Model Checking CTL

PAQUET Michaël

Université Libre de Bruxelles

Résumé Le "model-checking" est une technique générale de "vérification automatique de systèmes informatiques". Elle permet donc de prouver, de façon automatique (à l'aide d'un algorithme) qu'un système est correct ... ou de détecter des bugs. Au cours de notre rapport, nous allons donc expliquer de manière plus détaillée ce qu'est le model-checking CTL et comment celui-ci fonctionne d'un point de vue algorithmique. Ensuite, nous vérifierons via une implémentation qui nous est propre le fonctionnement de ce modèle.

Table des matières

Introduction

Avec l'évolution technologique faisant acte de présence jours après jours, la plupart des services, informatique ou non, sont maintenant gérés via des système informatiques vérifiant leur bon fonctionnement. Mais pour certains services, il est impératif que ces systèmes fonctionnent correctement.

Prenons l'exemple d'une voie ferrée. Dans ce cas de figure-ci, il est impératif que les portes soient fermées lorsque le train traverse le passage à niveau. Comme dit précédemment, un système informatique s'occupera de gérer cela.
Mais puisqu'on ne peut pas se permettre la moindre erreur, nous nous devons de vérifier que le système n'échouera jamais, et c'est dans ce cas de figure que le

Model Checking fut créé et utilisé.

Au cours de l'explication de ce qu'est le model checking CTL, nous passerons donc en revue chacun des points de vue qu'on peut adopter lorsqu'on parle de *Model Checking CTL*.
Le premier point que nous aborderons sera l'aspect analytique du modèle (Qu'est-ce qu'on doit vérifier ?). Lors de ce chapitre, nous allons montrer la façon dont on découpe un système pour en faire des structures propice à l'analyse et à la vérification. Des structures comme les *Kripke models* ou encore les arbre seront bien évidemment abordées. Une fois ces objets observés, nous étudierons la façon dont on peut vérifier l'efficacité d'un système.

1 Chapitre 1

1.1 Du graphe aux arbres de recherche

Pour pouvoir analyser un système et pouvoir vérifier son efficacité, il est utile de le dériver en un graphe. Afin d'imager nos propos, reprenons l'exemple de la voie ferrée :
imaginons donc une voie ferrée sur laquelle aucun train ne circule. La voie est donc vide. On peut constate là un premier état de notre voie ferrée, l'état **vide**. Lorsque notre voie est vide, il se peut qu'à tout moment, un train soit **en approche**, et qu'on doive commencer à fermer les portes. Après quoi la voie sera **occupée** par le train pour qu'ensuite celui-ci reparte et laisse la voie **vide**.

Un tel système peut être représenté sous cette forme :

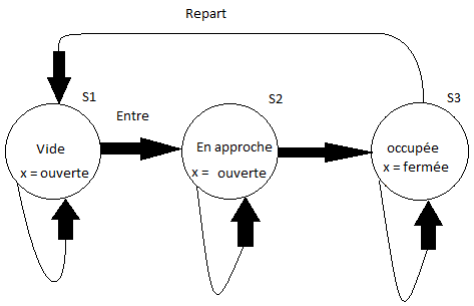

Figure 1. Modèle Kripke

Avec x étant une variable propositionnelle pour déterminer l'état des portes.
Un tel modèle est appelé *Kripke Model*.
Par convention, le modèle *Kripke* :

$$K = (V, S, S_0, I, R)$$

est le modèle *Kripke* dont :

1. **V**{ouverte, se ferme, fermée} est un ensemble fini de propositions atomiques.
2. **S**{vide, en approche, occupée} est en ensemble fini d'état.
3. $\mathbf{S_0}$ est l'état initial.
4. $\mathbf{I : S \rightarrow 2^v}$ est la fonction qui lie chaque états avec les propositions qui y sont liées.
5. **R** est l'ensemble des relations entre chacun des états.

