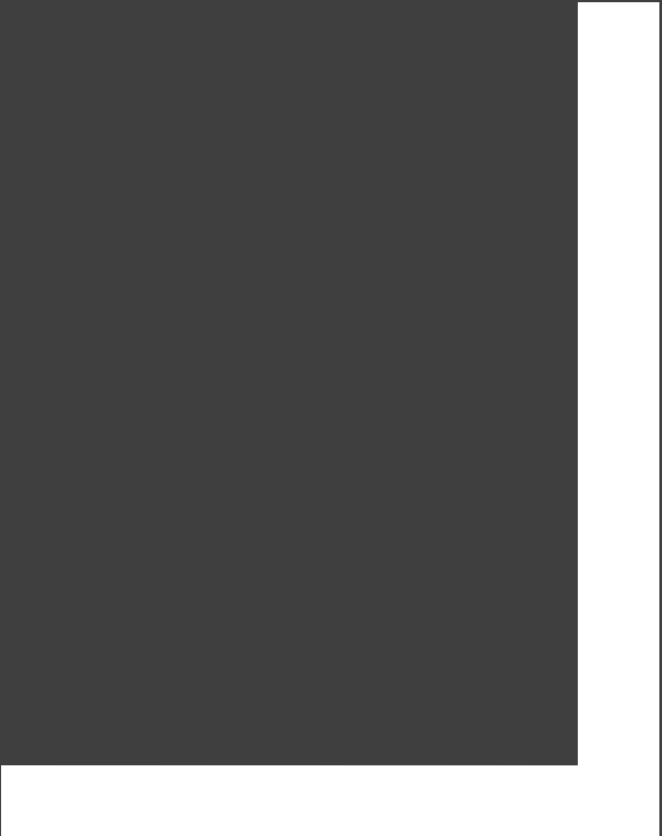




ALLOY

David BRELLMANN CI 2020 Rob

C'est quoi Alloy ?

- Un **langage déclaratif** de spécification pour définir des modèles
- Un **analyseur** pour tester des modèle par des contre-exemples

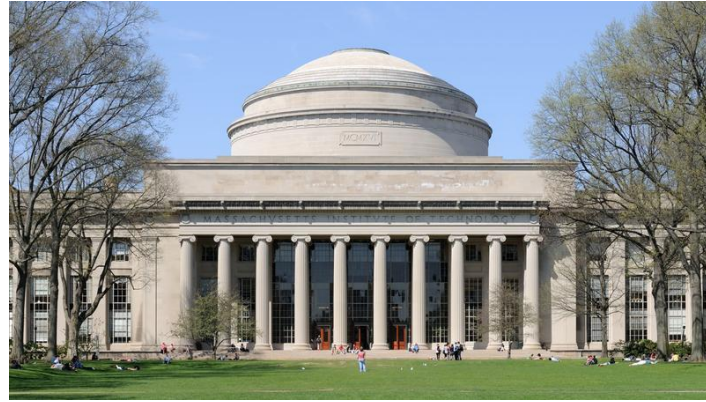


C'est quoi Alloy ?

- Syntaxe inspirée du langage Z
- Analyseur inspiré de l'analyseur SMV
- Première version d'Alloy développée en 1997 par l'équipe de Daniel Jackson au M.I.T



Z (Oxford)



Alloy (MIT)

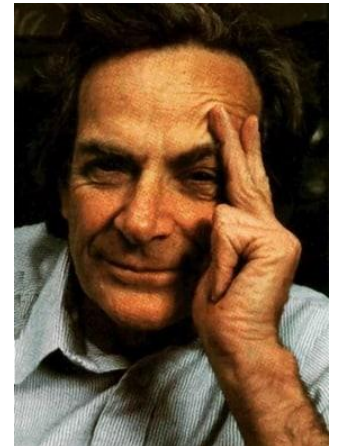


SMV (CMU)

Applications d'Alloy

- Utile dans la gestion d'un projet
- Modélisation de différents systèmes comme des robots
- Vérification de la fiabilité du système

“The first principle is that you must not fool yourself, and you are the easiest person to fool.” – Richard P. Feynman



