* le taux minimum de boursiers dans ce groupe d α ppel (nombre min de boursiers pour
        100 candidats) */
    public final int tauxMinBoursiersPourcents;

    /* le taux minimum de résidents dans ce groupe d α ppel (nombre min de résidents pour
        100 candidats) */
    public final int tauxMinResidentsPourcents;

    /* la liste des voeux du groupe de classement */
    public final List<VoeuClasse> voeuxClasses = new LinkedList≠();

    public GroupeClassement(
            int C_GP_COD,
            int tauxMinBoursiersPourcents,
            int tauxMinResidentsPourcents) {
        this.C_GP_COD = C_GP_COD;
        this.tauxMinBoursiersPourcents = tauxMinBoursiersPourcents;
        this.tauxMinResidentsPourcents = tauxMinResidentsPourcents;
    }

    public void ajouterVoeu(VoeuClasse v) {
        voeuxClasses.add(v);
    }

    /* calcule de l'ordre d α ppel */
    OrdreAppel calculerOrdreAppel() {


        /* on crée autant de listes de voeux que de types de candidats, triées par ordre
            de classement */
        Map<VoeuClasse.TypeCandidat, Queue<VoeuClasse>> filesAttente
                = new HashMap≠();

        for (VoeuClasse.TypeCandidat type : VoeuClasse.TypeCandidat.values()) {
            filesAttente.put(type, new LinkedList≠());
        }

        /* Chaque voeu classé est ventilé dans la liste correspondante, en fonction du
        type du candidat. Les quatre listes obtenues sont ordonnées par rang de
        classement, comme l'est la liste voeuxClasses. */
        int nbBoursiersTotal = 0;
        int nbResidentsTotal = 0;

        /* on trie les voeux par classement */
        voeuxClasses.sort((VoeuClasse v1, VoeuClasse v2) → v1.rang - v2.rang);

        for (VoeuClasse voe : voeuxClasses) {

            /* on ajoute le voeu à la fin de la file (FIFO) correspondante */
            filesAttente.get(voe.typeCandidat).add(voe);

            if (voe.estBoursier()) {
                nbBoursiersTotal++;
            }
            if (voe.estResident()) {
```

```
                nbResidentsTotal++;
            }
        }

        int nbAppeles = 0;
        int nbBoursiersAppeles = 0;
        int nbResidentsAppeles = 0;

        /* la boucle ajoute les candidats un par un à la liste suivante, dans l'ordre
            d α ppel */
        OrdreAppel ordreAppel = new OrdreAppel();

        while (ordreAppel.voeux.size() < voeuxClasses.size()) {

            /* on calcule lequel ou lesquels des critères boursiers et résidents
                contraignent le choix du prochain candidat dans l'ordre d α ppel */
            boolean contrainteTauxBoursier
                    = (nbBoursiersAppeles < nbBoursiersTotal)
                    && (nbBoursiersAppeles * 100 < tauxMinBoursiersPourcents * (1 + nbAppeles));

            boolean contrainteTauxResident
                    = (nbResidentsAppeles < nbResidentsTotal)
                    && (nbResidentsAppeles * 100 < tauxMinResidentsPourcents * (1 + nbAppeles));

            /* on fait la liste des voeux satisfaisant les deux contraintes à la fois,
                ordonnée par rang de classement */
            PriorityQueue<VoeuClasse> eligibles = new PriorityQueue≠();

            for (Queue<VoeuClasse> queue : filesAttente.values()) {
                if (!queue.isEmpty()) {
                    VoeuClasse voe = queue.peek();
                    if ((voe.estBoursier() || !contrainteTauxBoursier)
                            && (voe.estResident() || !contrainteTauxResident)) {
                        eligibles.add(voe);
                    }
                }
            }

            /* stocke le meilleur candidat à appeler tout en respectant les deux
            contraintes si possible ou à défaut seulement la contrainte sur le taux
            boursier */
            VoeuClasse meilleur = null;

            if (!eligibles.isEmpty()) {
                meilleur = eligibles.peek();
            } else {
                /* la liste peut être vide dans le cas où les deux contraintes ne peuvent
                être satisfaites à la fois. Dans ce cas nécessairement il y a une
                contrainte sur chacun des deux taux (donc au moins un boursier non encore
                sélectionné) et il ne reste plus de boursier résident, donc il reste au
                moins un boursier non résident */
                assert contrainteTauxBoursier && contrainteTauxResident;
                assert filesAttente.get(VoeuClasse.TypeCandidat.BoursierResident).isEmpty();
                assert !filesAttente.get(VoeuClasse.TypeCandidat.BoursierNonResident).isEmpty();

                Queue<VoeuClasse> CandidatsBoursierNonResident
                        = filesAttente.get(VoeuClasse.TypeCandidat.BoursierNonResident);

                meilleur = CandidatsBoursierNonResident.peek();
            }

            /* suppression du candidat choisi de sa file d α ttente */
            Queue<VoeuClasse> queue = filesAttente.get(meilleur.typeCandidat);
            assert meilleur == queue.peek();
            queue.poll();

            /* ajout du meilleur candidat à l'ordre d α ppel*/
            ordreAppel.voeux.add(meilleur);

            /* mise à jour des compteurs */

            nbAppeles++;

            if (meilleur.estBoursier()) {
                nbBoursiersAppeles++;
            }
```

```
146             if (meilleur.estResident()) {
147                 nbResidentsAppeles++;
148             }
149         }
150         for (int i=0; i<ordreAppel.voeux.size(); i++) {
151             ordreAppel.voeux.get(i).rangAppel = i+1;
152         }
153         return ordreAppel;
154     }
155 }
```

3.2.2 Adapted implementation

File `jml2why3/examples/GroupeClassement.java`.