Nous avons donc un graphe cyclique dont le nombre d'états est fini. Il est intéressant de voir que lorsque l'on créer un arbre sur base d'un tel graphe, celui-ci sera infini dû au carractère cyclique du graphe. Faisons le travail de transformer ce graphe en un arbre :

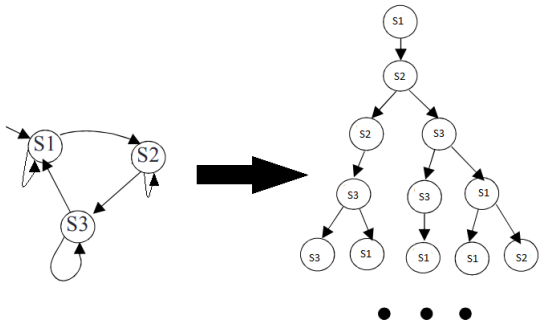

Figure 2. du Modèle Kripke à l'arbre de recherche

C'est maintenant grâce à de telles structures que nous allons pouvoir analyser notre système et vérifier que celui-ci soit infaillible.

1.2 Computational Tree Logic (CTL)

La syntaxe de la *Computational Tree Logic* utilise des formule booléennes : *True* et *False* sont donc des formules *CTL*.
Les variables propositionnelles sont des formules *CTL*.
Ainsi, si φ et ψ sont des formules *CTL* alors : $\neg\varphi$, $\varphi \wedge \psi$ et $\varphi \vee \psi$ le sont aussi.

En plus de cela, plusieurs opérations sont également des formules *CTL* :
— EX φ : φ apparait dans un des états suivants.
— EF φ : Dans au moins un chemin, φ apparait dans un état future.
— EG φ : Dans au moins un chemin, φ apparait dans tous les états futures.

— E[$\varphi \cup \psi$] : Dans au moins un chemin, φ apparait jusqu'à ce que ψ apparaisse.

Les opérations ici citées sont à chaque fois des "*There exists*" (EX = Exists next), mais il existe également les opérations "*ForAll*".
Ainsi, AX φ : φ apparait dans **tous** les états suivants. AF, AG, A$\cup$ sont donc aussi des opérations valables.

Voici à titre d'exemple un schéma pour se donner une idée de ce que ces opérations signifient.

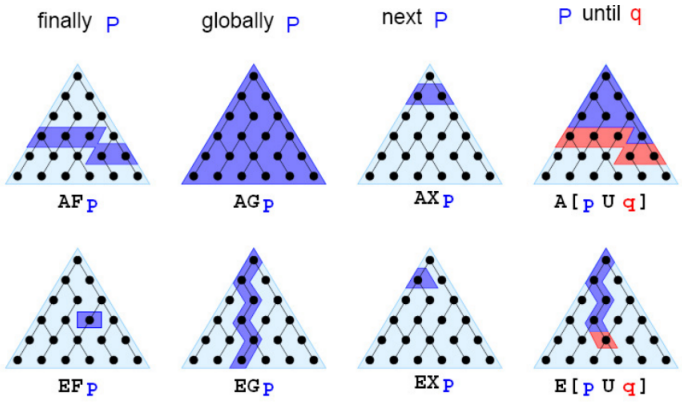

Figure 3. Exemple

De ces opérateurs découlent plusieurs de propriétés. En voici une liste non exhaustive :
— $\neg$ AX $\varphi \Leftrightarrow$ EX $\neg\varphi$
— $\neg$ AF $\varphi \Leftrightarrow$ EG $\neg\varphi$
— AF $\varphi \Leftrightarrow$ A[true $\cup$ φ]
— AG $\varphi \Leftrightarrow \varphi \wedge$ AX AG φ
— AF $\varphi \Leftrightarrow \varphi \vee$ AX AF φ
— A[false $\cup$ φ] $\Leftrightarrow$ E[false $\cup$ φ] $= \varphi$
— A[$\varphi \cup \psi$] $\Leftrightarrow \psi \vee (\varphi \wedge$ AX A[$\varphi \cup \psi$])
— E[$\varphi \cup \psi$] $\Leftrightarrow \psi \vee (\varphi \wedge$ EX E[$\varphi \cup \psi$])
— A[φ W ψ] $\Leftrightarrow \neg$ E[$\neg\psi \cup (\neg\varphi \wedge \neg\psi)$]

Maintenant que nous avons fait un tour non exhaustif de ce qu'était la syntaxe de *CTL*, abordons la sémantique :

— Soit φ une formule *CTL*, alors :

$$K, s \vDash \varphi$$

signifie que φ est vérifié à l'état **s**. La plupart du temps, K est omis puisque nous toujours au sein du même modèle *Kripke*.