Plan du cours

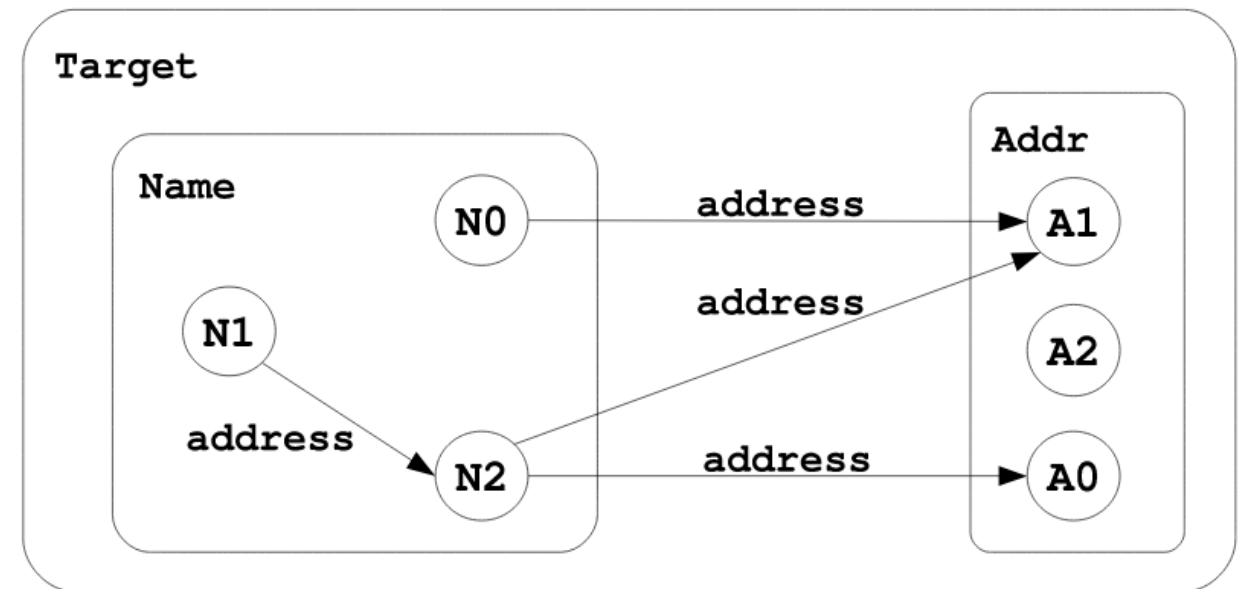
- Les éléments logiques sous Alloy
- Le langage Alloy
- Exemple en lien avec le projet
- Références

ELÉMENTS LOGIQUES SOUS ALLOY

Éléments logiques sous Alloy

Atoms & Relations

- **Atoms** : briques de base d'Alloy. Entités indivisibles et immuables.
- **Relations** : structures reliant les atoms entre eux sous forme de tuples.



```
Name = { (N0), (N1), (N2) }
Addr = { (A0), (A1), (A2) }
Target = { (N0), (N1), (N2), (A0), (A1), (A2) }
address = { (N0, A1), (N1, N2), (N2, A1), (N2, A0) }
```

Eléments logiques sous Alloy

Ensembles de références

none	<i>empty set</i>
univ	<i>universal set</i>
iden	<i>identity relation</i>

```
Name = { (N0), (N1), (N2) }
Addr = { (A0), (A1) }

none = {}
univ = { (N0), (N1), (N2), (A0), (A1) }
iden = { (N0, N0), (N1, N1), (N2, N2),
          (A0, A0), (A1, A1) }
```

Éléments logiques sous Alloy

Opérateurs sur les ensembles

+	<i>union</i>
&	<i>intersection</i>
-	<i>difference</i>
in	<i>subset</i>
=	<i>equality</i>

```
Name  = { (N0), (N1), (N2) }  
Alias  = { (N1), (N2) }  
Group  = { (N0) }  
RecentlyUsed = { (N0), (N2) }
```

```
Alias + Group = { (N0), (N1), (N2) }  
Alias & RecentlyUsed = { (N2) }  
Name - RecentlyUsed = { (N1) }  
RecentlyUsed in Alias = false  
RecentlyUsed in Name = true  
Name = Group + Alias = true
```

