Spectra Example:
Moving Obstacle Evasion Problem

Matan Yossef

Tel Aviv University

Abstract. The moving obstacle evasion problem is a two player game in which a robot is trying to avoid an obstacle over a bidimensional grid. We chose to model the solution as a safety game where the robot represents the system player. This report presents a formal specification of the problem in the Spectra language. In addition to assumptions and guarantees, the specification demonstrates the use of Spectra defines, predicates, and counters.

1 Introduction

The moving obstacle evasion problem is a two player game in which a robot is trying to avoid an obstacle over a bidimensional grid [1]. The context game consists of a bidimensional grid of size $m \times m$, a robot of size 1×1, initially placed in the upper left corner of the grid, and an obstacle of size 2×2, initially placed in the lower right corner of the grid. In the basic setting, the robot and the obstacle can move by at most one cell in the x dimension and by at most one cell in the y dimension. To give the robot some leeway, the robot can move two steps for every one step of the obstacle. As the obstacle tries to catch the robot, the robot plays a safety game of avoiding the obstacle.

The problem is to compute a behavior (formally, a strategy) for the robot such that the obstacle will never be able to catch it.

We present a Spectra [2] specification for the problem and a simulation of its solution. All related files are available from `http://smlab.cs.tau.ac.il/syntech/`.

2 The Basic Specification

We model the problem as a parametric specification that depends on the size of the grid represented by `DIM_SIZE`.

[1] A previous formulation of the problem appears in the git repository of the Slugs synthesizer `https://github.com/VerifiableRobotics/slugs`

```
spec MovingObstacle

define DIM_SIZE := 6;

env Int(1..(DIM_SIZE-1))[2] obstacle;
env boolean obsWait;
env boolean isObstacleTurn;
define isRobotTurn := !isObstacleTurn;

sys Int(1..DIM_SIZE)[2] robot;
```

Listing 1.1. Spectra specification of the moving obstacle evasion problem: defines and variables

```
asm initiallyObstacleAtLowerRightCorner:
(obstacle[0] = DIM_SIZE - 1) & (obstacle[1] = DIM_SIZE - 1);

gar initiallyRobotAtUpperLeftCorner:
robot[0] = 1 & robot[1] = 1;
```

Listing 1.2. Spectra specification of the moving obstacle evasion problem: initial state assumptions and guarantees

The obstacle location, which is represented by the `obstacle` variable, is controlled by the environment. The robot location, which is represented by the `robot` variable, is controlled by the system. The robot tries to evade the obstacle. The locations are represented by an array of two indices, one for each dimension of the grid. The obstacle is of size 2×2 and its location is determined by the location of its upper left corner (therefore its indices range between 1 and `DIM_SIZE` - 1).

At each step, it is either the robot's turn to move or the obstacle's turn to move. After each step of the obstacle, the robot performs two consecutive steps The turn is specified by the environment controlled Boolean variable `isObstacleTurn`. The variable is followed by a define made for convenience, which indicates if it's the robot's turn. The environment variable `obsWait` determines whether the obstacle moves during the next step or has to wait for another step. List. 1.1 shows the variables and defines in the specification.

Initially, the robot is at location $\langle 1, 1 \rangle$ and the obstacle is at location $\langle \texttt{DIM\_SIZE} - 1, \texttt{DIM\_SIZE} - 1 \rangle$. See the assumption and guarantee in List. 1.2.

The obstacle is the first to play. After each obstacle turn, the obstacle has to wait two turns before it plays again. See List. 1.3.

We now have to define constraints on the movement of the robot and the obstacle. We assume the obstacle moves only when it is its turn. We also constrain the obstacle to move by at most one cell in each dimension in each turn. Similar guarantees are defined for the movement of the robot. Notice the usage of

```
asm initiallyObsWaitTrue:
obsWait;

asm initiallyObstacleTurn:
isObstacleTurn;

asm turnSwitches:
G ((isObstacleTurn | obsWait)->next(isRobotTurn)) &
  ((isRobotTurn & !obsWait)->next(isObstacleTurn));

asm obswaitSwitches:
G (isRobotTurn->(next(obsWait) = !obsWait)) &
  (isObstacleTurn->(next(obsWait) = obsWait));
```

Listing 1.3. Spectra specification of the moving obstacle evasion problem: turn state assumptions

predicates, which define movement on one dimension, to make the specification clearer. See List. 1.4.

Finally and most importantly, the system, which is represented by the robot, guarantees that the robot will never collide with the obstacle, i.e., will not be located in any of the four cells that the obstacle covers. See List. 1.5.

3 Allowing Glitches in Obstacle Movements

To spice the problem up a little bit, we allow glitches in the obstacle movements. Specifically, for a finite number of times, defined by `NUM_OF_GLITCHES`, the obstacle can violate the basic setup and move after one step of the robot rather than only after two steps (as in the basic problem).