```
import java.util.Map;
import java.util.HashMap;
import java.util.Queue;
import java.util.PriorityQueue;
import java.util.LinkedList;

class VoeuClasse {

    public final boolean estBoursier;
    public final boolean estResident;
    public final int rang;

    //@ public invariant 1 ≤ rang;

    //@ requires 1 ≤ rang;
    public /*@ pure @*/ VoeuClasse(
            final int rang,
            final boolean estBoursier,
            final boolean estResident) {
        this.rang = rang;
        this.estBoursier = estBoursier;
        this.estResident = estResident;
    }

    /*@ ensures \result == (this.rang == other.rang && this.estBoursier == other.estBoursier && this.estResident == other.estResident);
        public pure model boolean equals(final VoeuClasse other); @*/
}

public class GroupeClassement {

    /*le code identifiant le groupe de classement dans la base de données
        Remarque: un même groupe de classement peut être commun à plusieurs formations
     */
    public final int C_GP_COD;

    /* le taux minimum de boursiers dans ce groupe d α ppel
        (nombre min de boursiers pour 100 candidats) */
    public final int tauxMinBoursiersPourcents;

    /* le taux minimum de candidats du secteur dans ce groupe d α ppel
        (nombre min de candidats du secteur pour 100 candidats) */
    public final int tauxMinDuSecteurPourcents;

    /*@ invariant 0 ≤ tauxMinBoursiersPourcents && tauxMinBoursiersPourcents ≤ 100;
        invariant 0 ≤ tauxMinDuSecteurPourcents && tauxMinDuSecteurPourcents ≤ 100; @*/

    /*@ requires 0 ≤ tauxMinBoursiersPourcents && tauxMinBoursiersPourcents ≤ 100;
        requires 0 ≤ tauxMinResidentsPourcents && tauxMinResidentsPourcents ≤ 100; @*/
    public /*@ pure @*/ GroupeClassement(
            final int C_GP_COD,
            final int tauxMinBoursiersPourcents,
            final int tauxMinResidentsPourcents) {
        this.C_GP_COD = C_GP_COD;
        this.tauxMinBoursiersPourcents = tauxMinBoursiersPourcents;
        this.tauxMinDuSecteurPourcents = tauxMinResidentsPourcents;
    }

    //@ requires voeuxClasses.size() * 100 < Integer.MAX_VALUE; // strictly smaller to enable 1-indexed rang, rangAppel
    //@ requires_redundantly voeuxClasses.size() < 21_474_836;
    //@ requires (\forall int i; 0 ≤ i && i < voeuxClasses.size(); voeuxClasses.get(i) != null);
    public LinkedList<VoeuClasse> calculerOrdreAppel(final LinkedList<VoeuClasse> voeuxClasses) {
        final Queue<VoeuClasse> BR = new LinkedList≠(), BnR = new LinkedList≠(), nBR = new LinkedList≠(), nBnR = new LinkedList≠();

        //@ loop_invariant \invariant_for(BR) && \invariant_for(BnR) && \invariant_for(nBR) && \invariant_for (nBnR);
        //@ loop_invariant (\forall int j; 0 ≤ j && j < voeuxClasses.size(); voeuxClasses.get(j) != null);
        //@ loop_invariant 0 ≤ i && i ≤ voeuxClasses.size();
        //@ loop_invariant BR.size() + BnR.size() + nBR.size() + nBnR.size() == i;
        //@ loop_invariant (\forall int j; 0 ≤ j && j < BR.size();   BR._get(j).estBoursier && BR._get(j).estResident);
        //@ loop_invariant (\forall int j; 0 ≤ j && j < BnR.size();  BnR._get(j).estBoursier && !BnR._get(j).estResident);
        //@ loop_invariant (\forall int j; 0 ≤ j && j < nBR.size();  !nBR._get(j).estBoursier && nBR._get(j).estResident);
```

```
        //@ loop_invariant (\forall int j; 0 ≤ j && j < nBnR.size(); !nBnR._get(j).estBoursier && !nBnR._get(j).estResident);
        //@ loop_modifies BR.content, BnR.content, nBR.content, nBnR.content;
        for (int i = 0; i < voeuxClasses.size(); i++) {
            final VoeuClasse voe = voeuxClasses.get(i);
            if (voe.estBoursier) {
                if (voe.estResident)
                    BR.add(voe);
                else
                    BnR.add(voe);
            } else {
                if (voe.estResident)
                    nBR.add(voe);
                else
                    nBnR.add(voe);
            }
        }

        final int nbBoursiersTotal = BR.size() + BnR.size();
        final int nbResidentsTotal = BR.size() + nBR.size();

        final LinkedList<VoeuClasse> ordreAppel = new LinkedList≠();

        //@ loop_invariant nbAppeles == ordreAppel.size();
        //@ loop_invariant \invariant_for(BR) && \invariant_for(BnR) && \invariant_for(nBR) && \invariant_for(nBnR);
        //@ loop_invariant 0 ≤ ordreAppel.size() && ordreAppel.size() ≤ voeuxClasses.size();
        //@ loop_invariant 0 ≤ BR.size() + BnR.size() && BR.size() + BnR.size() ≤ nbBoursiersTotal;
        //@ loop_invariant 0 ≤ BR.size() + nBR.size() && BR.size() + nBR.size() ≤ nbResidentsTotal;
        //@ loop_invariant ordreAppel.size() + BR.size() + BnR.size() + nBR.size() + nBnR.size() == voeuxClasses.size();

        //@ loop_invariant (\forall int i; 0 ≤ i && i < BR.size();   BR._get(i).estBoursier && BR._get(i).estResident);
        //@ loop_invariant (\forall int i; 0 ≤ i && i < BnR.size();  BnR._get(i).estBoursier && !BnR._get(i).estResident);
        //@ loop_invariant (\forall int i; 0 ≤ i && i < nBR.size();  !nBR._get(i).estBoursier && nBR._get(i).estResident);
        //@ loop_invariant (\forall int i; 0 ≤ i && i < nBnR.size(); !nBnR._get(i).estBoursier && !nBnR._get(i).estResident);

        //@ loop_invariant (\forall int i; 0 ≤ i && i < ordreAppel.size(); ordreAppel.get(i) != null);

        //@ loop_modifies ordreAppel.content, BR.content, BnR.content, nBR.content, nBnR.content;
        // while (ordreAppel.size() < voeuxClasses.size()) {
        for (int nbAppeles = 0; nbAppeles < voeuxClasses.size(); nbAppeles++) {

            final int nbBoursierRestants = BR.size() + BnR.size();
            final int nbResidentsRestants = BR.size() + nBR.size();
            final int nbBoursiersAppeles = nbBoursiersTotal - nbBoursierRestants;
            final int nbResidentsAppeles = nbResidentsTotal - nbResidentsRestants;

            final boolean contrainteTauxBoursier = 0 < nbBoursierRestants
                && (nbBoursiersAppeles * 100 < tauxMinBoursiersPourcents * (1 + ordreAppel.size()));

            //@ assert contrainteTauxBoursier ==> 0 < BR.content.theSize || 0 < BnR.content.theSize;

            final boolean contrainteTauxResident = 0 < nbResidentsRestants
                && (nbResidentsAppeles * 100 < tauxMinDuSecteurPourcents * (1 + ordreAppel.size()));

