Rapport de projet : Model Checker CTL

Systèmes complexes

MASTER 2 INFORMATIQUE (PLS) 2020/2021

Nady SADDIK | nady95160@gmail.com
Rémi PHYU THANT THAR | piyohiko182@gmail.com

A l'attention de : M. Kaïs KLAI

07/01/2021 version 1.0

Institut Galilée – Université Sorbonne Paris Nord

99 Avenue Jean Baptiste Clément, 93430 Villetaneuse

Table des matières

1. Structure et implémentation

Notre objectif était de créer un model-checker pour la logique CTL à partir d'algorithmes de vérification vus en cours. Pour ce faire, nous utilisons le langage **Java** afin de créer des classes permettant de représenter la structure de Kripke et les différentes structures pour la formule CTL.

A) STATE.JAVA

Afin de représenter une structure de Kripke, il nous a semblé primordial de commencer d'abord par représenter un état d'une telle structure. Ainsi, en nous référant au cours, nous avons décidé qu'un état (State) devait posséder un nom (name), une liste de propositions atomiques qu'il vérifie (labels), une liste d'états enfants (childStates) et parents (parentStates). Un état peut être ou non initial (isInitial) et comporte un certain nombre d'enfants (nbChild).

La liste d'états enfants permet d'ajouter des transitions entre deux états (d'où l'absence d'une classe Transition.java).

B) KRIPKESTRUCTURE.JAVA

Après cela, nous avons pu commencer à modéliser une structure de Kripke. Elle est simplement représentée par une table de hashage (HashMap) associant un état (valeur) à son nom (clé). Cela nous permet ensuite de récupérer les états facilement et d'économiser de la mémoire lorsqu'on aura besoin de créer les algorithmes de vérification CTL.

C) CTLFORMULA.JAVA

Nous nous sommes fait la réflexion qu'une formule CTL pouvait avoir différentes formes (i.e. : AG, AF, EU, ...). Ainsi, CTLFormula est une interface destinée à être implémentée par ces dernières. La seule obligation pour être une formule CTL est d'avoir implémenté la fonction « *toCTL()* ». Celle-ci permet de convertir une formule CTL en formule de base, permettant ainsi de simplifier quelques relations (i.e. : *AG(p) = Not(EF(Not(p)))*).

a) Les opérateurs temporels quantifiés, et les opérateurs booléens : AF, AG, AU, AX, EF, EG, EU, EX, And, Or, Not

Ces opérateurs sont des formules CTL et doivent implémenter l'interface CTLFormula. Ils permettent de représenter différents types de formule. Un tel objet est représenté par son/ses opérande(s) et peut être simplifié en formule CTL de base avec « *toCTL()* ». Nous nous sommes servis des formules du cours pour cette partie.

b) Les propositions atomiques

Ces dernières sont également des formules CTL qui implémentent l'interface CTLFormula. Elles ne peuvent pas être plus simplifiées que ce qu'elles sont déjà car elles sont terminales. On a aussi décidé de ne créer qu'une seule instance de True et False car ils seront toujours les mêmes dans notre programme et n'auront pas besoin d'être modifiés.

D) MODELCHECKERCTL.JAVA

Afin d'instancier tous nos objets (structure de Kripke, formule CTL, ...), nous avons créé la classe **ModelCheckerCTL.java** qui est la classe principale de notre programme.

D'abord, elle va se charger de lire le fichier json passé en argument, et de parser ce qui se trouve à l'intérieur afin de construire la structure et d'instancier la formule CTL.

Pour ce faire, nous avons utilisé une librairie externe appelée **Google Gson**, qui nous sert à extraire des informations d'un fichier json. La syntaxe à adopter dans ce fichier est indiquée dans le fichier README.md présent sur le dépôt GitHub. Ainsi, il fut très aisé d'extraire les informations souhaitées, et de construire les états, les transitions, les labels, et de les intégrer dans notre structure de Kripke. Cette librairie nous a aussi été très utile pour extraire la formule CTL sous forme de chaîne de caractères.

Pour instancier la formule CTL, nous avons utilisé du pattern matching (grâce à la librairie **java.util.regex**). Nous avons ainsi créé plusieurs motifs à reconnaître, et une fois un motif reconnu le programme va instancier les bons objets. Les patterns à reconnaître ont été stockés dans une classe appelée **PatternList**, qui n'a aucun autre but.

Dans un second temps, et une fois la formule instanciée, la classe **ModelCheckerCTL.java** va appliquer les algorithmes de vérification sur notre formule en fonction de son type (une fonction a été créé pour chacun des six cas).

Enfin, elle nous retourne le résultat sous forme de texte dans l'invité de commandes.

2. Perspectives d’amélioration

Avec un peu plus de temps, nous aurions pu peaufiner nos patterns afin de s’assurer que, même avec une petite erreur (par exemple des parenthèses en trop), la formule CTL soit correctement parsée. On aurait aussi pu rajouter des messages d’erreur un peu plus précis pour l’utilisateur afin qu’il sache exactement l’erreur qu’il a faite.

Enfin, on aurait aussi pu adapter notre implémentation au modèle des réseaux de Pétri. Cela pourrait être intéressant à faire.

