Génération et analyse automatisées d'ensembles de règles pour Picobot

Quentin BAILLEUL Jérémy BOSSUT Romain PHILIPPON

12 janvier 2015

Table des matières

1 Introduction

Picobot est un projet utilisé comme support au cours introductoire d'informatique du Dr. Zachary Dodds au Harvey Mudd College de Californie. Ce programme modélise le comportement d'un robot-aspirateur dont le but est d'examiner tous les emplacements inoccupés de son environnement. Il est possible de programmer le robot en lui fournissant un ensemble de règles à respecter.

Notre projet utilise le langage de spécification déclaratif Alloy pour générer ces règles. L'objectif est de chercher le plus petit ensemble de règles qui permet au robot de parcourir l'ensemble d'un niveau, quelque soit sa position initiale.

La section suivante présente l'architecture du projet. Nous présentons le fonctionnement du programme Picobot en détails dans la section 2 et les modèles Alloy utilisés dans la section 3. Les sections 4 et 5 montrent respectivement les résultats obtenus et leur analyse. Enfin nous concluons notre travail dans la section 6.

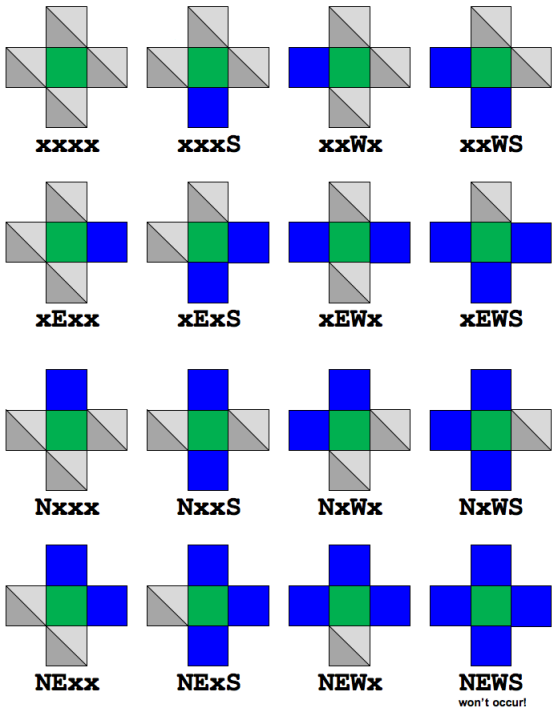
2 Picobot

Le robot Picobot se trouve dans un environnement rectangulaire modélisé par un tableau de cellules. Les cellules peuvent être inoccupées ou contenir un mur empêchant le passage du robot. Le robot est uniquement capable de percevoir l'occupation des cellules alentours. Comme il n'est pas doté de mémoire interne alors il ne peut pas savoir sur quelle cellule il se trouve et s'il l'a déjà parcouru. Il n'est donc pas possible d'utiliser un algorithme de recherche de chemins tel que Dijkstra ou A*.

Pour se déplacer, le robot respecte un ensemble de règles pré-établies. Une règle est formée de la façon suivante : *CurrentState Surroundings -> MovementDirection NewState*

CurrentState et *NewState* sont deux nombres entiers compris entre 0 et 99. Ces nombres représentent le numéro de l'état dans lequel se trouve le robot respectivement avant et après l'exécution de la règle. Au lancement du programme, le robot se trouve dans l'état 0.

Surroundings est une chaîne de quatre caractères représentant l'occupation des cellules voisines. Les environnement possibles sont les suivants :



Il est possible de remplacer un des caractères par '*'. Le caractère '*' signifie qu'on ne se soucie pas de la présence ou non d'un obstacle à l'emplacement désigné.

MovementDirection représente la direction dans laquelle on souhaite que le robot se déplace. Le robot peut se déplacer dans les quatre directions cardinales (N pour Nord, E pour Est, W pour Ouest, S pour Sud) ou choisir de ne pas bouger (X).

Exemple : Considérons l'ensemble de règles pour Picobot suivant.

0 x*** -> N 0
0 N*** -> W 1
1 ***x -> S 1
1 ***S -> X 0

La règle *0 x*** -> N 0* est suivie lorsque le robot est dans l'état 0 et qu'il détecte une cellule libre au Nord. Lorsque cette situation se produit, le robot se déplace d'une unité vers le Nord et reste à l'état 0. Lorsque le robot rencontre un obstacle au Nord, la règle *0 N*** -> W 1* change son état à 1 et le fait se déplacer vers l'Ouest. Une fois le robot dans l'état 1, il suit la règle règle *1 ***x -> S 1* et va se déplacer vers le Sud jusqu'à ce qu'il rencontre un mur dans cette direction. Dans ce cas, il revient à l'état 0 selon la règle *1 ***S -> X 0*.

