Generating and solving "Ricochet Robot" games with Alloy

Mathieu SCHEPENS
mathieu.schepens@etudiant.univ-lille1.fr
Lille 1 - Master 2 IAGL

Thomas VINCENT
thomas.vincent@etudiant.univ-lille1.fr
Lille 1 - Master 2 IAGL

January 13, 2015
## Contents

1 Introduction ...... 3

2 Game description ...... 4

3 Created UI ...... 5

4 Creating Alloy models for one robot from static games ...... 9
   4.1 Two-dimensional array .......... 9
      4.1.1 Board game representation .......... 9
      4.1.2 Model specifications .......... 10
   4.2 Linked reachable cells .......... 11
      4.2.1 Board game representation .......... 11
      4.2.2 Model specifications .......... 12
   4.3 Two-dimensional array with multiple robots .......... 13
      4.3.1 Board game representation .......... 13
      4.3.2 Model specifications .......... 13
   4.4 Linked reachable cells with multiple robots .......... 15
      4.4.1 Board game representation .......... 15
      4.4.2 Model specifications .......... 15

5 Generating or creating games using Alloy models to verify them ...... 17

6 Evaluation ...... 18
   6.1 Performances .......... 18
   6.2 Validation .......... 20

7 Conclusion ...... 21
1 Introduction

Appeared in 1999, Ricochet Robot is a puzzle board game which was declined in numerous versions. Its goal is to move some robots on a board, and to bring them from a begin location to a final location. A robot can only move by doing straight lines, and only stops when it encounters an obstacle like a border, a corner, an other robot, or an obstacle. An accurate description is present in this paper.

The main goal of this paper is to find a way to modelise and solve a game with Alloy. Once this goal will be reached, an other one is to generate various game which will be valid, a valid game means that there is at least one solution for this game. A final goal is to create and modelise a graphic interface (UI) while others goals are allowing to use new features. This UI has to allow a player to enter or generate a game, and to print a possible solution of the game entered.

Finally, this paper presents performances of created models depending on which model, on which solver, and on the number of processors used.
2 Game description

This part is present for describing more accurately how the board game "Ricochet robots" works. The board game is created from four quarters of the game and from four robots. The board game has a width and a height of 16 cells. Each quarter contains some final cells each robot has to reach. When beginning the game, the final cell of each robot is defined by a card took randomly by a player.

Each robot can move only in straight lines, and move to a direction until it encounters a wall or an other robot. The robot reaches its final cell only if it stops on this cell (while encountering a wall or an other robot).

Ricochet robots can have from 1 to an infinite number of players. When a player has a solution to the game for four robots and four final cells, he says to the others players the number of moves he found for solving the game and starts a timer. Other players has the time of the timer to find a best number of moves. A player wins when other players didn’t find a best solution during the time of the timer.

Figure 1: Example of real game with a solution

In the game presented, a solution is showed for the blue robot to reach its final position which is here a blue circle.
3 Created UI

In this part, the created UI will be presented. As a first view of this UI, the following figure is a good example of how to represent a game. Each cell is represented by a word, which is "Empty" when there is nothing in the cell, "Robot" when a robot is in the cell, and "Arrivee" when a cell is the final cell of a robot. A number will be added after "Robot" when multiples robots are defined. Each border is represented by a line, which is black where a border is set between two cells, or white if no border is set.

Figure 2: Example of a game without border set (except edges of the board game)
As an example, a possible static or generated game will have to be like the following game: the "Finish" cell is reachable for the "Robot" by doing some moves (right, top, left, top, right).

Figure 3: Example of static game

When a game has a solution, a popup will be shown to the user indicating a solution exists:

Figure 4: Popup shown when a game has a solution
This user can, now, show the solution if he wants to see it. For the example of an empty game as in the Figure 2, the solution shown will be the following:

![Figure 5: Popup shown when a game solution is asked](image)

There is also some games which won’t have any solution, because the final cell of a robot is not reachable. The following example shows a bad game: the finish cell is in the last column of the board game, but there is no way for the robot to reach this column, it means there is no solution for the robot to reach its final cell.

![Figure 6: Example of a game without solution](image)
When the user asks for a solution, the following popup will be shown for giving him the information that no solution was found for the current game.