Éléments logiques sous Alloy

Produit cartésien

-> *cross product*

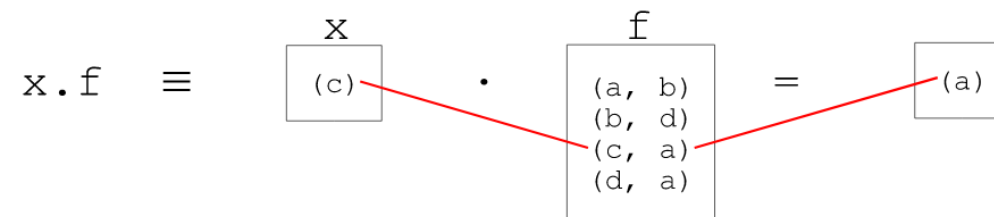
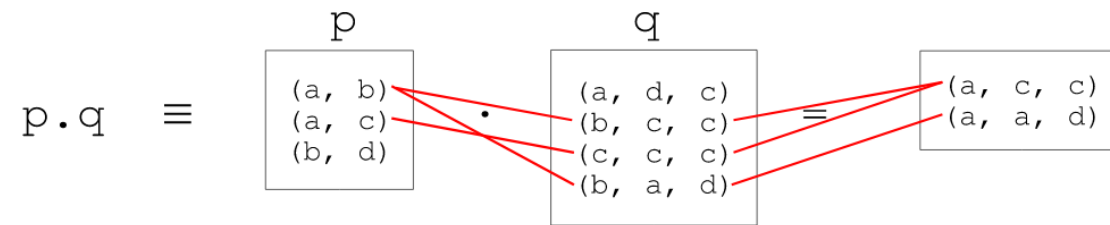
```
Name = { (N0), (N1) }  
Addr  = { (A0), (A1) }  
Book  = { (B0) }
```

```
Name->Addr = { (N0, A0), (N0, A1),  
               (N1, A0), (N1, A1) }
```

```
Book->Name->Addr =  
  { (B0, N0, A0), (B0, N0, A1),  
    (B0, N1, A0), (B0, N1, A1) }
```

Éléments logiques sous Alloy

Jointure



Éléments logiques sous Alloy

Jointure

.	<i>dot join</i>
[]	<i>box join</i>

$e1[e2] = e2.e1$
$a.b.c[d] = d.(a.b.c)$

```
Book = { (B0) }  
Name = { (N0), (N1), (N2) }  
Addr = { (A0), (A1), (A2) }  
Host = { (H0), (H1) }
```

```
myName = { (N1) }  
myAddr = { (A0) }
```

```
address = { (B0, N0, A0), (B0, N1, A0), (B0, N2, A2) }  
host = { (A0, H0), (A1, H1), (A2, H1) }
```

```
Book.address = { (N0, A0), (N1, A0), (N2, A2) }  
Book.address[myName] = { (A0) }  
Book.address.myName = { }
```

Éléments logiques sous Alloy

Opérateurs unaires

$\sim$ *transpose*
 $\wedge$ *transitive closure*
 $*$ *reflexive transitive closure*
apply only to binary relations

$\wedge r = r + r.r + r.r.r + \dots$
 $*r = \mathbf{iden} + \wedge r$

```
Node = { (N0), (N1), (N2), (N3) }  
next = { (N0, N1), (N1, N2), (N2, N3) }  
  
~next = { (N1, N0), (N2, N1), (N3, N2) }  
^next = { (N0, N1), (N0, N2), (N0, N3),  
          (N1, N2), (N1, N3),  
          (N2, N3) }  
*next = { (N0, N0), (N0, N1), (N0, N2), (N0, N3),  
          (N1, N1), (N1, N2), (N1, N3),  
          (N2, N2), (N2, N3), (N3, N3) }
```