3 Modèles Alloy

Le caractère '*' a été introduit dans le but d'obtenir une représentation plus succincte des règles. Pour tout ensemble de règles qui utilise ce caractère, il est alors possible de trouver un ensemble de règles équivalent qui ne l'utilise pas.

```
0 **x* -> W 0
0 **W* -> E 1

1 *x** -> e 1
1 *E** -> W 0
```

```
0 xxxx -> W 0   0 xxWx -> E 1   1 xxxx -> E 1   1 xExx -> W 0
0 xxxS -> W 0   0 xxWS -> E 1   1 xxxS -> E 1   1 xExS -> W 0
0 xExx -> W 0   0 xEWx -> E 1   1 xxWx -> E 1   1 xEWx -> W 0
0 xExS -> W 0   0 xEWS -> E 1   1 xxWS -> E 1   1 xEWS -> W 0
0 Nxxx -> W 0   0 NxWx -> E 1   1 Nxxx -> E 1   1 NExx -> W 0
0 NxxS -> W 0   0 NxWS -> E 1   1 NxxS -> E 1   1 NExS -> W 0
0 NExx -> W 0   0 NEWx -> E 1   1 NxWx -> E 1   1 NEWx -> W 0
0 NExS -> W 0   0 NEWS -> E 1   1 NxWS -> E 1   1 NEWS -> W 0
```

Dans cette section, nous présentons deux modèles Alloy - le premier sans '*' et le second avec - qui permettent de générer des ensembles de règles pour Picobot.

3.1 Modèle original

3.1.1 Ensembles

Le modèle comprend trois signatures et deux énumérations :

Ensemble	Description

`sig Rule {`
`  current_state: Int,`
`  env: Surroundings,`
`  next: Action`
`}`
	Rule définit une règle applicable où *current_state* désigne l'état initial du robot, *env* son environnement actuel et *next* l'action à effectuer

`sig Surroundings {`
`  north: Wall,`
`  east: Wall,`
`  west: Wall,`
`  south: Wall`
`}`
	Surroundings définit un environnement en indiquant la présence ou non d'un mur dans chacun des quatre points cardinaux.

`sig Action {`
`  next_state: Int,`
`  move: Move`
`}`
	Action définit une action à effectuer où *next_state* désigne le nouvel état du robot et *move* le déplacement à effectuer

`enum Move {N, E, W, S, X}`	**Move** définit les cinq mouvements possibles par le robot : un pour chaque direction cardinale (N,E,W,S) et X pour désigner l'absence de mouvement.
`enum Wall {True, False}`	**Wall** est identique à l'ensemble des booléens et définit l'état d'un mur (présent : True, absent : False)

3.1.2 Faits

Afin de garantir la compatibilité des règles générées avec le programme Picobot, six faits sont utilisés :

Numéro d'état valide

Le programme Picobot ne gère que 100 numéros d'état numérotés de 0 à 99 :

```
fact validState {
  all r:Rule | r.current_state >= 0 && r.current_state < 100
  all a:Action | a.next_state >= 0 && a.next_state < 100
}
```

État initial

Le robot débutant à l'état 0, il est nécessaire qu'il existe au moins une règle commençant par cet état. De plus, cette règle doit exécuter une action avec un mouvement effectif dans l'une des quatre directions :

```
fact initialState {
  some r:Rule | r.current_state = 0 && r.next.move != X
}
```

Déterminisme du comportement

Le comportement du robot doit être déterministe. Il ne peut donc pas y avoir d'ambiguïté sur le choix de la règle à appliquer dans une situation donnée. Autrement dit, deux règles ayant le même numéro d'état et le même environnement ne peuvent pas coexister.

```
fact noDuplicatedRule {
  all r1: Rule | all r2:Rule-r1 | r1.current_state =
      r2.current_state => r1.env != r2.env
}
```

En conséquence, il ne peut pas exister non plus deux instances Surroundings en Alloy qui représente le même environnement.

```
fact noDuplicatedSurroundings {
  all s1:Surroundings | no s2:Surroundings-s1 | s1.north =
      s2.north && s1.east = s2.east && s1.west = s2.west &&
      s1.south = s2.south
}
```

Enchaînement des règles

Pour empêcher les règles isolées, il est nécessaire que chaque action emmène vers un état pour lequel il existe une règle.

```
fact consistentStateNumbers {
  all a:Action | some r:Rule | a.next_state = r.current_state
}
```

On s'assure également que chaque action dépend d'une règle pour ne pas avoir d'actions inutilisées qui influent sur le fait précédent.

```
fact allActionsHaveRule {
  all a:Action | some r:Rule | r.next = a
}
```

3.1.3 Prédicats d'optimisation

Quatre prédicats sont ajoutés pour générer un meilleur ensemble de règles. Les modifications induites par ces prédicats ne sont pas nécessaires au bon fonctionnement de Picobot mais permettent d'éviter de générer des règles qui ne peuvent pas être appelées ou qui n'apportent aucune plus-value à l'ensemble.

