Construire un model checker avec Spot

E. Renault

L'objectif de ce TP est de vous familiariser avec la vérification formelle de systèmes. Dans ce TP les systèmes seront modélisés en utilisant le langage DVE (`https://is.muni.cz/www/208047/meandve.pdf`) et on utilisera Spot (`https://spot.lrde.epita.fr/`) pour exprimer des propriétés (via la logique LTL) sur ces systèmes et vérifier qu'elles sont correctes.

1 Remarques préliminaires & mise en place de l'environnement

Ce TP s'exécute dans les conditions suivantes :

1. Rendez-vous sur `http://spot-sandbox.lrde.epita.fr/`
2. Créez un Notebook IPython 3 à l'aide du bouton "New"
3. Changez le titre du Notebook : au lieu de `Untitled`, mettre un identifiant unique (login, nom, etc.)

Notes :

- Vous êtes tous en train d'exécuter du code sur le même compte de la même machine. Merci donc de ne pas faire de boucles infinies ou autres bétises pas drôles pour les copains.
- La machine `spot-sandbox` est remise à zéro tous les matins (et parfois plus souvent) ; si vous souhaitez conserver votre travail, pensez à télécharger le Notebook d'ici ce soir.
- Si vous avez une machine Debian, vous devez installer l'outil DiVinE (patché par LTSmin). Vous trouverez le paquet Debian associé ici : `http://lrde.epita.fr/repo/debian/unstable/`
- La machine `spot-sandbox` est configurée de manière à pouvoir générer les structure de Kripke en utilisant l'outil DiVinE. Avant de commencer le TP, il faut donc charger et initialiser les modules Python de Spot et vérifier la présence de DiVinE. Cela peut être fait simplement en tapant :

```
import spot
import spot.ltsmin
spot.setup()
```

Il ne vous reste plus qu'à faire shift-enter pour forcer l'évaluation de la cellule.

2 Traduire une formule simple

L'objectif de cette première partie est de vous familiariser avec la logique LTL et les ω-automates. On utilise pour cela Spot une bibliothèque de manipulation de formules LTL et d'ω-automates développée au LRDE.

Tapez le code Python qui suit et validez de la même façon :

```
spot.translate('a U b')
```

dans ce cas, vous avez obtenu un automate de Büchi, avec condition d'acceptation portant sur les états. Fair porter la condition d'acceptation sur les transitions n'apporterait rien de mieux (= pas plus petit).

Avec

```
spot.translate('G(req -> F resp)')
```

on a par défaut un automate avec condition d'acceptation sur les transitions. Spot préfère en général des automates avec conditions sur les transitions, sauf pour certaines formules simples où l'on sait qu'utiliser les états suffit. Si l'on tient absolument à avoir un automate de Büchi avec acceptation sur les états, on peut faire :

```
spot.translate('G(req -> F resp)', 'ba')
```

Comparez en particulier

```
spot.translate('(G F a) & (G F b)')
```

avec

```
spot.translate('(G F a) & (G F b)', 'ba')
```

Définissez les deux fonctions suivantes :

```
def implies(f1, f2):
   f1 = spot.formula(f1)
   f2 = spot.formula_Not(spot.formula(f2))
   return spot.product(spot.translate(f1), spot.translate(f2)).is_empty()

def equiv(f1, f2):
   return implies(f1, f2) and implies(f2, f1)
```

Maintenant on peut tester l'équivalence de deux formules LTL :

```
equiv('X!a', '!Xa')
```

Trouvez comment réécrire les formules suivantes en utilisant seulement **U** et **X** comme opérateurs temporels (vous pouvez aussi utiliser tous les opérateurs booléens). Par exemple la formule **F a** peut se réécrire **true U a**.

Comment réécririez-vous les formules qui suivent ?

— **G a**
— **a W b**
— **a R b**

Aidez-vous de la représentation sous forme d'automate pour deviner le sens des opérateurs dont je n'ai pas parlé. Utilisez **equiv()** pour vérifier votre réécriture.

L'exercice de traduire une spécification du français vers LTL n'est pas toujours facile. Dans ce qui suit, vérifiez que les automates produits par vos formules correspondent bien au scénario décrit en français.

- Écrivez une formule LTL pour spécifier le fait que la variable **light_on** est toujours vraie lorsque **door_open** est vraie.

- Écrivez une formule LTL pour spécifier qu'un feu peut être orange (**!v&o& !r**) pendant plusieurs instants, à condition que cela soit entre un instant où il est vert (**v& !o& !r**) et un instant celui où il est rouge (**!v& !o&r**).

- Quand on regarde l'automate de **G(req -> F resp)**, on se rend bien compte qu'on peut accepter des réponses qui ne suivent aucune demande. Comment pourrait-on spécifier qu'il y a exactement une réponse qui suit chaque requête ?

3 Vérifier un système simple

L'objectif de cet exercice est maintennt de vous faire manipuler un model et d'y vérifier certaines propriétés.