            //@ assert contrainteTauxResident ==> 0 < BR.content.theSize || 0 < nBR.content.theSize;

            /*@ nullable @*/ VoeuClasse meilleur = null;
            if (!BR.isEmpty()) {
                meilleur = BR.peek();
            }
            if (!nBR.isEmpty() && !contrainteTauxBoursier && (meilleur == null || meilleur.rang < nBR.peek().rang)) {
                meilleur = nBR.peek();
            }
            if (!BnR.isEmpty() && !contrainteTauxResident && (meilleur == null || meilleur.rang < BnR.peek().rang)) {
                meilleur = BnR.peek();
            }
            if (!nBnR.isEmpty() && !contrainteTauxBoursier && !contrainteTauxResident && (meilleur == null || meilleur.rang < nBnR.
                  peek().rang)) {
                meilleur = nBnR.peek();
            }
            if (meilleur == null) {
                //@ assert 0 < BnR.size();
                meilleur = BnR.peek();
            }

            /*@ assert (0 < BR.size() && meilleur.equals(BR._get(0))) ||
                      (0 < BnR.size() && meilleur.equals(BnR._get(0))) ||
```

```
                        (0 < nBR.size() && meilleur.equals(nBR._get(0))) ||
                        (0 < nBnR.size() && meilleur.equals(nBnR._get(0))); @*/

            if (meilleur.estBoursier) {
                if (meilleur.estResident) {
                    //@ assert 0 < BR.size();
                    BR.poll();
                } else {
                    //@ assert 0 < BnR.size();
                    BnR.poll();
                }
            } else {
                if (meilleur.estResident) {
                    //@ assert 0 < nBR.size();
                    nBR.poll();
                } else {
                    //@ assert 0 < nBnR.size();
                    nBnR.poll();
                }
            }

            ordreAppel.add(meilleur);
        }

        return ordreAppel;
    }
}
```

3.2.3 Arrays-based implementation for OpenJML

File `openjml/GroupeClassement__everything_arrays.java`.

```
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

class VoeuClasse {

    // public final int G_CN_COD;
    public final boolean estBoursier;
    public final boolean estResident;
    public final int rang;

    public /*@ pure @*/ VoeuClasse(
            int rang,
            boolean estBoursier,
            boolean estResident) {
        this.rang = rang;
        this.estBoursier = estBoursier;
        this.estResident = estResident;
    }
}

class GroupeClassement {

    /*le code identifiant le groupe de classement dans la base de données
        Remarque: un même groupe de classement peut être commun à plusieurs formations
     */
    public final int C_GP_COD;

    /* le taux minimum de boursiers dans ce groupe d'appel
        (nombre min de boursiers pour 100 candidats) */
    public final int tauxMinBoursiersPourcents;

    /* le taux minimum de candidats du secteur dans ce groupe d'appel
        (nombre min de candidats du secteur pour 100 candidats) */
    public final int tauxMinDuSecteurPourcents;

    /*@ invariant 0 ≤ tauxMinBoursiersPourcents && tauxMinBoursiersPourcents ≤ 100;
        invariant 0 ≤ tauxMinDuSecteurPourcents && tauxMinDuSecteurPourcents ≤ 100; @*/


    /*@ requires 0 ≤ tauxMinBoursiersPourcents && tauxMinBoursiersPourcents ≤ 100;
        requires 0 ≤ tauxMinResidentsPourcents && tauxMinResidentsPourcents ≤ 100; @*/
    public /*@ pure @*/ GroupeClassement(
            int C_GP_COD,
            int tauxMinBoursiersPourcents,
            int tauxMinResidentsPourcents) {
        this.C_GP_COD = C_GP_COD;
        this.tauxMinBoursiersPourcents = tauxMinBoursiersPourcents;
        this.tauxMinDuSecteurPourcents = tauxMinResidentsPourcents;
    }

    /* calcule de l'ordre d'appel */
    /*@ requires voeuxClasses.length * 100 < Integer.MAX_VALUE; // strictly smaller to enable 1-indexed rangAppel
        requires_redundantly voeuxClasses.length < 21_474_836;
        requires \nonnullelements(voeuxClasses);
        requires (\forall int i, j; 0 ≤ i < voeuxClasses.length && 0 ≤ j < voeuxClasses.length && i < j; voeuxClasses[i].rang <
              voeuxClasses[j].rang);
        ensures \nonnullelements(\result);
        assignable \nothing;
      @*/
    public VoeuClasse[] calculerOrdreAppel(final VoeuClasse[] voeuxClasses) {

        /* on crée autant de listes de candidats ue de types de candidats,
            triées par ordre de classement */

        // Look at these poor man's queues!
        VoeuClasse[]
            BR = new VoeuClasse[voeuxClasses.length],
            BnR = new VoeuClasse[voeuxClasses.length],
            nBR = new VoeuClasse[voeuxClasses.length],
            nBnR = new VoeuClasse[voeuxClasses.length];
        int BR_end = 0, BnR_end = 0, nBR_end = 0, nBnR_end = 0;
```

```
        /*@ loop_invariant 0 ≤ i && i ≤ voeuxClasses.length;
            loop_invariant BR_end + BnR_end + nBR_end + nBnR_end == i;
            loop_invariant 0 ≤ BR_end && BR_end ≤ i;
            loop_invariant 0 ≤ BnR_end && BnR_end ≤ i;
            loop_invariant 0 ≤ nBR_end && nBR_end ≤ i;
            loop_invariant 0 ≤ nBnR_end && nBnR_end ≤ i;

            loop_invariant (\forall int j; 0 ≤ j && j < BR_end;   BR[j] != null);
            loop_invariant (\forall int j; 0 ≤ j && j < BnR_end;  BnR[j] != null);
            loop_invariant (\forall int j; 0 ≤ j && j < nBR_end;  nBR[j] != null);
            loop_invariant (\forall int j; 0 ≤ j && j < nBnR_end; nBnR[j] != null);

            loop_invariant (\forall int j; 0 ≤ j && j < BR_end;    BR[j].estBoursier   &&  BR[j].estResident);
            loop_invariant (\forall int j; 0 ≤ j && j < BnR_end;   BnR[j].estBoursier  && !BnR[j].estResident);
            loop_invariant (\forall int j; 0 ≤ j && j < nBR_end;  !nBR[j].estBoursier  &&  nBR[j].estResident);
            loop_invariant (\forall int j; 0 ≤ j && j < nBnR_end; !nBnR[j].estBoursier && !nBnR[j].estResident);

            loop_modifies i, BR[*], BnR[*], nBR[*], nBnR[*];
            loop_modifies BR_end, BnR_end, nBR_end, nBnR_end;
          @*/
        for (int i=0; i<voeuxClasses.length; i++) {
            final VoeuClasse voe = voeuxClasses[i];
            if (voe.estBoursier) {
                if (voe.estResident) {
                    BR[BR_end++] = voe;
                } else {
                    BnR[BnR_end++] = voe;
                }
            } else {
                if (voe.estResident) {
                    nBR[nBR_end++] = voe;
                } else {
                    nBnR[nBnR_end++] = voe;
                }
            }
        }

        int nbBoursiersTotal = BR_end + BnR_end;
        int nbResidentsTotal = BR_end + nBR_end;

        int BR_first = 0, BnR_first = 0, nBR_first = 0, nBnR_first = 0;

