Université de Sherbrooke
Faculté des sciences
Département d'informatique

IGL501: Méthodes formelles en génie logiciel

professeur Marc Frappier

Modélisation avec Alloy : Notes synthétiques

1 Modélisation de système en Alloy

Alloy est un *model checker*. Il permet de définir des symboles à l'aide de signatures (`sig`) et des contraintes sur les valeurs de ces symboles avec des `fact`. Alloy peut ensuite chercher un modèle avec la commande `run`. Un modèle est une affectation d'une valeur à chaque symbole, en satisfaisant toutes les contraintes. Alloy peut aussi prouver que tous les modèles satisfont une formule, avec la commande `check` (*ie*, il peut prouver un théorème). Alloy travaille sur des modèles finis; la taille de chaque signature est définie avec les commandes `run` et `check`.

Si on veut modéliser un système, il faut introduire explicitement une signature pour représenter l'état du système, et une trace pour représenter une trace d'exécution du système. L'état est typiquement représenté par une signature `State` dont les attributs sont les variables d'état du système. La trace est représentée par un ordre total sur l'ensemble des états, que l'on définit avec le module `ordering` de la librairie `util` en l'instanciant avec la signature `State` comme suit

```
open util/ordering[State]
```

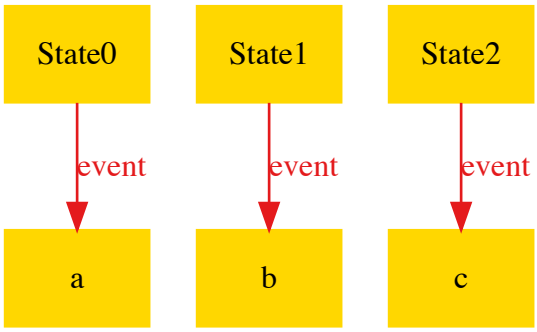
La spécification du module `ordering` est disponible dans Alloy dans le menu File→Open Sample Models..., dans le répertoire `util`.

Alloy permet de faire de la vérification sur des traces dont la longueur est déterminée par la taille de la signature `State`. On pourrait aussi calculer le graphe de transition d'un système, mais cela est généralement trop coûteux en temps et en mémoire

Voici un exemple simple de modèle de système, où un état n'a qu'un attribut, soit l'évènement qui amène le système dans cet état. La fonction `first` retourne le premier état de la trace, et `next` retourne le successeur d'un état de la trace. Ce modèle représente la trace $[a, b, c]$.

```
open util/ordering[State]
enum Event {a,b,c}

sig State {
event : Event
}

run test1 {
first.event = a
first.next.event = b
first.next.next.event = c
} for 3 State
```

Voici la représentation graphique de cet état.



On voit que l'état State0 est obtenu par l'évènement a, et ainsi de suite.

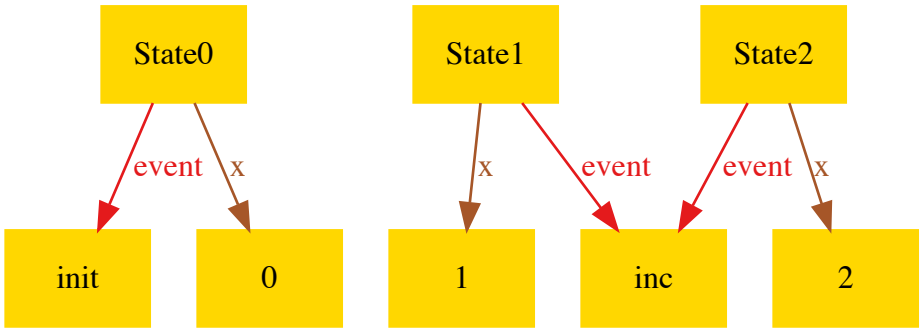
Nous allons maintenant définir un système avec une variable qui est un entier, dont la valeur initiale est 0 et une opération `inc` qui incrémente la valeur de cet entier. Une opération est représentée par un prédicat qui relie deux états consécutif `s` et `s'`.

```
open util/ordering[State]
enum Event {init,inc}

sig State {
x : Int,
event : Event
}

pred Init[s:State]
{
s.x = 0
s.event = init
}

pred Inc[s,s' : State]
{
s'.x = plus[s.x,1]
s'.event = inc
}

run test1 {
Init[first]
and all s : State-last |
    let s' = next[s] |
        Inc[s,s']
} for 3 State
```

La commande `run` impose deux contraintes sur la trace.

