

Laboratoire de Programmation Concurrente

semestre automne 2024

Model checking

Temps à disposition : 4 périodes

Objectifs pédagogiques

- Contrôler l'exécution de plusieurs threads en imposant un ordre d'exécution
- Mettre en place de quoi faire du *Model Checking*

⚠ Pour compiler le code il faut mettre à jour la librairie pcosynchro, en lançant :

```
sudo update_pcosynchro
```

Contexte

Tester un programme multi-threadé n'est pas chose aisée, étant donné que l'ordonnanceur peut imposer des changements de contexte à chaque instant. Il existe toutefois des techniques permettant de réaliser des validations sous certaines conditions. C'est le cas du *model checking*, qui peut être exploité pour vérifier une application multi-threadée.

Avant d'aller plus loin il est recommandé de visionner la vidéo suivante qui présente l'outil TLA+ ¹ (10 minutes) :

<https://youtu.be/jETQwnC1r4Q>

Lors de ce laboratoire nous allons exploiter un mécanisme relativement semblable, permettant de tester différents scénarios via une instrumentalisation du code. Le résultat ne sera pas aussi efficace que ce que vous venez de voir, mais permettra tout de même de réaliser des expériences intéressantes.

Description du framework

Le framework permet d'instrumentaliser un code multi-threadé simple et de tester tous les scénarios d'exécution possibles. Le framework nécessite d'instrumentaliser un code et d'offrir ensuite les méthodes nécessaires à la séquentialisation. Le code instrumentalisé devra être lancé par un `ObservableThread`, et une sous-classe de `PcoModel` contenant divers informations sur le modèle devra être implémentée.

L'instrumentalisation du code repose sur 3 méthodes :

1. `startSection()`
2. `endSection()`
3. `endScenario()`

1. <https://lamport.azurewebsites.net/tla/tla.html>

Chaque appel à `startSection()` doit être suivi d'un appel à une des trois fonctions :

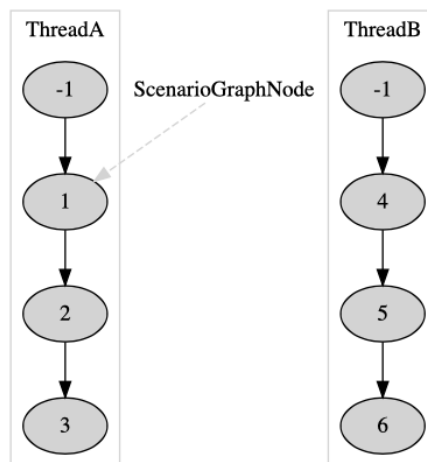
- `startSection(...)` : Démarre une nouvelle section et arrête la section en cours si une existe. Ceci permet de garantir l'atomicité de l'exécution du code de la section.
- `endSection()` : Arrête la section en cours, **ne doit pas** être appelé si un `startSection()` n'a pas été appelé précédemment. Ceci permet que la suite du code jusqu'à atteindre un nouveau `startSection()` ou `endScenario()` puisse s'exécuter d'une manière concurrente.
- `endScenario()` : Tout scénario **doit** se terminer par l'appel à cette méthode. De la même manière que `endSection()` cette méthode ne doit être appelée que si un `startSection()` a été appelé précédemment.

Comme indiquée précédemment la classe `ConcurrencyAnalyzer` permet d'instrumentaliser le code, mais pour ceci elle utilise la classe `ObservableThread`. Celle-ci permet de mettre en place des scénarios dans la méthode `run`. Cette deuxième classe implémente les scénarios, donc le code que nous voulons exécuter ainsi que les sections à instrumentaliser.

Pour avoir un exemple, vous pouvez vous rendre dans le fichier `tests/cablock.h`, voir classe `ThreadBlock` ou bien un autre exemple est illustré ci-dessous. Deux threads sont instrumentalisés :

```
void ThreadA::run() {
    startSection(1);
    number = 0;
    startSection(2);
    int reg = number;
    startSection(3);
    number = reg + 1;
    endScenario();
}

void ThreadB::run() {
    startSection(4);
    number = 1;
    endSection();
    startSection(5);
    int reg = number;
    startSection(6);
    number = reg * 2;
    endScenario();
}
```



L'enchaînement des différentes sections qui peut être observé sur l'exemple précédent est ce qu'on appelle un *scénario-graphe de thread*. Dans le même exemple chaque thread fournit au système son propre scénario-graphe. Les lecteurs intéressés par plus de détails sur la composition d'un scénario peuvent consulter la classe `ScenarioGraphNode`.