![Popup shown when a game hasn’t a solution](image)

Figure 7: Popup shown when a game hasn’t a solution
4 Creating Alloy models for one robot from static games

In this section, static games will be created and used for creating Alloy models. Games of 5 cells height and width are used as static games for having quick solvers. In this part, only one robot is taken for having a first view of game representations.

A static game is created by setting in a same Alloy model a board game representation, a way to find a solution, a board game initialisation and a start condition : for example, start problem resolution with a maximum of 25 cells, 20 moves, 1 robot and 20 cell borders.

For creating those models, two ways were used in Alloy :
- Using a two-dimensional array which contains every cells of the game
- Using only cells with a border, and linking them with every cells reachable from this one.

Each following section will show an Alloy model without the game generation part. It means that only rules of the board game representation and the way to find the solution will be described.

4.1 Two-dimensional array

4.1.1 Board game representation

In this case, the game is represented by a two-dimensional array of cell. Each cell has 4 borders (top, left, right, bottom), and a position in the grid, set by an x and y position.

```alloy
abstract sig Border{}
one sig BorderTop,BorderRight, BorderBottom, BorderLeft extends Border{}
abstract sig Cell{
  x:Int, y:Int,
  bordertop: lone BorderTop, borderbottom: lone BorderBottom,
  borderleft: lone BorderLeft, borderright: lone BorderRight
}
```

There is a cell where the robot has to finish his path.
The game contains an array structure, created from the cell list, with borders of the board game and verification of its coherence : if the left border of a cell is set, then the right border of its left cell is set too. It contains a specific cell which is the the location where a robot has to finish, this robot starts from a cell.

```alloy
abstract sig End extends Cell{}
abstract sig Robot{start:Cell}
one sig r1 extends Robot{}
fact Rowstructure{
  (c11+c12+c13+c14+c15).y = 1 and (c21+c22+c23+c24+c25).y = 2 and (c31+c32+c33+c34 +c35).y = 3 and (c41+c42+c43+c44+c45).y = 4 and (c51+c52+c53+c54+c55).y = 5
}
fact Colstructure{
  (c11+c21+c31+c41+c51).x = 1 and (c12+c22+c32+c42+c52).x = 2 and (c13+c23+c33+c43 +c53).x = 3 and (c14+c24+c34+c44+c54).x = 4 and (c15+c25+c35+c45+c55).x = 5
}
fact startrobot { r1.start in c51 }
```
4.1.2 Model specifications

The model is based on specific moves. A move can be oriented to the left, to the right, to the top or the bottom. The solution is represented by a list of moves. Each move has a previous move, except for the first one, a cell where the robot is located after this move, and an orientation.

```plaintext
abstract sig Move{ at:Cell, prev:lone Move}
one sig MoveStart extends Move{}
sig MoveTop, MoveRight, MoveBottom, MoveLeft extends Move{}
```

The location of a robot after a move is defined by the following rule: if the move is oriented to the left, take the first cell in the array to the left of the current cell which has a left border.

```plaintext
pred bestmovetop[m:MoveTop]{
  no c:Cell | m.at.y<c.y && m.prev.at.y>c.y && c.x=m.at.x && #c.bordertop = 1
}
Same for bottom, left and right
fact moveTop{
  all m:MoveTop | m.prev.at.y > m.at.y && m.at.x = m.prev.at.x &&
    #m.at.bordertop=1 && bestmovetop[m]
}
```

The first specification is that the same move is not present two times during the way to the end cell. It avoids cycles where the same way is took several times.

The second specification is that the robot is never two times in the same cell during its road to the end cell. This specification avoids cycles where the same cell is reached from different ways.

```plaintext
fact StartMove{
  one m:MoveStart | one r:Robot | #m.prev=0 && m.at=r.start
}
fact ListMove{
  all m:Move-MoveStart | #m.prev=1 && m not in m.ˆ prev
}
fact uniqprevMove{
  no disj m1, m2:Move | m1.prev=m2.prev
}
fact EndMove{
  one m:Move | one c:End | m.at=c && !(some m2:Move | m2.prev=m)
}
fact uniqcell{
  no disj m1, m2:Move | m1.at=m2.at
}
```
4.2 Linked reachable cells

4.2.1 Board game representation

In this case, the game is represented by a list of reachable cells. It means that each represented cell is reachable from an other. As an example, the game of the figure 2 has only its corners as reachable cells.