— $\pi = \pi^0\pi^1...$ est un chemin.
— π^0 est l'état initial (la racine).
— π^{i+1} est un des états succédant π^i.

A nouveau, une série de propriétés découle des définitions précédentes :
— AX $\varphi \Leftrightarrow \forall\pi \cdot \pi^1 \vDash \varphi$
— EX $\varphi \Leftrightarrow \exists\pi \cdot \pi^1 \vDash \varphi$
— AG $\varphi \Leftrightarrow \forall\pi \cdot \forall i \cdot \pi^i \vDash \varphi$
— EG $\varphi \Leftrightarrow \exists\pi \cdot \forall i \cdot \pi^i \vDash \varphi$
— AF $\varphi \Leftrightarrow \forall\pi \cdot \exists i \cdot \pi^i \vDash \varphi$
— EF $\varphi \Leftrightarrow \exists\pi \cdot \exists i \cdot \pi^i \vDash \varphi$
— A$[\varphi \ \cup \ \psi] \Leftrightarrow \forall\pi \cdot \exists i \cdot \pi^i \vDash \psi \wedge \forall j \cdot 0 \leq j < i \Rightarrow \pi^j \vDash \varphi$
— E$[\varphi \ \cup \ \psi] \Leftrightarrow \exists\pi \cdot \exists i \cdot \pi^i \vDash \psi \wedge \forall j \cdot 0 \leq j < i \Rightarrow \pi^j \vDash \varphi$

Définition : Une formule *CTL* est *ACTL* si elle n'utilise que les connecteurs universels temporels (AX, AF, AG, AU).
Définition : Une formule *CTL* est *ECTL* si elle n'utilise que les connecteurs existentiels temporels (EX, EF, EG, EU).
Définition : Les formules *CTL* qui ne sont ni *ACTL* ni *ECTL* sont appelées mixtes.

Grâce à la syntaxe et à la sémantique que nous avons évoqué, il est maintenant possible de parler de la sécurité et la vivacité d'un système.

Définition : Un système sûre implique qu'il ne se passera jamais quelque chose de mauvais. AG $\neg$ *bad*
Définition : Un système vivace implique que quelque chose de bien arrivera toujours. AG AF *good*

2 CTL Model Checking

Nous avons vu plus haut la liste des formules CTL possible. Ces huit opérations, à savoir EX, EF, EG, EU, AX, AF, AG et AU peuvent en réalité s'exprimer avec les trois opérations EX, EF et EU.
— AX $\varphi \Leftrightarrow \neg$ EX $\neg\varphi$
— EF $\varphi \Leftrightarrow$ E[True U φ]
— AG $\varphi \Leftrightarrow \neg$ EF $\neg\varphi$
— AF $\varphi \Leftrightarrow \neg$ EG $\neg\varphi$
— A$[\varphi$ U $\psi] \Leftrightarrow \neg$ E$[\neg\psi$ U $(\neg\varphi \wedge \neg\psi)] \wedge$ EG $\neg\psi$

Ainsi, il nous suffit d'implémenter EX, EF et EU afin d'implémenter l'ensemble des opérations possibles en CTL. Lors de cette section, les algorithmes choisis afin de remplir notre objectif seront abordés.

Avant toute chose, afin d'obtenir notre structure d'arbre sur laquelle nous pourrons effectuer nos opérations, il a été choisi de créer un modèle Kripke sur base d'un fichier .text. Ce modèle Kripke correspond aux modèles Kripke tels que vu sur la Figure 1. Une fois ce modèle Kripke construit, il nous est possible de créer l'arbre correspondant, à nouveau tel que vu sur la figure 2. A titre d'exemple, voici ce que contient le fichier .text sur lequel nous construisons notre modèle Kripke.

```
V  = {ouvert,se ferme,fermé}
S  = {vide,approche,occupé}
S0 = {vide}
I  = {vide:ouvert,approche:se ferme,occupé:fermé}
R  = {vide:vide-approche,approche:approche-occupé,occupé:occupé-vide}
```