        /* la boucle ajoute les candidats un par un à la liste suivante,
            dans l'ordre d'appel */
        VoeuClasse[] ordreAppel = new VoeuClasse[voeuxClasses.length];

        /*@ loop_invariant 0 ≤ nbAppeles && nbAppeles ≤ ordreAppel.length;
            loop_invariant 0 ≤ BR_first && BR_first ≤ BR_end;
            loop_invariant 0 ≤ BnR_first && BnR_first ≤ BnR_end;
            loop_invariant 0 ≤ nBR_first && nBR_first ≤ nBR_end;
            loop_invariant 0 ≤ nBnR_first && nBnR_first ≤ nBnR_end;
            loop_invariant 0 ≤ (BR_end-BR_first) + (BnR_end-BnR_first) && (BR_end-BR_first) + (BnR_end-BnR_first) ≤ nbBoursiersTotal
                ;
            loop_invariant 0 ≤ (BR_end-BR_first) + (nBR_end-nBR_first) && (BR_end-BR_first) + (nBR_end-nBR_first) ≤ nbResidentsTotal
                ;
            loop_invariant nbAppeles + (BR_end-BR_first) + (BnR_end-BnR_first) + (nBR_end-nBR_first) + (nBnR_end-nBnR_first) ==
                voeuxClasses.length;

            loop_invariant (\forall int j; BR_first ≤ j &&   j < BR_end;     BR[j] != null);
            loop_invariant (\forall int j; BnR_first ≤ j &&  j < BnR_end;   BnR[j] != null);
            loop_invariant (\forall int j; nBR_first ≤ j &&  j < nBR_end;   nBR[j] != null);
            loop_invariant (\forall int j; nBnR_first ≤ j && j < nBnR_end; nBnR[j] != null);

            loop_invariant (\forall int i; BR_first ≤ i   && i < BR_end;   BR[i].estBoursier && BR[i].estResident);
            loop_invariant (\forall int i; BnR_first ≤ i  && i < BnR_end;  BnR[i].estBoursier && !BnR[i].estResident);
            loop_invariant (\forall int i; nBR_first ≤ i  && i < nBR_end;  !nBR[i].estBoursier && nBR[i].estResident);
            loop_invariant (\forall int i; nBnR_first ≤ i && i < nBnR_end; !nBnR[i].estBoursier && !nBnR[i].estResident);

            loop_invariant (\forall int i; 0 ≤ i && i < nbAppeles; ordreAppel[i] != null);

            loop_modifies nbAppeles, ordreAppel[*];
            loop_modifies BR_first, BnR_first, nBR_first, nBnR_first;
          @*/
        for (int nbAppeles = 0; nbAppeles < voeuxClasses.length; nbAppeles++) {
```

```
145             final int nbBoursierRestants = (BR_end-BR_first) + (BnR_end-BnR_first);
146             final int nbResidentsRestants = (BR_end-BR_first) + (nBR_end-nBR_first);
147             final int nbBoursiersAppeles = nbBoursiersTotal - nbBoursierRestants;
148             final int nbResidentsAppeles = nbResidentsTotal - nbResidentsRestants;
149
150             /* on calcule lequel ou lesquels des critères boursiers et candidats du secteur
151                 contraignent le choix du prochain candidat dans l'ordre d α ppel */
152             boolean contrainteTauxBoursier
153                     = (nbBoursiersAppeles < nbBoursiersTotal)
154                     && (nbBoursiersAppeles * 100 < tauxMinBoursiersPourcents * (1 + nbAppeles));
155
156             //@ assert contrainteTauxBoursier ==> 0 < BR_end-BR_first || 0 < BnR_end-BnR_first;
157
158             boolean contrainteTauxResident
159                     = (nbResidentsAppeles < nbResidentsTotal)
160                     && (nbResidentsAppeles * 100 < tauxMinDuSecteurPourcents * (1 + nbAppeles));
161
162             //@ assert contrainteTauxResident ==> 0 < BR_end-BR_first || 0 < nBR_end-nBR_first;
163
164             /*@ nullable @*/ VoeuClasse meilleur = null;
165             if (0 < BR_end-BR_first) {
166                 meilleur = BR[BR_first];
167             }
168             if (0 < nBR_end-nBR_first && !contrainteTauxBoursier && (meilleur == null || meilleur.rang < nBR[nBR_first].rang)) {
169                 meilleur = nBR[nBR_first];
170             }
171             if (0 < BnR_end-BnR_first && !contrainteTauxResident && (meilleur == null || meilleur.rang < BnR[BnR_first].rang)) {
172                 meilleur = BnR[BnR_first];
173             }
174             if (0 < nBnR_end-nBnR_first && !contrainteTauxBoursier && !contrainteTauxResident && (meilleur == null || meilleur.rang <
                      nBnR[nBnR_first].rang)) {
175                 meilleur = nBnR[nBnR_first];
176             }
177             if (meilleur == null) {
178                 //@ assert 0 < BnR_end-BnR_first;
179                 meilleur = BnR[BnR_first];
180             }
181             /*@ assert meilleur != null; @*/
182
183             /*@ assert (0 < BR_end-BR_first     && meilleur == BR[BR_first]) ||
184                       (0 < BnR_end-BnR_first   && meilleur == BnR[BnR_first]) ||
185                       (0 < nBR_end-nBR_first   && meilleur == nBR[nBR_first]) ||
186                       (0 < nBnR_end-nBnR_first && meilleur == nBnR[nBnR_first]); @*/
187
188             /* suppression du candidat choisi de sa file d α ttente */
189              if (meilleur.estBoursier) {
190                 if (meilleur.estResident) {
191                     //@ assert 0 < BR_end-BR_first;
192                     BR_first++;
193                 } else {
194                     //@ assert 0 < BnR_end-BnR_first;
195                     BnR_first++;
196                 }
197             } else {
198                 if (meilleur.estResident) {
199                     //@ assert 0 < nBR_end-nBR_first;
200                     nBR_first++;
201                 } else {
202                     //@ assert 0 < nBnR_end-nBnR_first;
203                     nBnR_first++;
204                 }
205             }
206
207             /* ajout du meilleur candidat à l'ordre d α ppel*/
208             ordreAppel[nbAppeles] = meilleur;
209         }
210
211         /* retourne les candidats classés dans l'ordre d α ppel */
212         return ordreAppel;
213     }
214 }
215
216 public class GroupeClassement__everything_arrays {}
```

3.2.4 Permutation implementation

File `jml2why3/examples/GroupeClassement__permutation.java`.