For representing this game by some Alloy rules: each cell is an object which has a position on the board game (x and y), and is linked to a maximum of four cells: one to its left, its right, its top and its bottom.

```alloy
definition Cell {
    x: Int, y: Int,
    linktop: one Cell, linkright: one Cell,
    linkbottom: one Cell, linkleft: one Cell
}
```

The game contains only its reachable cells, including the end cell and the robot cell in a grid representation. The robot has a cell where its start from.

```alloy
definition End extends Cell {
}
definition c11 extends Cell {
}
definition c51, c55 extends Cell {
}
definition c15 extends End {
}
definition Rowstructure{
    (c11+c15).y = 1 and (c51+c55).y = 5
}
definition Colstructure{
    (c11+c51).x = 1 and (c15+c55).x = 5
}
definition Robot{
    start: Cell // Start position of the robot
}
definition r1 extends Robot {
}
definition startrobot{
    r1.start in c51
}
```
4.2.2 Model specifications

It has the same main specifications than the previous model:
This model is also base on specific moves, which can be oriented to the left, the right, the top or the bottom. The solution is also represented as a list of moves. Each move has a previous move, except for the first one, a cell where the robot is located after this move, and an orientation.

abstract sig Move{
    at:Cell, // Arrived position
    prev:lone Move // Previous movement
}
one sig MoveStart extends Move{}
sig MoveTop,MoveRight,MoveBottom,MoveLeft extends Move{}

fact StartMove{
    one m:MoveStart | one r:Robot | #m.prev=0 && m.at=r.start
}
fact EndMove{
    one m:Move | one c:End | m.at=c && !(some m2:Move | m2.prev=m)
}
fact ListMove{
    all m:Move-MoveStart | #m.prev=1 && m not in m.prev
}

The first specification is that the same move is not present two times during the way to the end cell. It avoids cycles where the same way is took several times.

The second specification is that the robot is never two times in the same cell during its road to the end cell. This specification avoids cycles where the same cell is reached from different ways.

fact uniqprevMove{
    no disj m1, m2:Move | m1.prev=m2.prev
}
fact uniqcell{
    no disj m1, m2:Move | m1.at=m2.at
}
fact moveTop{
    all m:MoveTop | m.prev.at.linktop = m.at
}
// Same for left, right, bottom

The main difference is that the position of a robot after a move is not computed by a rule depending on border, but is directly defined by a rule which sets the reachable cells from the current cell. This difference allows the model to not compute each move destination cell.
4.3 Two-dimensional array with multiple robots

4.3.1 Board game representation

In this model, the board game is described as the first model. The main difference with this model is the number of robots: this model allows multiple robots whereas the previous didn’t. For this work, it was decided that there is only one end for every robot. When a robot reaches the end, the game is ended. The main different rule about the game description is that robots are not beginning on the same cell, and they are stopping where encountering a cell where a robot is already placed.

```plaintext
fact startrobot{ r1.start = c51 && r2.start = c25 && r3.start = c45 }
```

4.3.2 Model specifications

The model rules are pretty the same than the version with only one robot. A move is now containing a robot, for knowing which robot is moving, and an index, which is the index of this move for allowing only one move at a time.

```plaintext
abstract sig Move{
    indexGlobal:Int, robot:Robot,
    at:Cell, prev:lone Move
}
```