1. Le premier élément de la trace satisfait le prédicat `Init`.
2. Les autres éléments de la trace incrémentent x de 1.

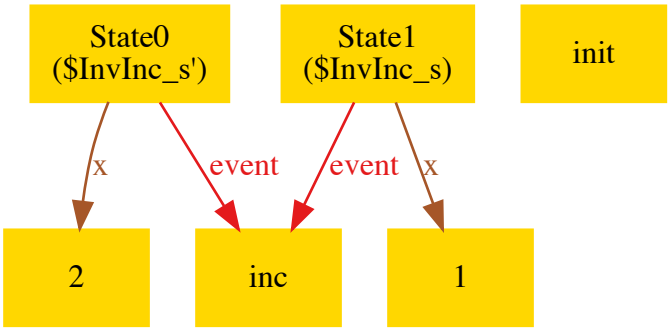
Voici la représentation graphique de cette trace.



Nous allons maintenant tenter de prouver un invariant pour ce système, c'est-à-dire que la valeur de `x` est comprise entre 0 et 1.

```
pred Inv[s:State]
{
s.x >= 0 and s.x <= 1
}

check InvInit {
all s : State |
    Init[s] => Inv[first]
} for 1 State

check InvInc {
all s,s' : State |
       Inv[s] and Inc[s,s']
    =>
       Inv[s']
} for 2 State
```

La commande `check InvInit` vérifie que l'état initial satisfait l'invariant. La commande `check InvInc` vérifie que si un état satisfait l'invariant, alors opération `Inc` exécutée à partir de cet état retourne un état qui satisfait l'invariant. Le premier `check` est satisfait, mais pas le deuxième, car l'opération `Inc` ne vérifie pas que la valeur de `x` peut dépasser 1. Voici le contre-exemple trouvé par Alloy.



Si `x` vaut 1 dans `s`, alors sa valeur augmente à 2 dans `s'`, ce qui viole l'invariant. Si l'invariant était $x \in 0..8$, alors la commande `check` n'aurait pas trouvé de contre-exemple parce que, par défaut, la valeur maximale d'un entier est 7. Cette valeur est spécifiée dans la signature prédéfinie `int`, et

sa valeur par défaut est `4 int`, ce qui donne le nombre de bits utilisés pour représenter un entier. Soit n le nombre de bits. La valeur maximale est $2^{n-1} - 1$, ce qui vaut 7 pour la valeur par défaut $n = 4$. Il faut donc faire attention à la taille de chaque signature quand on effectue une vérification.

Nous allons définir une nouvelle opération `IncSafe` qui préserve l'invariant en testant la valeur de `x` dans `s`.

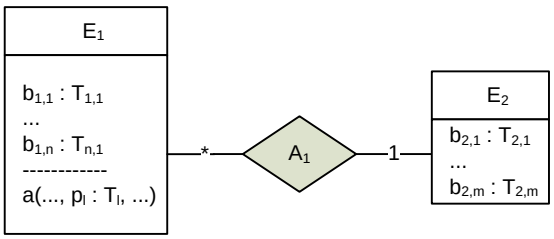
```
pred IncSafe[s,s' : State]
{
s.x < 1
s'.x = plus[s.x,1]
s'.event = inc
}

check InvIncSafe {
all s,s' : State |
       Inv[s] and IncSafe[s,s']
    =>
       Inv[s']
} for 2 State
```

L'opération `IncSafe` préserve l'invariant.

2 Spécification de système à partir d'un modèle entité-association

2.1 Spécification de l'état du système

Le diagramme ci-dessous illustre un modèle entité-association générique.



Voici comment ce modèle est traduit en Alloy.

1. Pour chaque entité E_i, définir une signature `Ei`. Cette signature définit les instances *possibles* de l'entité E_i.

   ```
   sig Ei {}
   ```

2. Définir une signature `State` qui représente les états possibles du système. Dans l'exemple de la bibliothèque, c'est ce que j'ai appelé `Librairie` (ou `Lib` dans la version anglaise).