Éléments logiques sous Alloy

Quantifieurs

```
all x: e | F
all x: e1, y: e2 | F
all x, y: e | F
all disj x, y: e | F
```

all	<i>F holds for every x in e</i>
some	<i>F holds for at least one x in e</i>
no	<i>F holds for no x in e</i>
lone	<i>F holds for at most one x in e</i>
one	<i>F holds for exactly one x in e</i>

```
some n: Name, a: Address | a in n.address
some name maps to some address — address book not empty

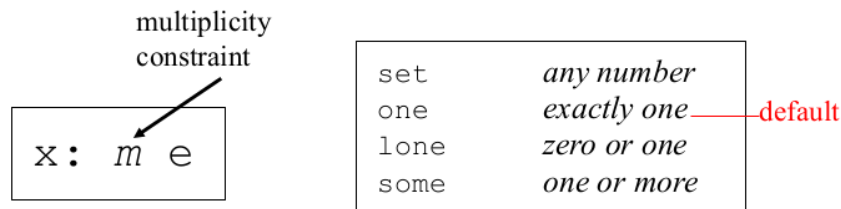
no n: Name | n in n.^address
no name can be reached by lookups from itself — address book acyclic

all n: Name | lone a: Address | a in n.address
every name maps to at most one address — address book is functional

all n: Name | no disj a, a': Address | (a + a') in n.address
no name maps to two or more distinct addresses — same as above
```

Eléments logiques sous Alloy

Déclarations de relations



RecentlyUsed: set Name
RecentlyUsed is a subset of the set Name

senderAddress: Addr
senderAddress is a singleton subset of Addr

senderName: lone Name
senderName is either empty or a singleton subset of Name

receiverAddresses: some Addr
receiverAddresses is a nonempty subset of Addr

Éléments logiques sous Alloy

Définitions d'ensembles

$$\{x1: e1, x2: e2, \dots, xn: en \mid F\}$$
$$\{n: \text{Name} \mid \text{no } n.^{\wedge}\text{address} \ \& \ \text{Addr}\}$$

set of names that don't resolve to any actual addresses

$$\{n: \text{Name}, a: \text{Address} \mid n \rightarrow a \text{ in } ^{\wedge}\text{address}\}$$

binary relation mapping names to reachable addresses

Éléments logiques sous Alloy

Cardinalités

#r	<i>number of tuples in r</i>
0, 1, ...	<i>integer literal</i>
+	<i>plus</i>
-	<i>minus</i>

=	<i>equals</i>
<	<i>less than</i>
>	<i>greater than</i>
=<	<i>less than or equal to</i>
>=	<i>greater than or equal to</i>

sum x: e | ie
sum of integer expression ie for all singletons x drawn from e

all b: Bag | #b.marbles =< 3
all bags have 3 or less marbles

#Marble = **sum** b: Bag | #b.marbles
*the sum of the marbles across all bags
equals the total number of marbles*

LANGUAGE ALLOY

Langage Alloy

Signatures

Les **signatures** sont les définitions des types et des relations composant le modèle (ensemble d'atoms).

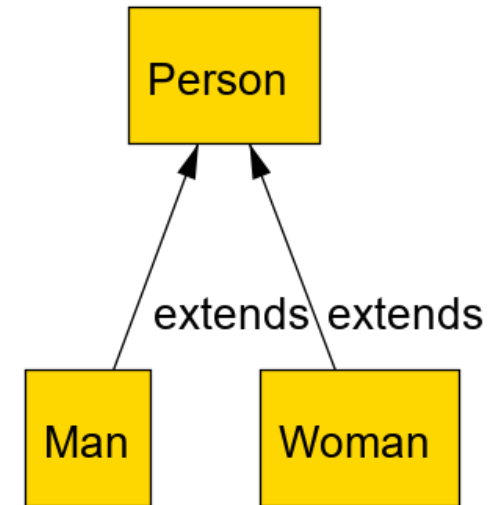
```
sig A {}  
set of atoms A  
  
sig A {}  
sig B {}  
disjoint sets A and B (no A & B)  
  
sig A, B {}  
same as above  
  
sig B extends A {}  
set B is a subset of A (B in A)  
  
sig B extends A {}  
sig C extends A {}  
B and C are disjoint subsets of A  
(B in A && C in A && no B & C)  
  
sig B, C extends A {}  
same as above
```

```
abstract sig A {}  
sig B extends A {}  
sig C extends A {}  
A partitioned by disjoint subsets B and C  
(no B & C && A = (B + C))  
  
sig B in A {}  
B is a subset of A – not necessarily  
disjoint from any other set  
  
sig C in A + B {}  
C is a subset of the union of A and B  
  
one sig A {}  
lone sig B {}  
some sig C {}  
A is a singleton set  
B is a singleton or empty  
C is a non-empty set
```