A move has an index which equals the index of the last move plus one. When a robot is moved, no move has to be done at the same index. When a robot encounters another robot, it stays on the cell before the collision with this robot.

```plaintext
pred norecentmove[r:Robot,m1:Move,m2:Move]{
    no m:Move | r=m.robot && m1.indexGlobal>m.indexGlobal &&
    m2.indexGlobal<m.indexGlobal
}
pred bestcollisiontop[m1:Move,m2:Move]{
    one m:Move | m.robot!=m1.robot && m.robot!=m2.robot &&
    norecentmove[m.robot,m,m1] && m.at.x=m1.at.x
    && m.at.y=m1.at.y.minus[1] && m2.at.x=m.at.x &&
    m2.at.y>m.at.y && !bestcollisiontop[m,m1]
}
//same for left, right, bottom
pred collisiontop[m1:Move,m2:Move]{
    one m:Move | m.robot!=m1.robot &&
    norecentmove[m.robot,m1,m] &&
    m.at.x=m1.at.x && m.at.y=m1.at.y.minus[1] &&
    m2.at.x=m.at.x &&
    m2.at.y>m.at.y && !bestcollisiontop[m,m1]
}
fact moveTop{
    all m:MoveTop | one m2:Move |
    m.robot=m2.robot &&
    m2.indexGlobal<m.indexGlobal &&
    norecentmove[m.robot,m2]
    && m2.at.y > m.at.y && m.at.x = m2.at.x &&
    no m2.at.bordertop
    && ((#m.at.bordertop=1 && bestmovetop[m] &&
    !collisiontop[m,m2]) ||
    collisiontop[m,m2])
}
//same for left, right, bottom
```
Other rules of the same model for single robot are affected by the multiple robots rules:

```plaintext
fact firstmove{
  one m:MoveStart | m.indexGlobal=1
}

fact StartMove{
  all m:MoveStart | one r:Robot | m.at=r.start && m.robot=r
}

fact ListMove{
  all m:Move | m not in m.prev && m.indexGlobal = m.prev.indexGlobal.plus[1]
}

fact uniqprevMove{
  no disj m1, m2:Move | m1.prev=m2.prev
}

fact EndMove{
  one m:Move | m.at in End && !(some m2:Move | m2.prev=m)
}

fact moveStartRobot{
  all m:MoveStart | m.prev.robot != m.robot
}

pred bestmovetop[m:MoveTop]{
  no c:Cell | m.at.y<c.y && m.prev.at.y>c.y && c.x=m.at.x && #c.bordertop = 1
}

// same for left, right, bottom

fact uniqcell{
  no disj m1, m2:Move | m1.at=m2.at
}

A last big rule is restricting moves to assume coherence of the found solution:

```plaintext
fact move{
  all m:MoveTop | one m2:Move-MoveTop |
  m.robot=m2.robot && m2.indexGlobal<m.indexGlobal && norecentmove[m.robot,m,m2] && m2.at.y > m.at.y && m.at.x = m2.at.x && no m2.at.bordertop && ((#m.at.bordertop=1 && bestmovetop[m] && !collisiontop[m,m2]) || collisiontop[m,m2])
  // Same for left, right, bottom
}
```
4.4 Linked reachable cells with multiple robots

4.4.1 Board game representation

In this model, the game representation is almost the same than in the single robot model. The main difference is that a reachable cell can be added when a robot is moved: for the example presented Figure 2, if a robot is located in the top left corner, it means the robot of the bottom left corner can reach the cell of second line and first column, which was unreachable without any robot. The way of adding a cell like this is presented by the following line in Alloy:

\[
\text{sig addingCell extends Cell{} }
\]

By the way, allowing cells to be created dynamically means a cell can be duplicated when its reachable from two different ways. In Alloy, it’s possible to avoid this duplication with this rule:

\[
\text{fact nosamecell}\{
\quad \text{no disj c1,c2:Cell | c1.x=c2.x && c1.y=c2.y}
\}
\]

4.4.2 Model specifications

As in the previous model, each move contains now a robot and an index, for blocking multiple moves at the same time, and detecting which robot is moving at which time. Collisions between robots are almost set like in the previous model:

\[
\text{pred bestcollisiontop|m1:Move,m2:Move|}
\quad \text{no m:Move | m.robot!=m1.robot && m.robot!=m2.robot && norecentmove[m.robot,m,m1] && m.at.x=m1.at.x && m2.at.y>m.at.y \&\& bestcollisiontop[m,m1]}
\]

//Same for left, right, bottom

\[
\text{pred collisiontop|m1:Move,m2:Move|}
\quad \text{one m:Move | m.robot!=m1.robot && norecentmove[m.robot,m1,m] \&\& m.at.x=m1.at.x \&\& m.at.y=m1.at.y.minus[1] \&\& m2.at.x=m.at.x \&\& m2.at.linktop.y<m.at.y \&\& m2.at.y>m.at.y \&\& bestcollisiontop[m,m1]}
\]

//Same for left, right, bottom

\[
\text{fact moveTop}\{
\quad \text{all m:MoveTop | one m2:Move-MoveTop | let col=collisiontop[m,m2] |}
\quad \text{m.robot=m2.robot \&\& m2.indexGlobal<m.indexGlobal \&\& norecentmove[m.robot,m,m2] \&\&}
\]

//Same for left, right, bottom
The most important added rules are the following: Linking a created cell with an existing cell, and considering created cell in moves is now done by the following rules:

```
fact linkcellvertical{
    all c1:addingCell | one c2:Cell-addingCell |
    c1.x=c2.x && c1.y>c2.y && c1.y>c2.linkbottom.y &&
    c1.linktop=c2 && c1.linkbottom=c2.linkbottom
    // Same for horizontal
}

fact addingCellinmove{
    all c:addingCell | one m:Move | m.at=c
}
```
5 Generating or creating games using Alloy models to verify them

Now models are ready, it is now doable to generate games by adding a new rule. For generating a game which has a solution, the easiest way is to put in the same file the game, the model, the begin and the end cells of each robot, and to start it with a rule which says how much border the game can had. The following extract of start instruction shows how have a doable game with a maximum border number of 4:

\[
\text{run\{} \text{ for 4 Move, 25 Cell, 4 Border}\]

Generating a game is also doable by setting more robots or more end cells for robots. It doesn’t appears in the "run" rule because of the initialisation of start cell of each robot.

Note that for the game definition with reachable cells, the game generation is not doable directly with Alloy because of its conditions. This game definition is more doable for developers which wants to create an optimised model, from an UI and the Alloy java library for example in this paper.

In fact, with the library provided by Alloy community, it is possible to use the alloy model in java. By this way, a graphic interface can be proposed to the user for creating a game, and then check if this game has a solution.
6 Evaluation

6.1 Performances

In this part, performance tests will be done between different models and solvers. It was decided to cut performance tests in two parts. Firstly a comparison of models 1 and 2 for solving static games more or less easy to solve. Using static games allows statistics about "best path" and "used path" in number of moves.

In the following array, four configurations were used: model with array of cells and sat4j solver, model with array of cell and minisat solver, model with reachable cells and sat4j solver, and model with reachable cells and minisat solver.

A configuration can be "LB" for "Low number of borders" and "HB" for "High number of borders". Every tests are used on a board game of 5 columns and 5 lines.

<table>
<thead>
<tr>
<th>Model - Solver</th>
<th>Min nb moves</th>
<th>Nb moves found</th>
<th>Average exec time (ms)</th>
<th>Nb game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array - Sat4J LB</td>
<td>2</td>
<td>2.5</td>
<td>8000</td>
<td>5</td>
</tr>
<tr>
<td>Array - MiniSat LB</td>
<td>2</td>
<td>2</td>
<td>3100</td>
<td>5</td>
</tr>
<tr>
<td>Reachable - Sat4J LB</td>
<td>2</td>
<td>2</td>
<td>260</td>
<td>5</td>
</tr>
<tr>
<td>Reachable - MiniSat LB</td>
<td>2</td>
<td>2</td>
<td>240</td>
<td>5</td>
</tr>
<tr>
<td>Array - Sat4J HB</td>
<td>2</td>
<td>2.2</td>
<td>60000</td>
<td>5</td>
</tr>
<tr>
<td>Array - MiniSat HB</td>
<td>2</td>
<td>2</td>
<td>3800</td>
<td>5</td>
</tr>
<tr>
<td>Reachable - Sat4J HB</td>
<td>2</td>
<td>2</td>
<td>5500</td>
<td>5</td>
</tr>
<tr>
<td>Reachable - MiniSat HB</td>
<td>2</td>
<td>2</td>
<td>6000</td>
<td>5</td>
</tr>
<tr>
<td>Array - Sat4J LB</td>
<td>5</td>
<td>6.5</td>
<td>19500</td>
<td>5</td>
</tr>
<tr>
<td>Array - MiniSat LB</td>
<td>5</td>
<td>5.5</td>
<td>6000</td>
<td>5</td>
</tr>
<tr>
<td>Reachable - Sat4J LB</td>
<td>5</td>
<td>5.2</td>
<td>4000</td>
<td>5</td>
</tr>
<tr>
<td>Reachable - MiniSat LB</td>
<td>5</td>
<td>5.2</td>
<td>2750</td>
<td>5</td>
</tr>
<tr>
<td>Array - Sat4J HB</td>
<td>5</td>
<td>6.5</td>
<td>30000</td>
<td>5</td>
</tr>
<tr>
<td>Array - MiniSat HB</td>
<td>5</td>