```
import java.util.Arrays;
import java.util.Map;
import java.util.HashMap;
import java.util.Queue;
import java.util.PriorityQueue;
import java.util.LinkedList;

class VoeuClasse {

    public final boolean boursier;
    public final boolean resident;
    public final int rang; // Index in list of wishes
    // public int rangAppel; // Value in Permutation.inv

    /*@ public invariant 1 ≤ rang; @*/

    /*@ requires 1 ≤ rang; @*/
    /*@ pure @*/ public VoeuClasse(
            final int rang,
            final boolean boursier,
            final boolean resident) {
        this.rang = rang;
        this.boursier = boursier;
        this.resident = resident;
    }
}

class Permutation {
    public final int[] map;
    public final int[] inv; // TODO ghostify

    /*@ // public invariant map != inv;
        public invariant map.length == inv.length;
        public invariant (\forall int i; 0 ≤ i && i < map.length;
          0 ≤ map[i] && map[i] < map.length && inv[map[i]] == i);
        public invariant (\forall int i; 0 ≤ i && i < map.length;
          0 ≤ inv[i] && inv[i] < map.length && map[inv[i]] == i); @*/

    // // Errors with
    // // [@vc:sp]
    // // ((Utils.ignore (this.map ← map0));
    // // (Utils.ignore (this.inv ← inv0)))
    // // This expression prohibits further usage of the variable map0 or any function that depends on it
    // /*@ requires map0.length == inv0.length;
    //     requires (\forall int j; 0 ≤ j && j < map0.length; 0 ≤ map0[j] && map0[j] < map0.length);
    //     requires (\forall int j; 0 ≤ j && j < map0.length; inv0[map0[j]] == j);
    //     requires (\forall int j; 0 ≤ j && j < map0.length; 0 ≤ inv0[j] && inv0[j] < map0.length);
    //     requires (\forall int j; 0 ≤ j && j < map0.length; map0[inv0[j]] == j);
    //     ensures map.length == map0.length;
    //     ensures (\forall int j; 0 ≤ j && j < map.length; map[j] == map0[j]);
    //     ensures (\forall int j; 0 ≤ j && j < map.length; inv[j] == inv0[j]); @*/
    // Permutation(final int[] map0, final int[] inv0) {
    //     this.map = map0;
    //     this.inv = inv0;
    // }


    /*@ requires map0.length == inv0.length;
        requires (\forall int j; 0 ≤ j && j < map0.length;
          0 ≤ map0[j] && map0[j] < map0.length && inv0[map0[j]] == j);
        requires (\forall int j; 0 ≤ j && j < map0.length;
          0 ≤ inv0[j] && inv0[j] < map0.length && map0[inv0[j]] == j);
        ensures this.map.length == map0.length;
        ensures this.map.length == inv.length;
        ensures (\forall int j; 0 ≤ j && j < map.length; this.map[j] == map0[j]);
        ensures (\forall int j; 0 ≤ j && j < map.length; this.inv[j] == inv0[j]); @*/
    Permutation(final int[] map0, final int[] inv0) {
        this.map = map0.clone();
        this.inv = inv0.clone();
    }
}

```

```
class GroupeClassement {

    /*le code identifiant le groupe de classement dans la base de données
        Remarque: un même groupe de classement peut être commun à plusieurs formations
     */
    public final int C_GP_COD;

    /* le taux minimum de boursiers dans ce groupe d α ppel
        (nombre min de boursiers pour 100 candidats) */
    public final int tauxBoursiers;

    /* le taux minimum de candidats du secteur dans ce groupe d α ppel
        (nombre min de candidats du secteur pour 100 candidats) */
    public final int tauxResidents;

    /*@ invariant 0 ≤ tauxBoursiers && tauxBoursiers ≤ 100;
        invariant 0 ≤ tauxResidents && tauxResidents ≤ 100; @*/

    /*@ requires 0 ≤ tauxBoursiers && tauxBoursiers ≤ 100;
        requires 0 ≤ tauxMinResidentsPourcents && tauxMinResidentsPourcents ≤ 100; @*/
    public /*@ pure @*/ GroupeClassement(
            final int C_GP_COD,
            final int tauxBoursiers,
            final int tauxMinResidentsPourcents) {
        this.C_GP_COD = C_GP_COD;
        this.tauxBoursiers = tauxBoursiers;
        this.tauxResidents = tauxMinResidentsPourcents;
    }

    /*@ ensures \result == (\forall int i; 0 ≤ i && i < q.size(); 0 ≤ q._get(i) && q._get(i) < vs.size());
        public static pure model boolean inRange(final LinkedList<VoeuClasse> vs, final Queue<Integer> q);

        ensures \result == (\forall int i; 0 ≤ i && i < q.size();   vs.get(q._get(i)).boursier == boursier && vs.get(q._get(i)).
              resident == resident);
        public static pure model boolean homogenic(final LinkedList<VoeuClasse> vs, final Queue<Integer> q, final boolean boursier,
              final boolean resident);

        ensures \result == (\num_of int j; l ≤ j && j < u; a[j] == x);
        public static pure model int occArray(final int x, final int[] a, final int l, final int u);

        ensures \result == (\num_of int j; 0 ≤ j && j < Q.size(); Q._get(j) == x);
        public static pure model int occQueue(final int x, final Queue<Integer> Q); @*/

    /*@ // Propriété 4.1 b: Un candidat résident boursier qui a le rang r dans le classement
        // pédagogique aura donc un rang inférieur ou égal à r dans l'ordre d α ppel.
        requires 0 ≤ n && n ≤ vs.size();
        requires vs.size() == inv.length;
        ensures \result == (\forall int j; 0 ≤ j && j < n; vs.get(j).boursier && vs.get(j).resident ==>
            // j ≜ index in voeuxClasses ≜ rang; inv[j] ≜ rangAppel
          inv[j] ≤ j);
        public static pure model boolean property41b(final LinkedList<VoeuClasse> vs, final int[] inv, final int n); @*/

