**Nantes Université** **7 novembre 2023**

# Modélisation et Vérification des Systèmes Concurrents

X3IA110

-

# Devoir maison n°1

Matthéo LECRIVAIN
Quentin GOMES DOS REIS

M2 ALMA

# 1 Introduction

Ce rapport présente le travail effectué pour le projet de Modélisation et Vérification des Systèmes Concurrents. Les implémentations ont été réalisées en python, et le code est exécutable avec la commande `python main.py` à la racine du projet. Le code est fonctionnel, et testé avec la version **3.11.5** de Python.

**Nous ne pouvons donc pas garantir que le programme s'exécutera correctement si une version de Python inférieure à 3.11 est utilisée.**

# 2 Implémentation

Dans cette partie, nous allons détailler les différentes implémentations réalisées dans ce projet, en justifiant les choix effectués.

## 2.1 Système de Transition

Les systèmes de transition sont définis par une classe d'objets `TransitionSystem`, dans le fichier `TransitionSystem.py`. Ainsi, la classe contient :

- Une liste contenant les états initiaux.
- Un constructeur de classe, qui assigne une valeur à la liste.
- Un getter, pour récupérer la valeur de la liste.
- Une méthode pour ajouter de nouveaux états dans la liste.
- Une méthode pour récupérer la liste sous forme de string (facilitant l'affichage dans la console).

Les états initiaux sont définis par une liste du type `State`. Il s'agit d'une autre classe d'objets, représentant les états d'un système de transition. Ainsi, à partir de la liste des états initiaux, on peut retourner les états suivants avec la méthode `get_next_state()`, retourner les labels des états suivants avec `get_labels()`, ou encore définir les états suivants avec `set_next_states()`.

## 2.2 Proposition Logique

Nous avons fait le choix de modéliser les propositions logiques sous la forme de plusieurs classes. Elle héritent toutes de la classe abstraite `Proposition`, qui donne la structure des classes qui représentent des propositions. Pour ce projet, 3 opérateurs logiques ont été implémentés : `And`, `Or` et `Not`. Les deux premiers sont des opérateurs logiques binaires, qui prennent donc deux propositions en entrées, et retournent l'évaluation de ces propositions avec la méthode `eval()`. `Not` est lui un opérateur logique unaire, qui prend en entrée une proposition, et retourne l'opposé lors de l'évaluation.

Enfin, pour permettre l'évaluation des variables, nous avons implémenté une classe `Unary`, qui hérite également de la classe abstraite `Proposition`. `Unary` prend en entrée une variable. La fonction `eval()` renvoie vrai si la variable existe dans notre liste de variables, ou faux dans le cas contraire.

Toutes ces classes nous permettent de définir des propositions logiques simples. Par exemple, la proposition $\neg A \wedge B$ s'écrit :

```
proposition = And(Not(Unary("a")), Unary("b"))
```

### 2.3 Algorithme de Vérification d'Invariant

L'algorithme a été implémenté en suivant l'énoncé. Ainsi, la fonction `verify()` prend en entrée un système de transition et une proposition logique. Elle retourne un booléen indiquant si le système de transition satisfait toujours la proposition logique, et une liste des états qui contredise la proposition (cette liste est vide si l'algorithme retourne vrai). Notre implémentation de l'algorithme est bien documentée avec des commentaires, selon chaque étape donnée dans l'énoncé.

## 3 Tests du Programme

Plusieurs systèmes ont été implémentés pour tester le fonctionnement du programme. Ces systèmes sont définis dans des fonctions, dans le fichier `main.py`.

### 3.1 Exclusion mutuelle contrôlée par sémaphore

Comme demandé dans l'énoncé, le premier représente un système de transition modélisant deux processus concurrents, dont l'entrée dans le processus critique est contrôlée par un sémaphore binaire, avec la propriété d'exclusion mutuelle.

Un processus possède 3 états possibles : idle (`i`), waiting (`w`) ou critical section (`cs`). Le système de transition possède donc 8 états, qui prennent en compte l'état des deux processus, et un sémaphore binaire `y`.

Les transitions sont définies en appelant la méthode `set_next_states()` sur les états du système.

Les tests de ce système ont été réalisés sur deux propositions :

— $\phi_1 = \neg(cs_1 \wedge cs_2)$ : La section critique ne peut pas être accédée par les deux processus en même temps. La condition est évaluée à vrai par le programme.