<td>5.5</td>
<td>10000</td>
<td>5</td>
</tr>
<tr>
<td>Reachable - Sat4J HB</td>
<td>5</td>
<td>5.2</td>
<td>5000</td>
<td>5</td>
</tr>
<tr>
<td>Reachable - MiniSat HB</td>
<td>5</td>
<td>5.2</td>
<td>4750</td>
<td>5</td>
</tr>
<tr>
<td>Array - Sat4J HB</td>
<td>15</td>
<td>15</td>
<td>60000</td>
<td>5</td>
</tr>
<tr>
<td>Array - MiniSat HB</td>
<td>15</td>
<td>15</td>
<td>30000</td>
<td>5</td>
</tr>
<tr>
<td>Reachable - Sat4J HB</td>
<td>15</td>
<td>15</td>
<td>35000</td>
<td>5</td>
</tr>
<tr>
<td>Reachable - MiniSat HB</td>
<td>15</td>
<td>15</td>
<td>25000</td>
<td>5</td>
</tr>
</tbody>
</table>

This array allows to think that the Minisat solver is better for finding solution with the presented models, and that the second modelisation (with reachable cells) is, most of the time, the best one. The first model is faster only where several borders are giving a lot of path for reaching end cell.
Secondly, this part presents a comparison of models 1 and 2 for solving static games more or less easy to solve with multiples robot. Using static games allows statistics about "best path" and "used path" in number of moves.

In the following array, four configurations were used: model with array of cells and sat4j solver, model with array of cell and minisat solver, model with reachable cells and sat4j solver, and model with reachable cells and minisat solver.

A configuration can be "LB" for "Low number of borders" and "HB" for "High number of borders". Every tests are used on a board game of 5 columns and 5 lines, and an average of 5 games is used for doing statistics.

<table>
<thead>
<tr>
<th>Model - Solver</th>
<th>Nb robot</th>
<th>Min nb moves</th>
<th>Nb moves found</th>
<th>Average exec time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array - Sat4J LB</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>80000</td>
</tr>
<tr>
<td>Array - MiniSat LB</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>77000</td>
</tr>
<tr>
<td>Reachable - Sat4J LB</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>60000</td>
</tr>
<tr>
<td>Reachable - MiniSat LB</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>65000</td>
</tr>
<tr>
<td>Array - Sat4J LB</td>
<td>3</td>
<td>4</td>
<td>4.2</td>
<td>90000</td>
</tr>
<tr>
<td>Array - MiniSat LB</td>
<td>3</td>
<td>4</td>
<td>4.1</td>
<td>70000</td>
</tr>
<tr>
<td>Reachable - Sat4J LB</td>
<td>3</td>
<td>4</td>
<td>4.2</td>
<td>80000</td>
</tr>
<tr>
<td>Reachable - MiniSat LB</td>
<td>3</td>
<td>4</td>
<td>4.1</td>
<td>75000</td>
</tr>
</tbody>
</table>

Using the second model for multi robots can have random performances, depending on border and robots number. Statistics were done only with 2 and 3 robots because of the amount of time multiple robots problems are taking to be solved.

This amount of time is caused by the number of possibilities of path for each robot. As an example, "Robot 1 moves to the top, Robot 2 moves to the top" or "Robot 2 moves to the tof, Robot 1 moves to the top" are two different paths for our models, and two times more path to explore for finding a solution.
6.2 Validation

As a validation, the following array shows how each possibility of model-solver reacts for solving a problem. Those statistics are based on same executions than in the previous part:

<table>
<thead>
<tr>
<th>Model - Solver</th>
<th>Nb robot</th>
<th>Min nb moves</th>
<th>Percentage of found solutions in less than 100s (best or not)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array - Sat4J LB</td>
<td>1</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Array - MiniSat LB</td>
<td>1</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Reachable - Sat4J LB</td>