To formalize, we define two environment variables that will specify the glitches behavior: a Spectra counter `glitches`, which counts the number of glitches, and a Boolean variable `isGlitch`, which specifies whether the obstacle is currently using a glitch, i.e., whether the obstacle will play immediately after one turn of the robot. We assume that the `glitches` counter counts from 0 to `NUM_OF_GLITCHES`, and increases by 1 following each glitch. See List. 1.6.

The obstacle can only use the glitch during its turn. We assume that there is no glitch during the first step, in order to avoid an initial deadlock. See List. 1.7.

Finally, the glitch effect takes place in the switching of `obsWait`. When the obstacle uses a glitch, `obsWait` switches to false immediately rather than after

```
asm obstacleDoesNotMoveAtRobotTurn:
G isRobotTurn ->
  (next(obstacle[0]) = obstacle[0] &
   next(obstacle[1]) = obstacle[1]);

gar robotDoesNotMoveAtObstacleTurn:
G isObstacleTurn ->
  (next(robot[0]) = robot[0] &
   next(robot[1]) = robot[1]);

predicate moveRobot(Int(1..DIM_SIZE) pos):
  pos+1 = next(pos) | pos   = next(pos) | pos-1 = next(pos);

predicate moveObstacle(Int(1..(DIM_SIZE-1)) pos):
  pos+1 = next(pos) | pos   = next(pos) | pos-1 = next(pos);

asm obstacleMovesAtMostOne:
G moveObstacle(obstacle[0]) & moveObstacle(obstacle[1]);

gar robotMovesAtMostOne:
G moveRobot(robot[0]) & moveRobot(robot[1]);
```

Listing 1.4. Spectra specification of the moving obstacle evasion problem: defining the movement of the players

```
gar robotAvoidsObstacle:
G (robot[0] != obstacle[0] | robot[1] != obstacle[1]) &
(robot[0] != obstacle[0] + 1 | robot[1] != obstacle[1]) &
(robot[0] != obstacle[0] | robot[1] != obstacle[1] + 1) &
(robot[0] != obstacle[0] + 1 | robot[1] != obstacle[1] + 1);
```

Listing 1.5. Spectra specification of the moving obstacle evasion problem: robot evasion gurantee

one turn of the robot. This shortens the obstacle's waiting from two steps to one. See List. 1.8.

4 Using the Specification

In the basic setup, realizabilty is rather simple. The specification is realizable, i.e., the robot can successfully avoid the obstacle, for every grid of size 6×6 or more.

Allowing glitches makes the problem more complicated. Table 1 shows some realizability checking results. For any fixed dimension size, when the number of glitches grows, at some point the specification turns from realizable to unrealizable. We report the results around these turning points. We executed the experiments on an ASUS laptop with $4GB$ RAM and Intel Core $i5-7200U$ CPU with a Spectra version dated 23.10.2019, using all heuristics described in [1].

```
define NUM_OF_GLITCHES := 13;

env boolean isGlitch;
define noGlitch := !isGlitch;

counter glitches(0..NUM_OF_GLITCHES) {
  glitches = 0;
  inc: isGlitch;
  overflow: false;
}
```

Listing 1.6. Spectra specification of the moving obstacle evasion problem: glitches' defines and variables

```
asm initiallyNoGlitch:
noGlitch;

asm maxGlitches:
G next(glitches = NUM_OF_GLITCHES -> noGlitch);

asm glitchOnlyWhenObstacleTurn:
G next(isRobotTurn->noGlitch);
```

Listing 1.7. Spectra specification of the moving obstacle evasion problem: glitches' assumptions

4.1 Simulating the Problem

We created an animation using Java Swing to simulate the synthesized controller. The simulation randomizes the movement of the obstacle by choosing at random one of the possible next states in the specification's symbolic controller. These are guaranteed to not violate the assumptions.

Interestingly, the original specification makes the animated obstacle choose irrational moves from time to time (e.g., moves that are not towards the robot). To make the simulation more intuitive, we added an assumption that forces the obstacle to always move towards the robot. See List. 1.9.

One may consider whether such an assumption should appear in the specification of the problem. An important principle is that assumptions or guarantees should be included in a specification only if they are necessary to describe the problem. Specifically, typically, one should not add assumptions or guarantees that describe the solution (i.e., force a specific strategy for the system or the environment). Therefore, an assumption as we showed above can be used for the purpose of simulation, but in general should not be part of the specification.

```
asm obswaitSwitches:
G ((isRobotTurn | isGlitch)->(next(obsWait) = !obsWait)) &
  ((isObstacleTurn & noGlitch)->(next(obsWait) = obsWait));
```

Listing 1.8. Spectra specification of the moving obstacle evasion problem: glitches' effect on turn switching

Table 1. Some realizability checking results

Dim. size	#glitches	Realizable?	Computation time
8	1	Y	77ms
8	2	N	154ms
16	5	Y	0.88s
16	6	N	1.04s
24	9	Y	12.5s
24	10	N	5.0s
32	13	Y	16s
32	14	N	17s
48	21	Y	53s
48	22	N	57s
64	29	Y	3m52s
64	30	N	4m56s