Immobilisme

Une règle qui conserve l'état et n'exécute pas de mouvement bloquerait le robot dans une boucle infinie. Le prédicat suivant permet d'empêcher la génération de ce type de règles.

```
pred neverHoldStill {
  all r:Rule | r.next.move = X => r.next.next_state !=
      r.current_state
}
```

Puisque le robot ne peut pas se retrouver dans une situation où il est encerclé de murs, il n'est pas nécessaire de générer un Surroundings où les quatre directions sont bloquantes.

```
pred neverStucked {
  no s:Surroundings | s.north = True && s.east = True && s.west
      = True && s.south = True
}
```

Il n'est pas pertinent de demander au robot de se déplacer dans une direction lorsque l'on sait au préalable qu'elle contient un mur.

```
pred noMoveIntoAWall {
  all r:Rule | r.env.north = True => r.next.move != N
  all r:Rule | r.env.east = True => r.next.move != E
  all r:Rule | r.env.west = True => r.next.move != W
  all r:Rule | r.env.south = True => r.next.move != S
}
```

Règles inatteignables

Pour qu'une règle soit atteignable, il faut que son état initial soit accessible depuis une autre règle. Il faut donc, pour chaque règle dont l'état initial *state* est différent de 0, qu'il existe une règle dont l'état initial est différent de *state* qui exécute une action amenant à l'état *state*. Il est important de noter que cette condition est nécessaire mais pas suffisante. Par exemple, dans un ensemble de trois règles dont les transformations d'état sont R1 : 0 -> 0, R2 : 1->2 et R3 : 2->1, le prédicat est respecté mais les règles R2 et R3 sont inatteignables.

```
pred preventInaccessibleRule {
  all r1:Rule | some r2:Rule | r1.current_state != 0 =>
      r1.current_state != r2.current_state && 2.next.next_state
      = r1.current_state
}
```

3.1.4 Autres prédicats

Le modèle comprend également quelques prédicats dont les apports sont essentiellement cosmétiques. Ceux-ci permettent l'utilisation de numéros d'états consécutifs, d'empêcher la génération de deux actions identiques ou d'environnements non utilisés. Ces prédicats n'ayant pas d'influence sur l'ensemble de règles généré, ils sont inactifs par défaut afin d'améliorer les performances d'Alloy.

3.2 Modèle alternatif

Le modèle alternatif autorise l'autorisation du caractère '*' pour décrire un environnement. Pour réaliser cette modification, un troisième état est ajouté dans l'énumération ***Wall*** :

```
enum Wall {True, False, Star}
```

La mise en place du caractère '*' impose également la création d'un nouveau fait pour garder l'aspect déterministe des ensembles de règles :

```
fact compatibleRules {
  all r1: Rule | no r2:Rule-r1 |
  (r1.current_state = r2.current_state) &&
  (r1.env.north = r2.env.north || r1.env.north = Star ||
      r2.env.north = Star) &&
  (r1.env.east = r2.env.east || r1.env.east = Star ||
      r2.env.east = Star) &&
  (r1.env.west = r2.env.west|| r1.env.west = Star ||
      r2.env.west = Star) &&
  (r1.env.south = r2.env.south || r1.env.south = Star ||
      r2.env.south = Star)
}
```

3.3 Comparaison des modèles

Modèle original	**Modèle alternatif**
Augmente grandement le nombre de règles nécessaires à la réalisation d'un niveau de Picobot	Permet l'utilisation d'une écriture réduite des règles.
Limite le nombre d'objets Surroundings possibles à $2^4 - 1 = 15$	Augmente le nombre d'objets Surroundings possibles à $3^4 - 1 = 80$
Pour 25 instances d'objets, Alloy réduit le problème à : 159374 variables, dont 4350 primaires et 429710 clauses	Pour 25 instances d'objets, Alloy réduit le problème à : 181326 variables, dont 4450 primaires et 480863 clauses.

4 Résultats

5 Analyse des résultats

6 Conclusion