Un détail sur notre implémentation est que le premier noeud du graphe a un identifiant numérique valant `-1`, et les noeuds suivants vont avoir un identifiant correspondant à la valeur utilisée comme argument de la fonction `startSection(int)`. Cette valeur doit être unique dans un scénario-graphe, mais ne doit pas nécessairement être unique entre différents scénario-graphes. Dans l'exemple ci-dessus il aurait été correct que les sections du `ThreadB` soient numérotées de 1 à 3.

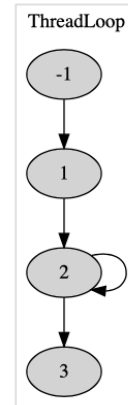
Il n'est par ailleurs pas nécessaire que tout le code se trouve dans la méthode `run()`. Il est tout à fait possible d'appeler d'autres méthodes, celles-ci pouvant être instrumentalisées avec les trois fonctions définissant les sections.

Il est intéressant de noter que ce système de graphe permet d'offrir des branchements. Ainsi nous pourrions avoir un thread décrit comme ceci :

```

void ThreadLoop::run() {
    bool cont = true;
    startSection(1);
    while (cont) {
        startSection(2);
        if (someCondition) {
            cont = false;
        }
    }
    startSection(3);
    doSomething();
    endScenario();
}

```



Ici nous observons que la section 2 peut être suivie de la section 3 **ou** de la section 2, de part la boucle while.

Ce système à base de graphe permet donc d'exprimer des scénarios potentiels.

Outre l'instrumentalisation, il faut évidemment définir le graphe. Ceci peut être fait dans le constructeur du Thread. Voici un exemple correspondant à l'exemple ci-dessus :

```

explicit ThreadLoop(std::string id = "") :
    ObservableThread(std::move(id))
{
    scenarioGraph = std::make_unique<ScenarioGraph>();
    auto scenario = scenarioGraph->createNode(this, -1);
    auto p1 = scenarioGraph->createNode(this, 1);
    auto p2 = scenarioGraph->createNode(this, 2);
    auto p3 = scenarioGraph->createNode(this, 3);
    scenario->next.push_back(p1);
    p1->next.push_back(p2);
    p2->next.push_back(p2);
    p2->next.push_back(p3);
    scenarioGraph->setInitialNode(scenario);
}

```

Nous y observons la création du graphe, puis du noeud initial (-1). Ensuite les trois noeuds de section sont créés, le noeud 1 étant ensuite ajouté au noeud de base. Les trois lignes suivantes présentent le graphe en lui-même, avec le fait que le noeud 2 est successeur du noeud 1 et de lui-même.

Un mécanisme est mis en place dans le reste du framework afin de pouvoir générer tous les scénarios possibles à partir d'un ensemble de threads. Cette génération va générer TOUS les scénarios possibles, en créant, pour chaque scénario, un vecteur où chaque élément indique quelle section de quel thread doit être exécutée ensuite.

Évidemment il peut y avoir des scénarios impossibles, notamment à cause des branchements qui dépendent des conditions de test. Certains scénarios termineront donc dans un *dead end*, ce qui est tout à fait normal.

Les threads instrumentalisés, et qui offrent leur graphe de scénario, doivent ensuite être combinés, et ce via une sous-classe de `PcoModel`. Dans celle-ci, plusieurs méthodes peuvent être surchargées. La principale, `build()` n'est appelée qu'une seule fois et vise à créer les threads et le constructeur de scénarios.

```

void build() override {
    threads.emplace_back(std::make_unique<ThreadA>("1"));
    threads.emplace_back(std::make_unique<ThreadA>("2"));
    threads.emplace_back(std::make_unique<ThreadB>("3"));

    scenarioBuilder = std::make_unique<ScenarioBuilderBuffer>();
    scenarioBuilder->init(threads, 9);
}

```

Dans l'exemple ci-dessus, nous observons la création de trois threads, deux de la classe `ThreadA` et un de la classe `ThreadB`. Chaque thread se voit attribuer un identifiant. Ensuite, un constructeur de

scénarios est créé et initialisé avec le tableau de threads et un nombre correspondant à la profondeur maximale qui pourra être jouée.

Avec un tel constructeur de scénario, tous les scénarios possibles seront joués. Le nombre total peut être très conséquent, et il existe donc deux méthodes alternatives.

