Devoir de *Model-checking*

Master d'Informatique 2021–22, Université de Rouen

26 octobre 2021

Résumé

Ce devoir est strictement individuel. Il est à rendre avec un rapport et le code source. On y étudie une version particulière du tri rapide.

1 Vue générale

Le *tri rapide* (*quicksort*) est un algorithme bien connu permettant de trier un tableau de N éléments. Il est simple et efficace en moyenne, même si dans certains cas il peut ne pas être plus efficace qu'un algorithme de tri naïf (réalisant $O(N^2)$ permutations d'éléments pendant le tri).

Le but du devoir est de vérifier, en utilisant SPIN, que le *quicksort* réalise bien ce qui lui est demandé, c'est-à-dire, trier un tableau. Nous nous intéressons plus particulièrement à l'algorithme décrit dans `https://en.wikipedia.org/wiki/Quicksort`, Section « *Lomuto partition scheme* », qui utilise une méthode particulière pour le partitionnement du tableau, bien connue car pédagogique, mais qui n'est pas la méthode de partitionnement originelle :

```
// Sorts a (portion of an) array, divides it into partitions, then sorts those
algorithm quicksort(A, lo, hi) is
  // If indices are in correct order
  if lo >= 0 && hi >= 0 && lo < hi then
    // Partition array and get pivot index
    p := partition(A, lo, hi)

    // Sort the two partitions
    quicksort(A, lo, p - 1) // Left side of pivot
    quicksort(A, p + 1, hi) // Right side of pivot

// Divides array into two partitions
algorithm partition(A, lo, hi) is
  pivot := A[hi] // The pivot must be the last element

  // Pivot index
  i := lo - 1

  for j := lo to hi do
    // If the current element is less than or equal to the pivot
    if A[j] <= pivot then
      // Move the pivot index forward
```

```
    i := i + 1

    // Swap the current element with the element at the pivot
    swap A[i] with A[j]
return i // the pivot index
```

En utilisant ce modèle, vous montrerez :

— que la fonction de partitionnement fait bien ce qu'elle est supposée faire, c'est-à-dire qu'à sa sortie, les éléments à gauche de la position du pivot sont bien tous inférieurs ou égaux au pivot, et ceux à droite bien supérieurs ou égaux ;

— qu'à la sortie du tout dernier appel de la fonction de tri, le tableau est bien trié dans sa totalité.

2 Détails implantatoires

Vous travaillerez sur des tableaux de N entiers, variants entre 1 et N, pouvants éventuellement contenir plusieurs fois le même élément. Pour limiter l'explosion combinatoire du *model-checking*, vous prendrez N petit et choisirez les types Promela adaptés.

Promela dispose des types d'entiers suivants :

— `bit` : $0, 1$

— `bool` : false,true

— `byte` : $0 \dots 255$

— `mtype` : $1 \dots 255$

— `short` : $-2^{15} \dots 2^{15} - 1$

— `int` : $-2^{31} \dots 2^{31} - 1$

— `unsigned` : $0 \dots 2^{32} - 1$

Les variables de type `unsigned` peuvent être stockées sur un nombre de bits choisi, et bien sûr cela impacte le nombre de valeurs représentables :

```
unsigned v : 5;     /* unsigned stored in 5 bits      */
unsigned w : 3 = 5; /* value range 0..7, initially 5  */
```

Vous utiliserez cette possibilité. Elle n'est malheureusement pas disponible pour les éléments d'un tableau, mais vous pourrez l'utiliser pour les variables contenant des indices pour les éléments du tableau. Elle ne permet pas de diminuer le nombre d'états ou de transitions du modèle, mais de diminuer (un peu) l'espace mémoire utilisé par SPIN pour représenter chacun des états.