   ```
   sig State { ... }
   ```

3. Pour chaque entité E_i, définir un attribut `Eis` dans `State`, pour représenter les instances *actuelles* de cette entité dans un état.

```
sig State {
...
Eis : set Ei,
...
}
```

4. Pour chaque attribut $b_{i,j}$ d'une entité E_i, définir un attribut `bij` de type `Ei -> Tij` dans la signature `Ei`. `Tij` est une signature existante (par exemple, dénotant une autre entité), ou à définir (si nécessaire) ou bien un type prédéfini comme `Int`.

```
sig State {
...
bij : Ei -> Tij,
...
}
```

5. Pour chaque association A_k entre des entités E_i et E_j , définir un attribut `Aks` dans `State`, pour représenter les instances *actuelles* de cette association dans un état.

```
sig State {
...
Aks : Ei -> Ej,
...
}
```

2.2 Spécification des actions du système

Une action représente un service que l'utilisateur peut invoquer pour interagir avec le système. Pour chaque action $a(\ldots, p_i : T_i, \ldots)$, définir un prédicat `a[..., pi : Ti, ..., s, s' : State]`.

```
pred a[..., pi : Ti, ..., s, s' : State]
{
predicat precondition
predicat postcondition
predicat nochange
}
```

3 Spécification des propriétés

3.1 Propriété d'invariance

Une propriété d'invariance `q(s:State)` est satisfaite par tous les états du système. On peut la prouver comme suit:

1. On définit un prédicat `q[s:State]` qui représente la propriété.

```
pred q[s : State]
{
...
}
```

2. On montre qu'elle est satisfaite pour l'état initial du système. Soit `Init[s:State]`, un prédicat qui retourne vrai ssi s est un état initial du système.

```
assert qVraiInit {all s : State | Init[s] => q[s]}
check qVraiInit for x but 1 State
```

3. Pour chaque action a, on montre, que si la propriété est satisfaite avant l'exécution de a, alors elle est aussi satisfaite après l'exécution de a.

```
assert qPreserveParA {all s,s' : State | a[p,s,s'] and q[s] => q[s']}
check qPreserveParA for x but 2 State
```

4 Traduction en Alloy d'une spécification B

1. Clause `SETS`. Les ensembles sont définis comme des signatures Alloy

`SETS A,B`	`sig A {}` `sig B {}`

2. Clauses `CONSTANTS` et `PROPERTIES`. Les ensembles sont définis comme des signatures en Alloy. Utilisez `extends` ou `in` pour représenter les sous-ensembles, les partitions, etc. Les scalaires sont définis comme des singletons (`one sig`). Les autres contraintes sont données comme des `fact`.

`CONSTANTS b,B1,f`
`PROPERTIES`
`b` $\in$ `B` $\wedge$ `B1` $\subseteq$ `B` $\wedge$ `f` $\in$ `B1` $\rightarrow$ `B` $\wedge$
$\exists$ `x` $\cdot$ `(x` $\in$ `B1` $\wedge$ `f(x) = b)`
	`one sig b extends B`
	`sig B1 in B { f : one B}`
	`fact`
	`{ some x : B1 | x.f = b }`

3. Clauses `VARIABLES` et `INVARIANT`. Les variables sont déclarées comme des attributs de la signature `State` décrite dans la section précédente, en déclarant les fonctions comme des *relations*. On ajoutera une commande pour prouver que ce sont des fonctions. En B, on doit prouver que les opérations préservent l'invariant. On prouve l'invariant en Alloy en utilisant la commande `check`. On n'utilise pas un `fact` pour décrire l'invariant, car cela revient à restreindre les modèles à ceux où l'invariant est vrai. On veut plutôt prouver que *tous* les modèles de la spécification satisfont l'invariant. On définit l'invariant à l'aide d'un prédicat qui sera utilisé dans une commande `check ...` {`...Invariant...`}.

VARIABLES a3,A1,g
INVARIANT
a3 ∈ A1 ∧
A1 ⊆ A ∧
g ∈ A1 ⇸ A1 ∧
∃ y · (y ∈ A1 ∧ a3 ↦ y ∈ g)
	sig State {
	a3 : A,
	A1 : set A,
	g : A1 -> A1 // g relation }
	
	pred Invariant[s : State]{
	one s.a3 // a3 scalaire
	s.a3 in s.A1
	s.A1 in A
	// g est une fonction
	s.g in s.A1 -> lone s.A1
	some y : s.A1 |
	s.a3 -> y in s.g
	}

4. Clauses INITIALISATION. On utilise un prédicat sur un état pour vérifier que les valeurs des variables sont égales aux valeurs initiales.