3. Application sur quelques exemples

Tout d'abord, nous nous permettons de montrer quelques exemples de notre système de pattern matching, qui a fonctionné dans tous les cas où nous l'avons testé jusqu'à présent pour reconnaître et « simplifier » des formules CTL.

```
MATCH_AU:A(req1Ucs2)
MATCH_ATOM:req1
MATCH_ATOM:cs2
A(req1 U cs2)
La fonction CTL "A(req1 U cs2)" est vérifiée par les états : 5, 7,
```

Figure 1 : Test simple du pattern matching

```
MATCH_EX:EX(A(req1Ucs2)&&req2)
MATCH_AND:A(req1Ucs2)&&req2
MATCH_AU:A(req1Ucs2)
MATCH_ATOM:req1
MATCH_ATOM:cs2
MATCH_ATOM:req2
EX((A(req1 U cs2) AND req2))
```

```
MATCH_EG:EG(A(req1Ucs2)&&req2)
MATCH_AND:A(req1Ucs2)&&req2
MATCH_AU:A(req1Ucs2)
MATCH_ATOM:req1
MATCH_ATOM:cs2
MATCH_ATOM:req2
NOT A(true U NOT (A(req1 U cs2) AND req2))
```

Figure 2 : Des tests un peu plus compliqués

Pour l'exemple, on construit le graphe représentant l'accès à la section critique de deux processus.

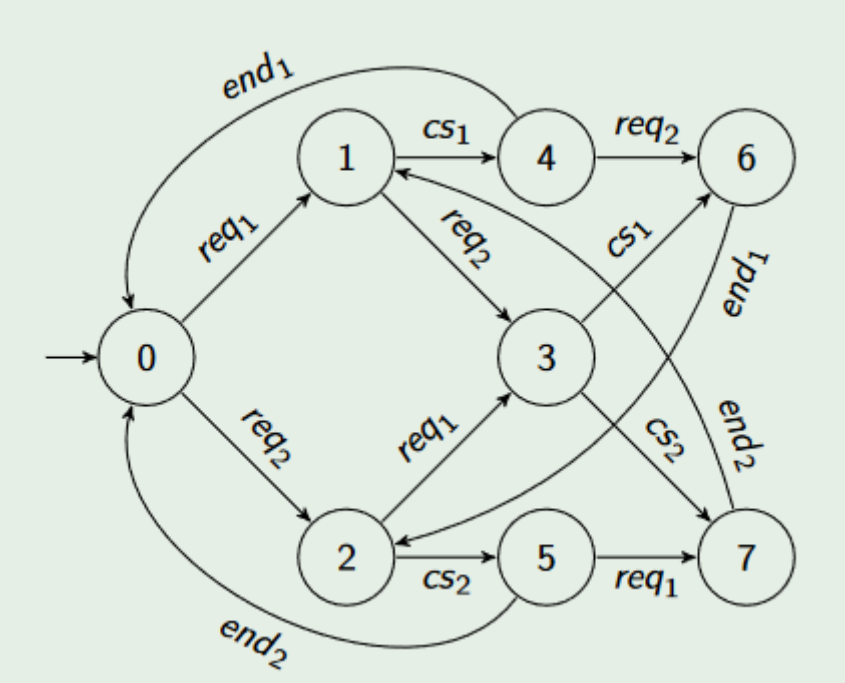

Figure 3 : Structure de Kripke de l'accès à la section critique par deux processus

On le représente par un fichier json en suivant la syntaxe présente dans le README.md (on ne peut malheureusement pas le copier dans ce rapport car cela prendrait beaucoup de place).

On choisit ensuite de tester des formules, dont plusieurs vues en cours.

- $\Phi = req_1$

```
"ctlformula": "req1"
```

```
La fonction CTL "req1" est vérifiée par les états : 1, 3, 7,
```

- $\Phi = \neg req_1$

```
"ctlformula": "not(req1)"
```

```
La fonction CTL "NOT req1" est vérifiée par les états : 0, 2, 4, 5, 6,
```

- $\Phi = req_1 \cap req_2$

```
"ctlformula": "req1 && req2"
```

```
La fonction CTL "(req1 AND req2)" est vérifiée par les états : 3,
```

- $\Phi = req_1 \cup req_2$

```
"ctlformula": "req1 || req2"
```

```
La fonction CTL "(req1 OR req2)" est vérifiée par les états : 1, 2, 3, 6, 7,
```

- $\Phi = EXreq_1$

```
"ctlformula": "EX(req1)"
```

```
La fonction CTL "EX(req1)" est vérifiée par les états : 0, 1, 2, 3, 5, 7,
```

- $\Phi = Ereq_1 \mathrm{U} cs_1$

```
"ctlformula": "E(req1 U cs1)"
```

```
La fonction CTL "E(req1 U cs1)" est vérifiée par les états : 1, 3, 4, 6, 7,
```

- $\Phi = Areq_1 \mathrm{U} cs_1$

```
"ctlformula": "A(req1 U cs1)"
```

```
La formule CTL "A(req1 U cs1)" est vérifiée par les états : 4, 6,
```

- $\Phi = Areq_1 \mathrm{U} cs_2$

```
"ctlformula": "A(req1 U cs2)"
```

```
La formule CTL "A(req1 U cs2)" est vérifiée par les états : 5, 7,
```