Figure 4. Fichier text

Le parsing de ce fichier text en un modèle Kripke donne exactement le modèle vu sur la Figure 1. Lorsque l'on créer désormais l'arbre sur base de ce modèle Kripke, nous obtenons un arbre ayant cette structure :

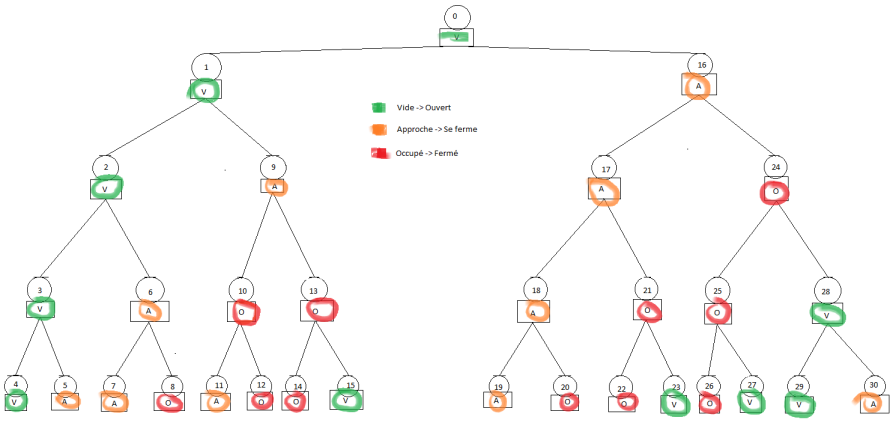

Figure 5. arbre

Attention cependant, les propriétés du noeud N 10 ont été modifiées pour que celui-ci corresponde à l'état "Fermé" afin de pouvoir illustrer d'avantage de résultats. Nous pouvons désormais implémenter les opérations voulue afin d'observer le résultat qu'auront celles-ci sur notre arbre.

2.1 EX

EX s'avère extrêmement simple puisqu'il suffit d'étiqueter l'état "EX φ" aux noeuds ayant un enfant avec l'étiquetage φ.

Algorithm 1 Exist Next

```
1: procedure EX(Node noeud, label φ)
2:     if un des enfants de noeud a le label φ then
3:         ajouter le label EXφ au noeud actuel
4:     enfants ← enfantsDuNoeudActuel
5:     for each enfant in enfants do :
6:         if enfant existe then
7:             EX(enfant, φ)
```

2.2 EG

Voici l'algorithme permetant d'implémenter EG φ :
Tout d'abord, on étiquette les noeuds ayant le label φ avec le label EG φ. Ensuite, et ce tant qu'il n'y a plus de changement, on enlève le label EG φ des noeuds n'ayant pas d'enfants avec le label EG φ.

Algorithm 2 Exist Global

```
1: procedure EG(Node noeud, label φ)
2:     ajouter le label EGφ à tous les noeuds ayant le label φ
3:     do
4:         SomethingHasChanged ← False
5:         Enlever le label EGφ aux noeuds n'ayant pas d'enfant avec le label EGφ
6:         if Une modification a eu lieu durant l'étape précédente then
7:             SomethingHasChanged ← True
8:     while SomethingHasChanged
```

2.3 EU

Voici l'algorithme afin d'implémenter EU[φ U ψ] :
En premier lieu, on étiquette avec le label EU[φ U ψ] les noeuds ayant le label ψ. Ensuite, et ce tant qu'il n'y a pas de changement, on ajoute le label EU[φ U ψ] aux noeuds ayant le label φ et ayant un enfant avec le label EU[φ U ψ].

Algorithm 3 Exist Union

1: **procedure** EU(Node *noeud*, label φ, label ψ)
2: ajouter le label EU[φ U ψ] à tous les noeuds ayant le label ψ
3: **do**
4: *SomethingHasChanged* $\leftarrow$ False
5: Ajouter le label EU[φ U ψ] aux noeuds ayant le label φ et ayant un enfant avec le label EU[φ U ψ]
6: **if** Une modification a eu lieu durant l'étape précédente **then**
7: *SomethingHasChanged* $\leftarrow$ True
8: **while** *SomethingHasChanged*