    /*@ requires 0 ≤ n && n ≤ vs.size();
        requires vs.size() == map.length;
        ensures \result == (\forall int j; 0 ≤ j && j < n; vs.get(map[j]).boursier && vs.get(map[j]).resident ==>
            // j ≜ rangAppel; map[j] ≜ index in voeuxClasses ≜ rang
            j ≤ map[j]);
        public static pure model boolean property41a(final LinkedList<VoeuClasse> vs, final int[] map, final int n); @*/

    /*@ requires vs.size() * 100 < Integer.MAX_VALUE; // strictly smaller to enable 1-indexed rang, rangAppel
        requires_redundantly vs.size() < 21_474_836;
        requires (\forall int j; 0 ≤ j && j < vs.size(); vs.get(j) != null && \invariant_for(vs.get(j)));
        requires (\forall int i1, i2; 0 ≤ i1 && i1 < vs.size() && 0 ≤ i2 && i2 < vs.size() && i1 < i2; vs.get(i1).rang < vs.get(i2).
              rang);
        ensures \result.map.length == vs.size();
        ensures property41b(vs, \result.inv, vs.size()); @*/
    public Permutation calculerOrdreAppel(final LinkedList<VoeuClasse> vs) {

        /*@ // We can always refer to indices instead of rangs:
            assert (\forall int i, j; 0 ≤ i && i < vs.size() && 0 ≤ j && j < vs.size();
                i < j ≤=> vs.get(i).rang < vs.get(j).rang); @*/

        final Queue<Integer> qBR = new LinkedList≠(), qBnR = new LinkedList≠(), qnBR = new LinkedList≠(), qnBnR = new LinkedList≠
              ();

        /*@ loop_invariant 0 ≤ i && i ≤ vs.size();
```

```
            loop_invariant \invariant_for(qBR) && \invariant_for(qBnR) && \invariant_for(qnBR) && \invariant_for(qnBnR);
            loop_invariant (\forall int j; 0 ≤ j && j < vs.size(); \invariant_for(vs.get(j)));
            loop_invariant (\forall int j; 0 ≤ j && j < vs.size(); vs.get(j) != null);

            loop_invariant inRange(vs, qBR) && inRange(vs, qBnR) && inRange(vs, qnBR) && inRange(vs, qnBnR);
            loop_invariant homogenic(vs, qBR, true, true) && homogenic(vs, qBnR, true, false) && homogenic(vs, qnBR, false, true) &&
                  homogenic(vs, qnBnR, false, false);

            loop_invariant qBR.size() ≤ vs.size() && qBnR.size() ≤ vs.size() && qnBR.size() ≤ vs.size() && qnBnR.size() ≤ vs.size
                  ();
            loop_invariant qBR.size() + qBnR.size() + qnBR.size() + qnBnR.size() == i;

            loop_invariant (\forall int j; 0 ≤ j && j < i; occQueue(j, qBR) + occQueue(j, qBnR) + occQueue(j, qnBR) + occQueue(j,
                  qnBnR) == 1);
            loop_invariant (\forall int j; i ≤ j && j < vs.size(); occQueue(j, qBR) + occQueue(j, qBnR) + occQueue(j, qnBR) +
                  occQueue(j, qnBnR) == 0);

            loop_modifies qBR.content, qBnR.content, qnBR.content, qnBnR.content; @*/
        for (int i = 0; i < vs.size(); i++) {
            final VoeuClasse voe = vs.get(i);
            if (voe.boursier) {
                if (voe.resident)
                    qBR.add(i);
                else
                    qBnR.add(i);
            } else {
                if (voe.resident)
                    qnBR.add(i);
                else
                    qnBnR.add(i);
            }
        }

        final int nbBoursiersTotal = qBR.size() + qBnR.size();
        final int nbResidentsTotal = qBR.size() + qnBR.size();

        final int[] map = new int[vs.size()];
        final int[] inv = new int[vs.size()];

        /*@ loop_invariant 0 ≤ i && i ≤ vs.size();
            loop_invariant (\forall int j; 0 ≤ j && j < vs.size(); vs.get(j) != null);
            loop_invariant \invariant_for(map);
            loop_invariant \invariant_for(inv);
            loop_invariant map.length == vs.size() && inv.length == vs.size();
            loop_invariant \invariant_for(qBR) && \invariant_for(qBnR) && \invariant_for(qnBR) && \invariant_for(qnBnR);

            loop_invariant inRange(vs, qBR) && inRange(vs, qBnR) && inRange(vs, qnBR) && inRange(vs, qnBnR);
            loop_invariant homogenic(vs, qBR, true, true) && homogenic(vs, qBnR, true, false) && homogenic(vs, qnBR, false, true) &&
                  homogenic(vs, qnBnR, false, false);

            loop_invariant qBR.size() ≤ vs.size() && qBnR.size() ≤ vs.size() && qnBR.size() ≤ vs.size() && qnBnR.size() ≤ vs.size
                  ();

            loop_invariant 0 ≤ qBR.size() + qBnR.size() && qBR.size() + qBnR.size() ≤ nbBoursiersTotal;
            loop_invariant 0 ≤ qBR.size() + qnBR.size() && qBR.size() + qnBR.size() ≤ nbResidentsTotal;

            loop_invariant i + qBR.size() + qBnR.size() + qnBR.size() + qnBnR.size() == vs.size();

            loop_invariant // index_in_array_or_queue
              (\forall int j; 0 ≤ j && j < vs.size();
                (occArray(j, map, 0, i) == 0 && occQueue(j, qBR) + occQueue(j, qBnR) + occQueue(j, qnBR) + occQueue(j, qnBnR) == 1) ||
                (occArray(j, map, 0, i) == 1 && occQueue(j, qBR) + occQueue(j, qBnR) + occQueue(j, qnBR) + occQueue(j, qnBnR) == 0));

            loop_invariant (\forall int j; 0 ≤ j && j < i;
              0 ≤ map[j] && map[j] < map.length);
            loop_invariant (\forall int j; 0 ≤ j && j < i;
              inv[map[j]] == j);