Langage Alloy

Signatures

```
abstract sig Person {  
    . . .  
}  
  
sig Man extends Person {  
    . . .  
}  
  
sig Woman extends Person {  
    . . .  
}
```



- Tous les hommes et femmes sont des personnes.
- Aucune personne ne peut être à la fois un homme et une femme.
- Toutes les personnes sont soit des hommes soit des femmes.

Langage Alloy

Fields

Les **champs** (fields) d'une signature sont des relations dont le domaine est un sous-ensemble de la signature.

```
sig A {f: e}  
f is a binary relation with domain A  
and range given by expression e  
f is constrained to be a function  
(f: A -> one e) or (all a: A | a.f: e)
```

```
sig A {  
  f1: one   e1,  
  f2: lone  e2,  
  f3: some  e3,  
  f4: set   e4  
}  
(all a: A | a.fn : m e)
```

```
sig A {f, g: e}  
two fields with same constraints
```

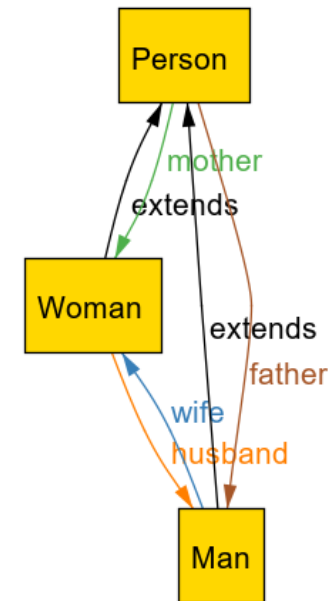
```
sig A {f: e1 m -> n e2}  
(f: A -> (e1 m -> n e2)) or  
(all a: A | a.f: e1 m -> n e2)
```

```
sig Book {  
  names: set Name,  
  addrs: names -> Addr  
}  
dependent fields  
(all b: Book | b.addrs: b.names -> Addr)
```

Langage Alloy

Fields

```
abstract sig Person {  
  father: lone Man,  
  mother: lone Woman  
}  
  
sig Man extends Person {  
  wife: lone Woman  
}  
  
sig Woman extends Person {  
  husband: lone Man  
}
```



- Les pères sont des hommes. Chaque personne a au plus un père.
- Les mères sont des femmes. Chaque personne a au plus une mère.
- Les épouses sont des femmes. Chaque homme a au plus une épouse.
- Les maris sont des hommes. Chaque femme a au plus un mari.

Langage Alloy

Macros

- Les macros paramétrées définissent des termes ou des formules.
- Elles peuvent :
 - Être nommées et réutilisées dans différents contextes.
 - Avoir aucun ou plusieurs paramètres.
 - Être utile pour factoriser certains patterns communs.
- Deux types de macros : **functions** et **predicates**.

Langage Alloy

Functions

- Les **functions** sont utilisées pour définir l'expression de différents ensembles.

```
fun f[x1: e1, ..., xn: en] : e { E }
```

*functions are named expression with declaration
parameters and a declaration expression as a result
invoked by providing an expression for each parameter*

```
sig Name, Addr {}  
sig Book {  
  addr: Name -> Addr  
}  
  
fun lookup[b: Book, n: Name] : set Addr {  
  b.addr[n]  
}  
  
fact everyNameMapped {  
  all b: Book, n: Name | some lookup[b, n]  
}
```

Langage Alloy

Predicates

- Les **predicates** définissent des formules utilisées dans différents contextes. Ils sont souvent employés pour définir les **facts**.

```
pred p[x1: e1, ..., xn: en] { F }
```