4.2 Additional Analyses

One may want to check whether all assumptions about the obstacle's behavior are necessary for realizability. In order to check that, the engineer can ask Spectra to compute an assumptions core by choosing Spectra Add-ons / Cores / Find an Assumptions Core. This will find a locally minimal subset of assumptions that are sufficient and necessary for realizability. For example, the analysis shows an assumptions core that contains the assumption `initiallyObstacleAtLowerRightCorner` (see List. 1.2). Indeed, if this assumption is commented out, the specification becomes unrealizable, as the environment may choose to initially locate the obstacle such that it touches the robot. This can also be demonstrated by Spectra as follows: we can comment out this assumption, check that the specification is now unrealizable, compute a concrete counter-strategy (Spectra / Synthesize Concrete Counter-Strategy), and observe the deadlock already at the initial state, as expected.

Acknowledgements

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638049, SYNTECH).

```
asm obstacleIsSmart:
G isObstacleTurn ->
  ((obstacle[0] > robot[0] ->
      next(obstacle[0]) = obstacle[0] - 1) &
   (obstacle[0] < robot[0] - 1 ->
      next(obstacle[0]) = obstacle[0] + 1) &
   (obstacle[1] > robot[1] ->
      next(obstacle[1]) = obstacle[1] - 1) &
   (obstacle[1] < robot[1] - 1 ->
      next(obstacle[1]) = obstacle[1] + 1));
```

Listing 1.9. Spectra specification of the moving obstacle evasion problem: obstacle always moves towards the robot

References

1. Firman, E., Maoz, S., Ringert, J.O.: Performance heuristics for GR(1) synthesis and related algorithms. Acta Inf. **57**(1-2), 37–79 (2020). https://doi.org/10.1007/s00236-019-00351-9, https://doi.org/10.1007/s00236-019-00351-9
2. Maoz, S., Ringert, J.O.: Spectra: A specification language for reactive systems. CoRR **abs/1904.06668** (2019), http://arxiv.org/abs/1904.06668

A Complete Specification

```
spec MovingObstacle

define DIM_SIZE := 64;
define NUM_OF_GLITCHES := 30;

env Int(1..(DIM_SIZE-1))[2] obstacle;
env boolean obsWait;

env boolean isGlitch;
define noGlitch := !isGlitch;

counter glitches(0..NUM_OF_GLITCHES) {
  glitches = 0;
  inc: isGlitch;
  overflow: false;
}

env boolean isObstacleTurn;
define isRobotTurn := !isObstacleTurn;

sys Int(1..DIM_SIZE)[2] robot;

asm initiallyObstacleAtLowerRightCorner:
(obstacle[0] = DIM_SIZE - 1) & (obstacle[1] = DIM_SIZE - 1);

asm initiallyObsWaitTrue:
obsWait;

asm initiallyObstacleTurn:
isObstacleTurn;

gar initiallyRobotAtZero:
robot[0] = 1 & robot[1] = 1;

asm initiallyNoGlitch:
noGlitch;

asm maxGlitches:
G next(glitches = NUM_OF_GLITCHES -> noGlitch);
```

```
asm glitchOnlyWhenObstacleTurn:
G next(isRobotTurn->noGlitch);

asm turnSwitches:
G ((isObstacleTurn | obsWait)->next(isRobotTurn)) &
  ((isRobotTurn & !obsWait)->next(isObstacleTurn));

asm obswaitSwitches:
G ((isRobotTurn | isGlitch)->(next(obsWait) = !obsWait)) &
  ((isObstacleTurn & noGlitch)->(next(obsWait) = obsWait));

asm obstacleDoesNotMoveAtRobotTurn:
G isRobotTurn ->
    (next(obstacle[0]) = obstacle[0] &
     next(obstacle[1]) = obstacle[1]);

gar robotDoesNotMoveAtObstacleTurn:
G isObstacleTurn ->
    (next(robot[0]) = robot[0] &
     next(robot[1]) = robot[1]);

predicate moveRobot(Int(1..DIM_SIZE) pos):
  pos+1 = next(pos) |
  pos   = next(pos) |
  pos-1 = next(pos);

predicate moveObstacle(Int(1..(DIM_SIZE-1)) pos):
  pos+1 = next(pos) |
  pos   = next(pos) |
  pos-1 = next(pos);

asm obstacleMovesAtMostOne:
G moveObstacle(obstacle[0]) & moveObstacle(obstacle[1]);

gar robotMovesAtMostOne:
G moveRobot(robot[0]) & moveRobot(robot[1]);

gar robotAvoidsObstacle:
G (robot[0] != obstacle[0] | robot[1] != obstacle[1]) &
(robot[0] != obstacle[0] + 1 | robot[1] != obstacle[1]) &
(robot[0] != obstacle[0] | robot[1] != obstacle[1] + 1) &
(robot[0] != obstacle[0] + 1 | robot[1] != obstacle[1] + 1);
```

Listing 1.10. Complete Spectra specification of the moving obstacle evasion problem