            loop_invariant (\forall int j; 0 ≤ j && j < vs.size();
                occQueue(j, qBR) + occQueue(j, qBnR) + occQueue(j, qnBR) + occQueue(j, qnBnR) == 0 ==>
                (0 ≤ inv[j] && inv[j] < i));
            loop_invariant (\forall int j; 0 ≤ j && j < vs.size();
                occQueue(j, qBR) + occQueue(j, qBnR) + occQueue(j, qnBR) + occQueue(j, qnBnR) == 0 ==>
                map[inv[j]] == j);

            loop_invariant // almost property41b
```

```
            (\forall int j; 0 ≤ j && j < vs.size();
              occQueue(j, qBR) + occQueue(j, qBnR) + occQueue(j, qnBR) + occQueue(j, qnBnR) == 0 ==>
              vs.get(j).boursier && vs.get(j).resident ==>
              inv[j] ≤ j);
          // loop_invariant property41a(vs, map, i);

          loop_modifies map[*], inv[*];
          loop_modifies qBR.content, qBnR.content, qnBR.content, qnBnR.content; @*/
      for (int i = 0; i < vs.size(); i++) {

          final int boursierRestants = qBR.size() + qBnR.size();
          final int residentsRestants = qBR.size() + qnBR.size();
          final int boursiersAppeles = nbBoursiersTotal - boursierRestants;
          final int residentsAppeles = nbResidentsTotal - residentsRestants;

          final boolean contrainteBoursier = 0 < boursierRestants
              && (boursiersAppeles * 100 < tauxBoursiers * (1 + i));

          /*@ assert contrainteBoursier ==> 0 < qBR.size() || 0 < qBnR.size(); @*/

          final boolean contrainteResident = 0 < residentsRestants
              && (residentsAppeles * 100 < tauxResidents * (1 + i));

          /*@ assert contrainteResident ==> 0 < qBR.size() || 0 < qnBR.size(); @*/

          /*@ nullable @*/ Integer m = null;
          if (!qBR.isEmpty()) {
              /*@ // The assertion is relevant for the non-nullablity of the result of Queue.peek() below and for the condition in
                    the other cases.
                  assert 0 ≤ qBR._get(0) && qBR._get(0) < vs.size(); @*/
              m = qBR.peek();
          }
          if (!qnBR.isEmpty()) {
              /*@ assert 0 ≤ qnBR._get(0) && qnBR._get(0) < vs.size(); @*/
              if (!contrainteBoursier && (m == null || m < vs.get(qnBR.peek()).rang))
                  m = qnBR.peek();
          }
          if (!qBnR.isEmpty()) {
              /*@ assert 0 ≤ qBnR._get(0) && qBnR._get(0) < vs.size(); @*/
              if (!contrainteResident && (m == null || m < vs.get(qBnR.peek()).rang))
                  m = qBnR.peek();
          }
          if (!qnBnR.isEmpty()) {
              /*@ assert 0 ≤ qnBnR._get(0) && qnBnR._get(0) < vs.size(); @*/
              if (!contrainteBoursier && !contrainteResident && (m == null || m < vs.get(qnBnR.peek()).rang)) {
                  m = qnBnR.peek();
              }
          }
          if (m == null) {
              /*@ assert 0 < qBnR.size();
                  assert 0 ≤ qBnR._get(0) && qBnR._get(0) < vs.size();  @*/
              m = qBnR.peek();
          }

          final int m1 = m;
          /*@ assert 0 ≤ m1 && m1 < vs.size();
              assert // some queue
                (0 < qBR.size() && m == qBR._get(0)) ||
                (0 < qBnR.size() && m == qBnR._get(0)) ||
                (0 < qnBR.size() && m == qnBR._get(0)) ||
                (0 < qnBnR.size() && m == qnBnR._get(0));
                assert 0 < qBR.size()   && m == qBR._get(0)   ==> vs.get(m).boursier && vs.get(m).resident;
                assert 0 < qBnR.size()  && m == qBnR._get(0)  ==> vs.get(m).boursier && !vs.get(m).resident;
                assert 0 < qnBR.size()  && m == qnBR._get(0)  ==> !vs.get(m).boursier && vs.get(m).resident;
                assert 0 < qnBnR.size() && m == qnBnR._get(0) ==> !vs.get(m).boursier && !vs.get(m).resident;
              assert occQueue(m1, qBR) + occQueue(m1, qBnR) + occQueue(m1, qnBR) + occQueue(m1, qnBnR) == 1; @*/
          x: {
              if (vs.get(m1).boursier) {
                  if (vs.get(m1).resident) {
                      final int m2 = qBR.poll();
                  } else {
                      final int m2 = qBnR.poll();
                  }
              } else {
                  if (vs.get(m1).resident) {
                      final int m2 = qnBR.poll();
```

```
                    } else {
                        final int m2 = qnBnR.poll();
                    }
                }
                /*@ assert // shifted_queue
                      (0 < \old(qBR, x).size()   && m == \old(qBR, x)._get(0)   && \old(qBR, x).size() == qBR.size() + 1     && (\
                            forall int j; 0 ≤ j && j < qBR.size();   qBR._get(j)   == \old(qBR, x)._get(j+1)))  ||
                      (0 < \old(qBnR, x).size()  && m == \old(qBnR, x)._get(0)  && \old(qBnR, x).size() == qBnR.size() + 1   && (\
                            forall int j; 0 ≤ j && j < qBnR.size();  qBnR._get(j)  == \old(qBnR, x)._get(j+1))) ||
                      (0 < \old(qnBR, x).size()  && m == \old(qnBR, x)._get(0)  && \old(qnBR, x).size() == qnBR.size() + 1   && (\
                            forall int j; 0 ≤ j && j < qnBR.size();  qnBR._get(j)  == \old(qnBR, x)._get(j+1))) ||
                      (0 < \old(qnBnR, x).size() && m == \old(qnBnR, x)._get(0) && \old(qnBnR, x).size() == qnBnR.size() + 1 && (\
                            forall int j; 0 ≤ j && j < qnBnR.size(); qnBnR._get(j) == \old(qnBnR, x)._get(j+1)));
                    assert occArray(m1, map, 0, i) == 0;
                    assert (\forall int j; 0 ≤ j && j < i; map[j] != m1); @*/
                map[i] = m1;
                inv[m1] = i;
                /*@ assert occQueue(m1, qBR) + occQueue(m1, qBnR) + occQueue(m1, qnBR) + occQueue(m1, qnBnR) == 0;
                    assert (\forall int j; 0 ≤ j && j < vs.size() && j != i; map[j] == \old(map, x)[j]);
                    assert (\forall int j; 0 ≤ j && j < i; inv[map[j]] == \old(inv, x)[\old(map, x)[j]]); @*/
            }
        }

        return new Permutation(map, inv);
    }
}

public class GroupeClassement__permutation {}
```

3.2.5 Rust implementation

File `rust/src/main.rs`.