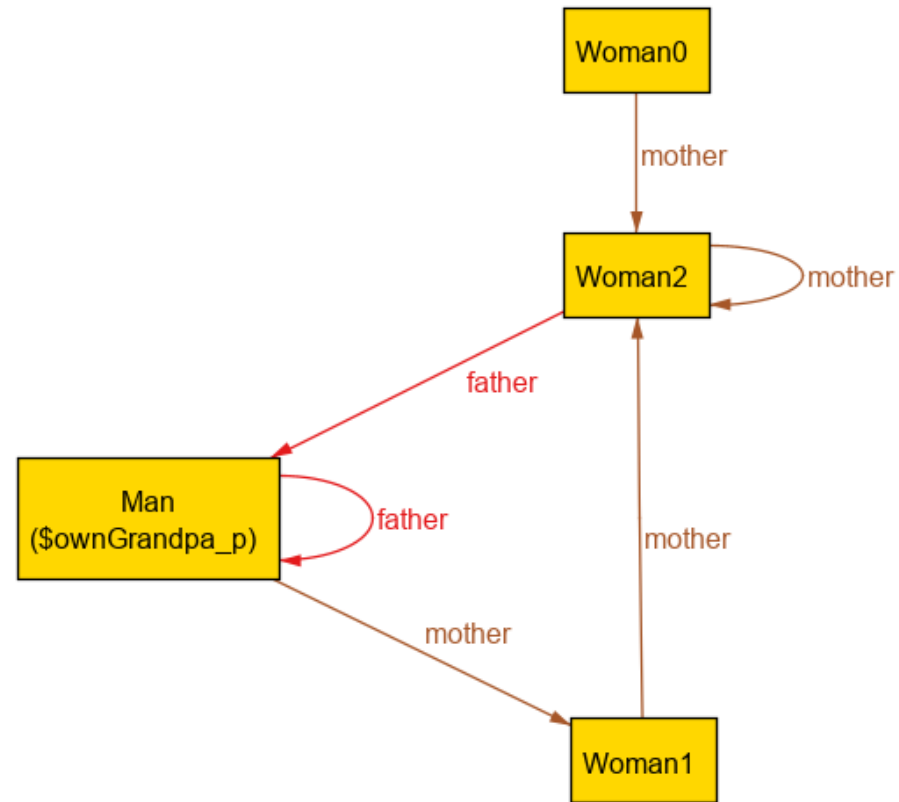
named formula with declaration parameters

```
sig Name, Addr {}  
sig Book {  
  addr: Name -> Addr  
}  
  
pred contains[b: Book, n: Name, d: Addr] {  
  n->d in b.addr  
}  
  
fact everyNameMapped {  
  all b: Book, n: Name |  
    some d: Addr | contains[b, n, a]  
}
```

Langage Alloy

Macros

```
fun grandpas[p: Person] : set Person {  
  p.(mother + father).father  
}  
  
pred ownGrandpa[p: Person] {  
  p in grandpas[p]  
}
```



Langage Alloy

Facts

Les **facts** sont les propriétés (= contraintes) de notre modèle.

```
fact {  
  no p: Person |  
    p in p.^(mother + father)  
  wife = ~husband  
}
```

- Personne ne peut être son propre ancêtre.
- Si X est le mari de Y alors Y est la femme de X.

```
sig Host {}  
sig Link {from, to: Host}  
  
fact {all x: Link | x.from != x.to}  
no links from a host to itself  
  
fact noSelfLinks {all x: Link | x.from != x.to}  
same as above  
  
sig Link {from, to: Host} {from != to}  
same as above, with implicit 'this.'
```

Langage Alloy

Assertions - Check

- Les **assertions** sont des hypothèses sur notre modèle. Elles peuvent être testées à l'aide de **check**.

```
fact {  
  no p: Person | p in p.^(mother + father)  
  wife = ~husband  
}  
  
assert noSelfFather {  
  no m: Man | m = m.father  
}  
  
check noSelfFather
```

Executing "Check noSelfFather"

```
Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20  
495 vars. 45 primary vars. 731 clauses. 16ms.  
No counterexample found. Assertion may be valid. 0ms.
```

EXEMPLE EN LIEN AVEC
LE PROJET

Exemple

Définition d'un Segment

```
sig Point{
}

sig Segment{
  start: one Point,
  end: one Point
}

fact {
  all s: Segment |
    s.start != s.end
}

fact noTwoSeqWithSameStartEnd {
  no disj x1, x2: Segment |
    x1.start = x2.start and x1.end = x2.end
}
```