En global dans le modèle Promela, vous pourrez par exemple avoir

```
#define ELEMENT_TYPE byte // Type des éléments du tableau
#define BITS_POS 3        // Nombre de bits à utiliser pour les variables
                          //     représentant des indices du tableau.
                          //     Doit être cohérent avec N
#define N 7               // Nombre d'éléments du tableau
                          //     A adapter en fonction de la puissance de votre machine
ELEMENT_TYPE a[N];        // Le tableau à trier
```

L'exemple ci-dessous montre comment implanter des appels de fonctions en utilisant des patrons de processus Promela. Chaque appelant attends la terminaison de l'appelé avant de continuer.

```
// Un exemple d'implantation de fonction

proctype fact(chan c; int n) {
  int r=1;
  do
  :: n==0 -> break;
  :: n>0  -> r=r*n; n=n-1;
  od
  c!r;
}

active proctype main() {
  chan intFuncall = [0] of {int};
  int r;
  run fact(intFuncall,5);
  intFuncall?r;
  printf("r=%d\n", r);
}
```

Votre travail devra contenir au moins 2 patrons et une section d'initialisation :

- un patron correspondant à la fonction `quicksort`, et prenant en entrée au moins les indices de début et de fin de la portion du tableau à trier. Comme nous supposons le tableau global, il n'est pas nécessaire de le mettre en argument ;
- un patron correspondant à la fonction `partition`, et prenant en entrée au moins les indices de début et de fin de la portion du tableau à trier. Comme nous supposons le tableau global, il n'est pas nécessaire de le mettre en argument ;
- une section d'initialisation du système, qui initialise aléatoirement le tableau puis le trie en utilisant `quicksort`.

Vous paramétrerez votre code source avec 2 macros :

```
// Pour vérifier seulement la fonction de partionnement
// Ne doit pas être activé en même temps que VERIFY_SORT
#define VERIFY_PARTITION

// Pour vérifier que le tri fonctionne
// Ne doit pas être activé en même temps que VERIFY_PARTITION
#define VERIFY_SORT
```

Si la première est activée, le tri ne nous intéresse pas : ce qui nous intéresse, c'est uniquement vérifier que la fonction de partitionnement fait son travail. Si la seconde est activée, on ne vérifie pas le partitionnement : ce qui nous intéresse, c'est uniquement vérifier que la fonction de tri fait son travail. Dans les deux cas, vous pourrez par exemple désactiver les éventuelles parties de code inutiles en utilisant les macros.

Il est possible que l'automate résultant du modèle ait une taille importante, et qu'une machine puissante soit nécessaire. Dans ce cas, laissez éventuellement *swapper* un peu votre machine, mais pas plus de quelques dizaines de minutes, et en surveillant la quantité totale de mémoire virtuelle restant disponible afin de ne pas mettre votre système en péril pour cause de mémoire insuffisante.

3 Livrable

Le code source devra être soigneusement présenté et commenté.

Votre rapport contiendra :

— un rappel du problème ;

— une description de votre implantation du modèle. Vous justifierez vos choix ;

— ce que vous avez vérifié, comment, vos difficultés, les résultats, vos conclusions. Vous pouvez avoir vérifié en utilisant plusieurs méthodes ; dans ce cas, vous les décrirez et les comparerez.

Votre travail devra être développé *individuellement*, et rendu au plus tard le *19 décembre 2021* au soir, dans une archive au format `tar` gzippé de nom `FrancoisDupont.tar.gz` si Francois Dupont est votre nom, envoyée en pièce jointe à un courriel de sujet « Devoir de Model-checking » à l'adresse `Nicolas.Bedon@univ-rouen.fr`. Vous vous mettrez en copie du courriel pour vérifier que vous n'oubliez pas la pièce jointe. L'extraction du fichier d'archive devra produire un répertoire de nom `FrancoisDupont` contenant le code source de votre travail, votre rapport et tout ce que vous jugerez nécessaire.

4 Question subsidiaire

Elle demande un peu de travail et n'est pas du tout obligatoire.

On remarque que les tris récursifs sur les parties à gauche et à droite du pivot sont indépendants. Alors, pourquoi par exemple attendre que la partie à gauche soit triée avant de commencer à trier celle de droite ? Modifiez votre code pour permettre la concurrence. Refaites votre vérification SPIN du bon fonctionnement du tri et concluez.