<td>1</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Reachable - MiniSat LB</td>
<td>1</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Array - Sat4J HB</td>
<td>1</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Array - MiniSat HB</td>
<td>1</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Reachable - Sat4J HB</td>
<td>1</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Reachable - MiniSat HB</td>
<td>1</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Array - Sat4J LB</td>
<td>1</td>
<td>5</td>
<td>100%</td>
</tr>
<tr>
<td>Array - MiniSat LB</td>
<td>1</td>
<td>5</td>
<td>100%</td>
</tr>
<tr>
<td>Reachable - Sat4J LB</td>
<td>1</td>
<td>5</td>
<td>100%</td>
</tr>
<tr>
<td>Reachable - MiniSat LB</td>
<td>1</td>
<td>5</td>
<td>100%</td>
</tr>
<tr>
<td>Array - Sat4J HB</td>
<td>1</td>
<td>5</td>
<td>100%</td>
</tr>
<tr>
<td>Array - MiniSat HB</td>
<td>1</td>
<td>5</td>
<td>100%</td>
</tr>
<tr>
<td>Reachable - Sat4J HB</td>
<td>1</td>
<td>5</td>
<td>100%</td>
</tr>
<tr>
<td>Reachable - MiniSat HB</td>
<td>1</td>
<td>5</td>
<td>100%</td>
</tr>
<tr>
<td>Array - Sat4J HB</td>
<td>1</td>
<td>15</td>
<td>100%</td>
</tr>
<tr>
<td>Array - MiniSat HB</td>
<td>1</td>
<td>15</td>
<td>100%</td>
</tr>
<tr>
<td>Reachable - Sat4J HB</td>
<td>1</td>
<td>15</td>
<td>100%</td>
</tr>
<tr>
<td>Reachable - MiniSat HB</td>
<td>1</td>
<td>15</td>
<td>100%</td>
</tr>
<tr>
<td>Array - Sat4J LB</td>
<td>2</td>
<td>3</td>
<td>75%</td>
</tr>
<tr>
<td>Array - MiniSat LB</td>
<td>2</td>
<td>3</td>
<td>75%</td>
</tr>
<tr>
<td>Reachable - Sat4J LB</td>
<td>2</td>
<td>3</td>
<td>50%</td>
</tr>
<tr>
<td>Reachable - MiniSat LB</td>
<td>2</td>
<td>3</td>
<td>50%</td>
</tr>
<tr>
<td>Array - Sat4J LB</td>
<td>3</td>
<td>4</td>
<td>75%</td>
</tr>
<tr>
<td>Array - MiniSat LB</td>
<td>3</td>
<td>4</td>
<td>75%</td>
</tr>
<tr>
<td>Reachable - Sat4J LB</td>
<td>3</td>
<td>4</td>
<td>50%</td>
</tr>
<tr>
<td>Reachable - MiniSat LB</td>
<td>3</td>
<td>4</td>
<td>50%</td>
</tr>
</tbody>
</table>

As a performance validation, models are finding a solution in most of the cases for only one robot. But with more than one robot, models and solvers are not finding a solution at each occurrence.
7 Conclusion

As a conclusion, two ways of seeing the game were seen: with listing every cells, or with listing only reachable cells. It allowed us to see two pretty similars Alloy models, with only few rules more for the game definition with every cells.

A game generator was defined in alloy for creating solvable games, and to propose them to a player which wants to see a different game via an UI. This UI allows player to create its own game, to see if this game has a solution depending on a representation (reachable cells or cells array) and to show this solution.

Interesting performances were extracted, even if they could be better without all communications and conversions between Alloy and Java.

Each solver has good performances, even if the multi robots models can be optimised for finding a solution quickly and in all cases of map, instead of its 50-75% found results in less than 100s.

This work was, for us, a really interesting way to study a new language (Alloy) and its API for Java. We learnt a lot of things, and it allowed us to modelise a board game problem in a computed way.

As futures work, it’s believable to have best solvers for multi robots problems, to have best performances with all conversions/communications between Alloy and Java, and finally to modelise the truth ricochet robot board game: with 16 cells height and width, and an end for each robot.