INITIALISATION
a3 := a1
‖ A1 := {a1,a2}
‖ g := {a1 ↦ a1}
	pred Init[s:State]
	{
	s.a3 = a1
	s.A1 = a1+a2
	s.g = a1 -> a1
	}

5. Clauses OPERATIONS. On utilise un prédicat pour représenter chaque opération.

OPERATIONS
add_g(x) = PRE x ∈ A1
THEN g(x) := x END
	pred add_g[x : A, s,s' : State]
	{
	x in s.A1
	s'.g = s.g ++ x -> x
	s'.a3 = s.a3 // no change
	s'.A1 = s.A1 // no change
	}

Pour exécuter une action, on exécute un **run** avec un prédicat sur la trace. Par exemple, voici la commande qui fait l'exécution de add_g(a2) à partir de l'état initial.

```
run test3 {
    Init[first]
and let s = next[first] | add_g[a2,first,s]
} for 3 but exactly 2 State
```

Pour prouver l'invariant, on exécute la commande check suivante.

```
check CheckInv {
all s : State | Init[s] => Invariant[s]
all x : A, s,s' : State |
```

```
      Invariant[s] and add_g[x,s,s']
   =>
      Invariant[s']
} for 2 but 2 State
```

5 Spécification des intervalles de valeurs pour Run et Check

Les commandes `run` et `check` permettent de spécifier le nombre d'éléments (le *scope*) à utiliser pour chaque signature. Il faut faire attention aux aspects suivants.

- Le scope de `State` détermine la longueur de la trace qui sera cherchée par Alloy dans une commande `run`. Il faut prévoir un nombre suffisant d'éléments afin de vérifier une propriété. Par exemple, considérez la spécification `biblio.als` et essayons de vérifier si un livre peut être réservé. Il faut au moins deux membres, un livre et 6 `State`, pour observer un évènement `reserve`:

  ```
  init, acquire, join, join, lend, reserve.
  ```

  Pour observer un `take`, il faut au moins 8 `State`.

  ```
  init, acquire, join, join, lend, reserve, return, take.
  ```

- Le scope de la signature `Int` est déterminé par le scope name `int` (tout en minuscule). Sa valeur par défaut est 4 `int`. Un scope de n `int` indique qu'on utilise n bits pour représenter un entier signé; il dénote l'intervalle -2^{n-1} .. $(2^{n-1}) - 1$. Par exemple, le scope `for 4 int` dénote l'intervalle $-8..7$ pour la signature `Int`. Si on oublie de spécifier le scope de `int`, cela peut masquer certaines erreurs. En effet, Alloy ne parcoure que les instances qui respectent les scopes. Par exemple, les scopes suivants ne permettent pas de détecter un dépassement de la limite de prêt si la limite de prêt est fixée à 7 dans la spécification.

  ```
  run {...} for 15 State, 8 Book, 8 Member
  ```

  En effet, pour dépasser la limite de prêt établie à 7, il faut 8 prêts, mais le scope de `int` étant non spécifié, la valeur par défaut 4 `int` est utilisée par Alloy, et le plus grand entier utilisé sera 7, donc aucun modèle avec 8 prêts ne sera exploré, même s'il y a 8 livres. Il faut alors indiquer les scopes suivants pour trouver une violation du nombre de prêts.

  ```
  run {...} for 15 State, 8 Book, 8 Member, 5 int
  ```

  En effet, l'entier le plus grand est alors $2^{5-1} - 1 = 15$.

- Pour vérifier un invariant, il faut seulement 2 `State`, soit l'état avant et l'état après, ce qui rend la vérification des invariants très rapide en Alloy, beaucoup plus rapide qu'avec ProB. Notons que ProB ne vérifie que les états accessibles à partir de l'état initial, car il parcourt le graphe de transitions du système. Avec Alloy, le patron de vérification des invariants présentés dans la section précédente vérifie toutes les transitions $s \longrightarrow s'$, peut importe si s est accessible à partir de l'état initial; cela reproduit fidèlement les obligations de preuve du langage B pour les invariants.