— $\phi_2 = (cs_1 \wedge cs_2)$ : La section critique peut être accédée par les deux processus en même temps. Il s'agit de la proposition opposée, afin de tester une condition fausse sur le système. Ainsi, le programme retourne faux, et 3 contre-exemples.

### 3.2 Feux de circulation

Notre deuxième exemple représente deux feux de circulation à une intersection. Les feux peuvent être dans l'état rouge (`r`), jaune (`y`) ou vert (`g`), mais les deux doivent ne pas être jaune ou vert en même temps.

Les tests de ce système ont été réalisés sur trois propositions :

— $\phi_1 = \neg(g_1 \wedge g_2)$ : Les deux feux de circulation ne peuvent pas être dans l'état vert en même temps.

— $\phi_2 = \neg(y_1 \wedge y_2)$ : Les deux feux de circulation ne peuvent pas être dans l'état jaune en même temps.

— $\phi_3 = \neg((g_1 \wedge y_2) \vee (y_1 \wedge g_2))$ : Les deux feux de circulation ne peuvent pas être dans l'état vert et jaune en même temps.

Toutes ces propositions sont évaluées à vraies par notre programme.

### 3.3 Distributeur de boissons

Notre dernier exemple représente un distributeur de boissons, qui peut distribuer 4 types de boissons : `tea`, `coffee`, `hot chocolate` et `tomato soup`.

La machine commence dans l'état `idle`, `no drink selected`, `no drink delivered`. Le système permet la sélection d'une boisson uniquement quand une pièce est insérée. Après la sélection d'une boisson, la machine doit passer dans l'état boisson délivrée correspondant à la boisson sélectionnée.

Les tests de ce système ont été réalisés sur six propositions :

— $\phi_1 = \neg((\text{idle} \vee \text{coin returned}) \wedge (\text{no drink selected} \wedge (\text{tea delivered} \vee (\text{coffee delivered} \vee (\text{hot chocolate delivered} \vee \text{tomato soup delivered})))))$ :

Le distributeur ne peut pas distribuer de boisson si aucune pièce n'a été insérée et qu'aucune boisson a été sélectionnée.

— $\phi_2 = \neg((\text{idle} \vee \text{coin returned}) \wedge ((\text{tea selected} \vee (\text{coffee selected} \vee (\text{hot chocolate selected} \vee \text{tomato soup selected}))) \vee (\text{tea delivered} \vee (\text{coffee delivered} \vee (\text{hot chocolate delivered} \vee \text{tomato soup delivered})))))$ :

Le distributeur ne peut pas distribuer une boisson ou laisser choisir une boisson si aucune pièce n'a été insérée.

— $\phi_3 = \neg(\text{coin inserted} \wedge (\text{tea selected} \wedge ((\text{tomato soup delivered} \vee \text{coffee delivered}) \vee \text{hot chocolate delivered})))$ :

Le distributeur ne doit pas distribuer une autre boisson que du thé si le thé a été sélectionné et qu'une pièce a été insérée.

— $\phi_4 = \neg(\text{coin inserted} \wedge (\text{coffee selected} \wedge ((\text{tomato soup delivered} \vee \text{tea delivered}) \vee \text{hot chocolate delivered})))$ :

Le distributeur ne doit pas distribuer une autre boisson que du café si le café a été sélectionné et qu'une pièce a été insérée.

— $\phi_5 = \neg(\text{coin inserted} \wedge (\text{hot chocolate selected} \wedge ((\text{tomato soup delivered} \vee \text{tea delivered}) \vee \text{coffee delivered})))$ :

Le distributeur ne doit pas distribuer une autre boisson que du chocolat chaud si le chocolat chaud a été sélectionné et qu'une pièce a été insérée.

— $\phi_6 = \neg(\text{coin inserted} \wedge (\text{tomato soup selected} \wedge ((\text{hot chocolate delivered} \vee \text{tea delivered}) \vee \text{coffee delivered})))$ :

Le distributeur ne doit pas distribuer une autre boisson que de la soupe à la tomate si la soupe à la tomate a été sélectionnée et qu'une pièce a été insérée.

Toutes ces propositions sont évaluées à vraies par notre programme.

# 4 Conclusion

Toutes nos implémentations sont donc fonctionnelles, et nos exemples donnent des résultats satisfaisants. Pour plus de détails sur notre travail, tout notre code a été correctement commenté, ce qui donne des explications supplémentaires dans le cas où certains détails manqueraient dans ce rapport. L'exécution du programme donne également un affiche ordonné, qui apporte plus de détails sur les systèmes définis dans nos tests.