Projet d'introduction à la vérification

Adrien Mollet
Romain Soumard

22 avril 2020

1 Introduction

Le présent document est un rapport écrit à l'issue du projet de fin d'année d'introduction à la vérification. Il a été rédigé par M.Adrien Mollet et M.Romain Soumard à l'université d'Aix-Marseille durant l'année scolaire 2019-2020. Le but de ce projet consistait en la résolution de deux exercices :

- Exercice 1 : Trouver la plus courte solution au problème du berger.
- Exercice 2 : Modélisation du jeu du solitaire à l'aide du langage Promela.

Les sections suivantes présenteront brièvement l'organisation du travail et la résolution de chaque exercice.

2 Organisation du travail

Initialement, le travail devait être partagé. Romain devait se charger de l'exercice 1 tandis qu'Adrien devait se charger de l'exercice 2. Cependant, il est vite apparu au cours de l'exercice 2 que deux personnes ne seraient pas de trop pour résoudre l'ensemble de l'exercice. Les deux exercices ont donc été effectués en commun au fur et à mesure de l'avancée du projet.

3 Exercice 1 : Problème du berger

3.1 Résolution du problème

La résolution du problème du berger s'est faite en deux temps. Il a tout d'abord fallu trouver les caractéristiques de l'exécution que l'on désirait obtenir. Une fois ceci fait, il a fallut l'exprimer à l'aide d'une formule LTL.
Bien que dans un premier temps, nous exprimions la propriété à vérifier à l'aide d'un processus never, il nous est apparu en parcourant la documentation de spin qu'il était bien plus aisé d'exprimer directement la propriété recherchée dans le code. Il s'agit en effet de la méthode recommandée depuis spin 6, celle-ci étant plus propre et équivalente (la formule reste la même, elle n'est juste pas automatiquement niée quand on génère un processus never).
Après avoir trouver la formule appropriée, une longue phase de débugguage s'en est suivi pour comprendre pourquoi nous n'arrivions pas à générer le modèle approprié. Il s'est avéré que cela était dû à l'ordre dans lequel nous modélisions les traversées des différents protagonistes.
En effet, pour vérifier la propriété voulue, il était essentiel que le berger arrive avant ses amis sur la rive opposée dans notre modèle.
Une fois terminée, nous nous sommes aperçus que l'exécution engendrée par notre propriété produisait déjà une solution optimale (facile à vérifier à l'aide d'un graphe.). Nous avons donc généré un graphe de séquence de message (MSC) afin de compléter l'exercice.

3.2 MSC du berger

Vous trouverez ci-joint dans le fichier MSC_berger.txt le MSC du problème du berger.

4 Exercice 2 : Modélisation du solitaire

4.1 Modélisation du problème de base

La modélisation du solitaire européen classique fut une grosse partie de cet exercice : en effet, nous voulions une modélisation propre et modulable afin de pouvoir générer automatiquement, dans un second temps, un programme Promela capable de gérer des plateaux de type et de tailles différentes.

Premier jet

Dans un premier temps, nous sommes partis sur la création d'une matrice pour représenter le plateau (le type matrice a été défini comme tableau de tableau avec un typedef). Chacune des cases de la matrice contient une valeur parmi les suivantes :

- 0 : la case est libre
- 1 : la case est occupée
- 2 : la case est interdite (cette dernière option permet de représenter n'importe quel plateau sous forme de matrice carrée, bien plus simple a gérer)

Nous avons conservé cette idée de matrice à état pour le reste de la modélisation.
Par la suite, et là se dessine un mauvais choix, nous avons programmé un processus pour initialiser le plateau avec les bonnes valeurs pour chaque case. C'est une erreur car cela ne permet pas de modularité au niveau du choix du plateau, mais également car il n'est pas nécessaire de faire appel à un processus pour cela. Nous avons donc décidé d'oublier cette approche pour la suite.
Ensuite, nous avons défini que chaque case serait représentée par un processus individuel. Nous n'avions a ce moment là aucune autre idée de comment gérer les coups possibles : nous avons donc décidé d'initialiser un processus par case vérifiant si un coup est jouable depuis celle-ci, et de placer dans un channel ledit coup. Cette solution fonctionne convenablement lorsqu'il s'agit de juste laisser jouer le solitaire, mais elle crée des problèmes de consommation exponentielle de RAM lorsque l'on utilise pan pour trouver une solution au jeu. En effet, autant de processus et de possibilités font consommer beaucoup trop de mémoire à pan, qui essaye tant bien que mal d'appliquer son algorithme (basé sur le principe du backtracking). Evidemment, cette partie de la modélisation à été remplacée par la suite.
Enfin, une partie de la modélisation que nous avons gardé (mais modifié plus tard) est le processus player. Ce processus attend un coup jouable sur un channel "rendez-vous" (coup envoyé par un des processus de case), puis joue ce coup-là, modifiant le plateau en conséquence. Il attend ensuite un autre coup et continue sur le même principe. Ce processus sera modifié plus tard pour s'adapter à la disparition des processus de case.
Dans l'ensemble, cette première modélisation du problème fut plutôt instinctive et satisfaisante si on ne considère pas de formule LTL à vérifier, dans le sens où le "jeu" se déroule tout de même. Néanmoins, l'utilisation d'un processus par case est une grosse erreur menant à de nombreux soucis dès lors que l'on utilise pan.
Vous pouvez retrouver le code de cette première approche dans le fichier "solitaireV1.pml".