- Commençons par définir un système simple composé d'un unique processus. Ce processus a pour objectif de trouver la valeur de la variable partagée **positive** *f*. Le code suivant permet de charger le modèle suivant dans la variable *m*. **Attention, le retour à la ligne après *m* est important !**

```
%%dve m
int f = 3;
process R {
  int p = 1, found = 0;
  state i, e;
  init i;
  trans
     i -> i {guard p != f; effect p = p +1;},
     i -> e {guard p == f; effect found = 1;},
     e -> e {};
}
system async;
```

- Vous pouvez maintenant voir quelles sont les propositions atomiques disponibles pour ce modèle en tapant simplement :

```
m
```

- À partir du modèle précédent, on peut construire une structure de Kripke, en spécifiant les propositions atomiques que l'on souhaite observer. L'exemple suivant construit une structure de Kripke en spécifiant que seule la proposition atomique "`R.found`" est observée.

```
k = m.kripke(["R.found"]); k
```

- On souhaite maintenant s'assurer que "*le processus R fini toujours par rester dans un état où la valeur de f est trouvée*". Déterminez la formule de logique LTL permettant d'exprimer ce comportement.
- On peut maintenant construire et afficher le produit synchronisé entre la structure de Kripke k et l'automate a (le résultat de la question précédente).

```
p = spot.otf_product(k, a); p
```

- Pour savoir si la propriété est vérifiée, le produit doit être vide : la commande suivante doit donc retourner **True** si vous avez correctement exprimé la formule LTL.

```
p.is_empty()
```

- Définissez maintenant la fonction python modelcheck qui prend en paramètre une formule LTL et un modèle et vérifie si cette formule est vérifiée ou non. *Rappel : à partir d'un automate a on peut récupérer la liste des propositions atomiques observées en faisant a.ap(). À partir d'une proposition atomique x, on peut récupérer son nom en faisant x.ap_name().*
- On souhaite maintenant augmenter notre système en rajoutant un processus Q qui a exactement le même comportement que le processus R. Définissez le modèle associé, générez la structure de Kripke associée, et vérifiez la formule précédente. Que se passe-t-il ?

4 Définir son propre système

Maintenant que vous avez bien compris comment fonctionne la chaîne de vérification de formules LTL, il est temps de modéliser votre propre système.

L'objectif ici est de construire un processus *Oven* qui représentera un four à micro-ondes. Un four à micro-ondes possède plusieurs états :

- *idle* : le four ne fait rien. Dans cet état, il peut passer de manière non déterministe à l'état *running* ou à l'état *open*.
- *running* : le four doit y rester pour 5 unités de temps. Il s'agit de l'état dans lequel il fait chauffer la nourriture. Depuis cet état, le micro ondes ne peut que retourner dans l'état *idle*
- *open* : le four peut y rester indéfiniment jusqu'à ce que quelqu'un décide de fermer la porte pour ramener dans l'état *idle*.

Il ne vous reste plus qu'à introduire deux variables booléennes : *dooropen* qui est à vraie dès que la porte est ouverte, et *heating* qui est à vraie quand le micro ondes est en train de chauffer. Vous pouvez maintenant vérifier que :

- Le micro onde ne peut être en train de chauffer la porte ouverte.
- Le micro onde qui passe dans l'état *running* fini toujours par repasser dans l'état *idle*.

On souhaite maintenant pouvoir ouvrir la porte du four à n'importe quel moment. Changez votre modélisation pour refléter cela. Quel impact cela a-t-il sur les propriétés ?

5 Vérifier un système plus complexe

On s'intéresse maintenant à la vérification d'un système plus complexe.

```
byte req[4];
int t,p;
byte v;
process cabin
{
  state idle,mov,open;
  init idle;
  trans
        idle -> mov  {guard v>0;},
        mov -> open  {guard t==p;},
        mov -> mov   {guard t<p; effect p=p-1;},
        mov -> mov   {guard t>p; effect p=p+1;},
        open -> idle {effect req[p]=0,v=0;}
  ;
}
process environment
{
  state read;
  init read;
  trans
        read -> read {guard req[0]==0; effect req[0]=1;} ,
        read -> read {guard req[1]==0; effect req[1]=1;} ,
        read -> read {guard req[2]==0; effect req[2]=1;} ,
        read -> read {guard req[3]==0; effect req[3]=1;}
  ;
}
process controller
{
  byte ldir;
  state wait,work,done;
  init wait;
  trans
    wait -> work {guard v==0; effect t=t+(2*ldir)-1;},
    work -> wait {guard t<0 || t==4; effect ldir=1-ldir;},
    work -> done {guard t>=0 && t<4 && req[t]==1;},
    work -> work {guard t>=0 && t<4 && req[t]==0; effect t=t+(2*ldir)-1;},
    done -> wait {effect v=1;}
  ;
}
system async;
```

- La structure de Kripke associée à ce modèle est trop grosse pour être affichée complètement à l'utilisateur (par défaut l'affichage est limité à 50 états). Pour afficher seulement 15 états de cette structure vous pouvez taper :

```
1  k.show('.<15')
```

- Vérifiez si la cabine peut finir par rester indéfiniment à l'étage 1 (i.e. "p == 1").