```
use std::collections::VecDeque;
extern crate prusti_contracts;

#[derive(Debug,PartialEq)]
struct VoeuClasse {
    rang: i32,
    est_boursier: bool,
    est_resident: bool,
    rang_appel: i32,
}

impl VoeuClasse {
    fn new(rang: i32, est_boursier: bool, est_resident: bool) → VoeuClasse {
        VoeuClasse { rang, est_boursier, est_resident, rang_appel: 0 }
    }
}

#[pure]
fn compare(voeux_classes: Vec<VoeuClasse>, v: VoeuClasse) → bool {
    v == voeux_classes[v.rang as usize - 1]
}

#[requires="forall i: usize :: (0 ≤ i && i < voeux_classes.len()) ==> voeux_classes[i].rang == (i+1) as i32"]
#[requires="0 ≤ taux_min_boursiers_pourcents && taux_min_boursiers_pourcents ≤ 100"]
#[requires="0 ≤ taux_min_residents_pourcents && taux_min_residents_pourcents ≤ 100"]
#[ensures="forall i: usize :: (0 ≤ i && i < result.len()) ==> result[i].rang == (i+1) as i32"]
#[ensures="after_expiry<result>(voeux_classes.len() == before_expiry(result).len())"]
#[ensures="after_expiry<result>(forall i: usize :: (0 ≤ i && i < voeux_classes.len()) ==>
    before_expiry(result)[i] == voeux_classes[before_expiry(result)[i].rang as usize - 1])"]
#[ensures="after_expiry<result>(forall i: usize :: (0 ≤ i && i < voeux_classes.len()) ==>
    voeux_classes[i] == before_expiry(result)[(voeux_classes[i].rang_appel-1) as usize])"]
//                   ^^ TODO How to test for physical equality?
fn groupe_classement(
    voeux_classes: Vec<VoeuClasse>,
    taux_min_boursiers_pourcents: i32,
    taux_min_residents_pourcents: i32,
) → Vec<VoeuClasse> {
    let n = voeux_classes.len();
    let mut br: VecDeque<VoeuClasse> = VecDeque::new();
    let mut bnr: VecDeque<VoeuClasse> = VecDeque::new();
    let mut nbr: VecDeque<VoeuClasse> = VecDeque::new();
    let mut nbnr: VecDeque<VoeuClasse> = VecDeque::new();
    for v in voeux_classes {
        if v.est_boursier {
            if v.est_resident {
                br.push_back(v)
            } else {
                bnr.push_back(v)
            }
        } else {
            if v.est_resident {
                nbr.push_back(v)
            } else {
                nbnr.push_back(v)
            }
        }
    }

    let nb_boursiers_total = br.len() as i32 + bnr.len() as i32;
    let nb_residents_total = br.len() as i32 + nbr.len() as i32;

    let mut ordre_appel : Vec<VoeuClasse> = Vec::new();
    for nb_appeles in 0..n {

        let nb_appeles = nb_appeles as i32;

        let nb_boursiers_restants = br.len() as i32 + bnr.len() as i32;
        let nb_residents_restants = br.len() as i32 + nbr.len() as i32;
        let nb_boursiers_appeles = nb_boursiers_total - nb_boursiers_restants;
        let nb_residents_appeles = nb_residents_total - nb_residents_restants;

        let contrainte_taux_boursiers = 0 < nb_boursiers_restants
```

```
            && (nb_boursiers_appeles * 100 < taux_min_boursiers_pourcents * (1 + nb_appeles));
        let contrainte_taux_residents = 0 < nb_residents_restants
            && (nb_residents_appeles * 100 < taux_min_residents_pourcents * (1 + nb_appeles));

        let mut meilleur : Option<&VoeuClasse> = Option::None;
        if !br.is_empty() {
            meilleur = Some(br.front().expect("br non-empty")); // Or: meilleur = br.front();
        }
        if !nbr.is_empty() && !contrainte_taux_boursiers && meilleur.map_or(true, |m| nbr.front().expect("nbr non-empty").rang < m.rang) {
            meilleur = Some(nbr.front().expect("nbr non-empty"));
        }
        if !bnr.is_empty() && !contrainte_taux_residents && meilleur.map_or(true, |m| bnr.front().expect("bnr non-empty").rang < m.rang) {
            meilleur = Some(bnr.front().expect("bnr non-empty"));
        }
        if !nbnr.is_empty() && !contrainte_taux_boursiers && !contrainte_taux_residents && meilleur.map_or(true, |m| nbnr.front().expect("nbnr non-empty").rang < m.rang) {
            meilleur = Some(nbnr.front().expect("nbnr non-empty"));
        }
        if meilleur.is_none() {
            meilleur = Some(bnr.front().expect("bnr is non-empty"));
        }

        let meilleur = {
            let meilleur = meilleur.expect("meilleur is not none");
            if meilleur.est_boursier {
                if meilleur.est_resident {
                    br.pop_front()
                } else {
                    bnr.pop_front()
                }
            } else {
                if meilleur.est_resident {
                    nbr.pop_front()
                } else {
                    nbnr.pop_front()
                }
            }
        };
        ordre_appel.push(meilleur.expect("not none"));
    }
    for i in 0..n {
        ordre_appel[i].rang_appel = i as i32 + 1;
    }
    ordre_appel
}

fn main() {
    let mut voeux_classes = Vec::new();
    voeux_classes.push(VoeuClasse::new(1, true, true));
    voeux_classes.push(VoeuClasse::new(2, true, false));
    voeux_classes.push(VoeuClasse::new(3, false, true));
    voeux_classes.push(VoeuClasse::new(4, false, false));
    voeux_classes.push(VoeuClasse::new(5, true, true));

    println!("appel - rang - boursier - resident");
    println!();
    println!("Voeux classes");
    for i in 0..voeux_classes.len() {
        let v = &voeux_classes[i];
        println!("{} - {} - {} - {}", v.rang_appel, v.rang, v.est_boursier, v.est_resident);
    }

    let ordre_appel = groupe_classement(voeux_classes, 1, 1);

    println!();
    println!("Ordre appel");
    for i in 0..ordre_appel.len() {
        let v = &ordre_appel[i];
        println!("{} - {} - {} - {}", v.rang_appel, v.rang, v.est_boursier, v.est_resident);
    }
}
```