Model Checking

Auteurs : Bleuer Rémy, Moschard Colin

1. Introduction

Dans ce labo, nous avons choisi de vérifier un buffer concurrent implémenté à l'aide de sémaphores basé sur un des exercices corrigé que Monsieur Cuisenaire nous a fournis (ex15.1_alexandre.h). L'objectif est de tester tous les scénarios (ordres d'exécution possible) entre :

- Un **producteur** (dépose un élément dans le buffer).
- Deux **consommateurs** (récupèrent chacun un élément dans le buffer).

Le buffer est conçu pour autoriser jusqu'à 2 consommateurs simultanés, d'où la gestion de plusieurs sémaphores (`mutex`, `waitFull`, `waitEmpty`).

2. Structure du code

Le code utilise un *framework* qui nous est fourni afin de faire du *model checking*, il permet :

1. D'instrumentaliser le code concurrent à l'aide de *sections* délimitées par :
   - `startSection(id)`
   - `endSection()`
   - `endScenario()`
2. D'observer chaque thread avec un graphe de scénario (`ScenarioGraph`).
3. De combiner tous les graphs dans un `PcoModel` qui utilise un `ScenarioBuilder` pour généré toutes les permutations d'exécution possibles jusqu'à une profondeur donnée (dans notre cas : 9).
4. De lancer le *model checker*, qui exécute chaque scénario, observe sa terminaison (deadend, deadlock, etc.) et affiche un résumé à la fin.

2.1 Balisage des threads

Au début, nous avions qu'une seule section par thread (le `put()` et les 2 `get()`), mais il n'y avait que très peu de scénarios finaux. C'est pourquoi, pour obtenir plus de scénarios, chaque thread est divisé en **3 sections** (Nous ne sommes néanmoins pas certains que cela soit une bonne appoche pour multiplier le nombre de scénarios):

- **Producteur** : sections (1, 2, 3)
  - Section 1 : préparation de l'élément
  - Section 2 : `put()` dans le buffer
  - Section 3 : fin de scénario
- **Consommateur 1** : sections (4, 5, 6)
  - Section 4 : éventuel pré-traitement
  - Section 5 : `get()`
  - Section 6 : fin de scénario
- **Consommateur 2** : sections (7, 8, 9)
  - Section 7 : éventuel pré-traitement
  - Section 8 : `get()`
  - Section 9 : fin de scénario

Chacune de ces sections est délimitée dans le code par `startSection(id)`, `endSection()` et, pour la dernière, `endScenario()`.

2.2 Vérifications

Dans le `PcoModel` nommé `BufferModel` :

1. Nous instancions le buffer partagé (`Buffer2ConsoSemaphore<int>`).

2. Nous créons un thread producteur et deux threads consommateurs en leur passant le buffer.
3. Nous initialisons un `ScenarioBuilderBuffer` avec une profondeur de 9.
    - 3 threads * 3 sections = 9 actions instrumentées possibles.
    - Le builder va générer tous les scénarios où ces 9 sections peuvent s'ordonner

À chaque nouveau scénario, nous exécutons :

- Les 3 threads dans l'ordre imposé par le scénario.
- Chaque thread attend le signal du *model checker* pour entrer dans sa prochaine section.
- À la fin, l'état est noté (si deadlock ou tout s'est bien passé).

3. Résultats et analyse

- 1670 scénarios explorés sur 1680 ($\frac{9!}{3!3!3!} = 1680$, 9 pour la profondeur, et 3 sections par thread).
- Résultats observé :
    - 1116 scénarios se terminent correctement (`AllScenario`).
    - 554 scénarios aboutissent à un `DeadEnd`, signifiant que l'ordonnancement tenté n'est pas réalisable compte tenu du graphe de scénario linéaire de chaque thread. (C'est une fin convenable)
    - Aucun deadlock n'a été détecté, montrant que le code ne s'est jamais retrouvé dans une situation de blocage complet pour tous les fils d'exécution.
    - On arrive à (1116 + 554) 1670 scénarios explorés. Il manque donc 10 scénarios qui ont "disparus", nous ne savons pas ce qu'il se passe concerant ces cas.

Ainsi nous validons que l'implémentation du `Buffer2ConsoSemaphore` se comporte correctement sous toutes les permutations possibles d'ordonnancement.

4. Conclusion

Cette expérience de *model checking* a permis de couvrir l'ensemble des cas de concurrence possibles autour d'un exercice vu en cours. Nous avons pu ainsi démontrer la qualité de la synchronisation, où aucun scénario ne fini avec un deadlock

Ce type de test offre une forte garantie de l'absence de comportements indésirables dans notre code concurrent, confirmant la robustesse du Buffer.