Premièrement, durant la phase de mise en place, vous pouvez passer un argument au constructeur de `ScenarioBuilderBuffer`. Celui-ci fera que seulement un scénario sur N sera généré et joué. Ceci permet de valider votre approche tout en ayant la limitation de ne pas jouer tous les scénarios.

```
scenarioBuilder = std::make_unique<ScenarioBuilderBuffer>(1000000);
```

La deuxième méthode permet de choisir les scénarios à jouer.

```
void build() override {
    threads.emplace_back(std::make_unique<ThreadA>("1"));
    threads.emplace_back(std::make_unique<ThreadB>("2"));

    auto t1 = threads[0].get();
    auto t2 = threads[1].get();
    auto builder = std::make_unique<PredefinedScenarioBuilderIter>();
    std::vector<Scenario> scenarios = {
        {{t1, 1},{t1, 2},{t1, 3},{t2, 4},{t2, 5},{t2, 6}},
        {{t2, 4},{t2, 5},{t2, 6},{t1, 1},{t1, 2},{t1, 3}}
    };
    builder->setScenarios(scenarios);
    scenarioBuilder = std::move(builder);
}
```

Il s'agit de créer un objet de la classe `PredefinedScenarioBuilderIter`, puis de lui ajouter à la main des scénarios à jouer. Un scénario est un vecteur composé de paires (pointeur sur `ObservableThread`, numéro), où le numéro correspond au numéro du noeud tel que défini dans la méthode `build()` du thread. Dans l'exemple ci-dessus nous créons deux scénarios. Attention à l'usage des `unique_ptr`, qui nécessitent de faire un `move` du constructeur.

D'autres méthodes peuvent être surchargées, permettant d'exécuter du code avant chaque nouveau scénario (utilise pour recréer des objets qui auraient pu être mis à mal dans un scénario précédent), à la fin d'un scénario, à la fin de tous les scénarios. Il existe également une méthode permettant de vérifier des invariants.

Une fois cette classe implémentée, lancer une vérification se fait simplement en créant le modèle, puis en l'assignant à un model checker :

```
ModelNumbers model;
PcoModelChecker checker;
checker.setModel(&model);
checker.run();
```

A la fin d'une vérification, un résumé de l'état du système indique le nombre de scénarios qui ont terminé dans un certain état.

```
End : Unknown      : 0
End : Depth        : 0
End : Deadlock     : 0
End : AllScenario  : 1680
End : DeadEnd      : 0
```


1. `Unknown` : Ne devrait pas arriver
2. `Depth` : Le scénario s'est terminé car il a atteint la profondeur maximale alors qu'il y a encore des threads en train de s'exécuter. C'est OK.
3. `Deadlock` : Le scénario termine sur un deadlock. Il y a donc au moins un thread qui est bloqué alors que plus rien ne se passe. Ceci ne devrait pas arriver avec du code correct.
4. `AllScenario` : Tous les threads ont terminé leur exécution. C'est parfait.
5. `DeadEnd` : Le scénario se retrouve dans un *dead end*, c'est à dire qu'il n'est en fait pas jouable. C'est tout à fait normal, notamment dans le cas où le graphe n'est pas linéaire. En effet, en fonction des conditions un thread ne passera que par un des chemins d'une bifurcation.

Il est également possible d'afficher un scénario, ce qui peut être utile pour voir se qui génère un deadlock par exemple. La fonction offerte affiche quelque chose ainsi :

Scenario : {3,7} {3,8} {3,9} {2,4} {1,1} {2,5} {1,2} {2,6} {1,3}

Travail à réaliser

Votre travail consiste à choisir un problème de programmation concurrente, et d'implémenter de quoi le vérifier via du model checking. Il pourrait potentiellement s'agir d'un des exercices ou d'un exercice vu dans un test.

 Il est impératif de réaliser un système à base de sémaphores, les variables condition et moniteurs de Hoare ne sont pour l'instant pas supportés.

Travail à rendre

- La description de l'exemple choisi et la manière dont vous l'avez instrumentalisé, de même que la vérification que vous avez mise en place, doit être placée dans un rapport. Les fichiers sources doivent évidemment être correctement documentés et commentés.
- Inspirez vous du barème de correction pour savoir là où il faut mettre votre effort.

Barème de correction

Conception, conformité au cahier des charges et simplicité	45%
Exécution et fonctionnement	10%
Codage	15%
Documentation	20%
Commentaires au niveau du code	10%