Deuxième, troisième et quatrième jets

Nous avons décidé de se baser sur modèle précédent mais en tentant une autre approche. Nous avons conservé l'idée de matrice, mais nous avons changé le principal problème du modèle précédent : la gestion des cases et des coups jouables. Nous avons migré du principe d'un processus par case, un processus player et un processus board vers une approche à seulement deux processus : board et player. L'idée principale étant de partir cette fois-ci des cases vides afin d'explorer l'ensemble des coups possibles.
Le processus player attend l'arrivée d'une case vide dans un canal de type file d'attente, génère une direction de manière non-déterministe puis l'envoie au processus board et se met en attente jusqu'à la fin du traitement par celui-ci. La fin du traitement est signalée par un canal ready de type rendez-vous.
Le processus board, lui, attend l'arrivée des coordonnées d'une case libre ainsi que d'une direction. Une fois reçue, il applique divers traitements de sécurité et met à jour la matrice du plateau en fonction de la direction qui lui a été communiquée. Si le coup est valide, alors le nombres de pions est décrémenté avant de rendre la main au processus player. Dans le cas contraire, la case vide est remise dans la file d'attente et le nombre de pions n'est pas modifié.
Enfin, le processus init se charge d'initialiser le plateau et de mettre la première case libre dans la file d'attente avant de lancer player et board.
Nous avons regroupé dans cette section les versions 2, 3 et 4 car le modèle est sensiblement le même. La version 4 n'est qu'une réécriture plus propre de la version 2, et souffre des mêmes problèmes (voir plus loin). La version 3 quant à elle est plus particulière. En effet plutôt que de rendre impossible la génération de coups interdits directement dans le modèle, nous avons essayé (sans succès) d'utiliser une formule LTL à cet effet. Les différentes versions sont trouvables respectivements sous les noms "solitaireV2.pml", "solitaireV3.pml" et "solitaireV4.pml".

Problème récurrent

Malgré toutes nos tentatives, nous n'avons hélas pas réussi à nous défaire du principal problème de nos différents modèles : la saturation de la RAM. Il arrive en effet très souvent que pan ne puisse générer de fichier trail en un temps satisfaisant. Nous n'avons donc, au mieux, que réussi à générer des traces décrivant des exécutions non optimales, voir aucune trace du tout.
Ce problème peut avoir, à notre avis, deux causes possibles :

1. Le modèle n'est pas assez contraint par notre formule LTL. En effet, !(¡¿(number_pegs == 1)) n'est peut être pas suffisant à pan pour élaguer l'arbre des possibilités.

2. Notre modèle est trop complexe et comprend trop d'opérations, saturant l'algorithme de calcul de pan.

Après de nombreux essais et de nombreuses modification, aussi bien de nos modèles que de nos formules LTL, nous ne sommes pas parvenus à générer une solution optimale du jeu du solitaire. Nous avons par conséquent décidés de vous proposer un fichier "Proof of Concept", dans lequel pan cherche une exécution pour laquelle le nombre de bille finale est égal à 32, afin de démontrer la viabilité de notre approche. La preuve de concept se trouve dans le répertoire sources/solitaire/proof_of_concept/.

4.2 Gestion d'autres plateaux

Conformément au sujet, nous avons réalisé un script "board_generator.py" permettant de choisir le type de plateau et de générer le programme Promela adéquat. Il est possible de choisir un plateau européen, anglais ou allemand et d'en choisir la taille.
Au vu des problèmes de résolution de notre solitaire dans sa version de base, nous n'avons évidemment pas pu tester la résolution sur des plateaux plus grands. Néanmoins nous avons tout de même tenu à faire le script permettant de faire cette partie du projet.
Nous avons également rédigé deux makefile pour le berger et le solitaire afin de vous faciliter la tâche dans votre correction. Nous vous renvoyons au README pour en comprendre le fonctionnement. Sachez de plus que vous pouvez modifier le fichier pris en compte par le makefile pour le solitaire. il suffit pour cela de changer la variable "file" au début du script.

5 Conclusion

En conclusion, le projet fût très intéressant et instructif. Découvrir la vérification et les éléments essentiels de cette branche de l'informatique fût très enrichissant. Néanmoins, les outils spin et surtout Promela usant d'un paradigme peu familier, nous avons rencontrés quelques difficultés, en particulier vis-à-vis de la partie conception et modélisation, lors de nos travaux (en témoigne nos tentatives infructueuses de modélisation du jeu du solitaire). Malgré les complications occasionnées par le langage et les logiciels utilisés, nous avons donc au final apprécié effectuer ce travail qui constitue nos premiers pas dans le domaine de la vérification.