Exemple

Définition d'une Séquence

```
sig Sequence {  
    startSeg: one Segment,  
    endSeg: one Segment,  
    intermediateSeg: set Segment  
}  
{  
    no x: intermediateSeg | startSeg = x or endSeg = x  
    lone x: intermediateSeg | startSeg.end = x.start  
    lone x: intermediateSeg | endSeg.start = x.end  
    no x: intermediateSeg | startSeg.start = x.end  
    no x: intermediateSeg | startSeg.end = x.start  
    all x: intermediateSeg | x.end = endSeg.start or  
        one x1: intermediateSeg | x.end = x1.start  
    all x: intermediateSeg | x.start = startSeg.end or  
        one x1: intermediateSeg | x.start = x1.end  
    #intermediateSeg = 0 implies (startSeg.end = endSeg.start)  
}
```

Exemple

Définition d'un Cycle

```
sig Cycle extends Sequence{  
}  
{  
    startSeg.start = endSeg.end  
}
```

Exemple

Predicats

```
pred intersect[s1, s2: Segment] {  
    s1 != s2 and  
    (s1.start = s2.start or s1.start = s2.end or  
    s1.end = s2.end or s1.end = s2.start)  
}  
  
pred SequenceIntersec[seq1, seq2: Sequence] {  
    seq1 != seq2 and  
    (some s1: seq1.intermediateSeg, s2: seq2.intermediateSeg |  
        intersect[s1, s2])  
}
```

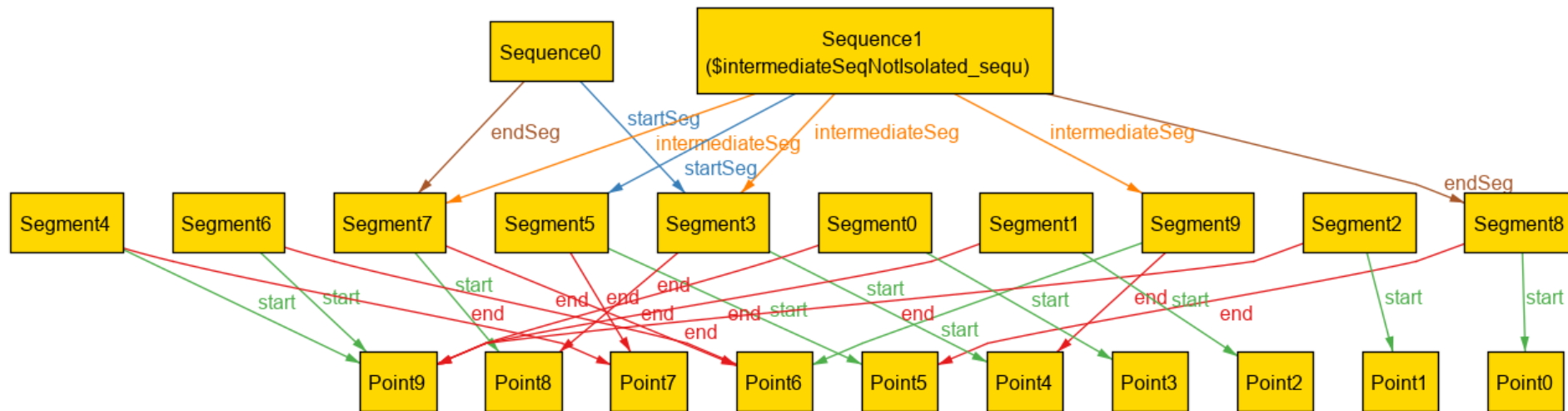
Example

Assertions

```
assert intermediateSeqNotIsolated {  
    all sequ: Sequence | #(sequ.intermediateSeg) > 0 implies  
    some i: Segment | i in sequ.intermediateSeg and  
    sequ.startSeg.end = i.start  
}  
  
check intermediateSeqNotIsolated for 10
```

Exemple

Résultats



REFERENCES

Pour aller plus loin...

- <https://alloytools.org/>
- <https://homepage.cs.uiowa.edu/~tinelli/classes/181/Fall17/Notes/>

