Richard Zimmerman [U16632404] | Alec Braynen [U41628244]

Two Robots Sensing Two Spaces Ahead

Background

Specifications

The Robot Does Not Hit Any Obstacles.

$$A[] \; grid[robotOne.myPosX][robotOne.myPosY] \; != 1$$

$$A[] \; grid[robotTwo.myPosX][robotTwo.myPosY] \; != 1$$

The array represented by the variable grid is a 2-d boolean array with the dimension of p and q where p represents the number of rows in the grid and q represents the number of columns in the grid. A status of false indicates the grid space represent by *s* and *t* in grid[*s*][*t*] indicates the grid square (*s,t*) is not blocked by a static obstacle. When grid[*s*][*t*] is true, this indicates the grid square (*s,t*) is blocked.

The statement above confirms that there is no such trace where the current position of either robot (in effect, any position the robots have traveled) is blocked by a static obstacle.

$$A[] \; (robotOne.myPosX \; != robotTwo.myPosX \; and \; robotOne.myPosY \; != \\ robotTwo.myPosY)$$

The statement above ensures that the two robots never exist in the same place at once by confirming that there are no traces such that the robots x and y coordinates match.

Currently, our model does not satisfy this requirement and allows the robots to be in the same place at the same time. There are certain cases where one robot could enter a space that blocks the other robot from being able to reach the destination. To work around this issue, the robots could use a shared memory space to communicate where each robot currently is and, importantly, organize a "shuffle" operation. This "shuffle" operation could be implemented by allowing the blocking robot to move back a space at a time until the blocked robot was able to move a space that was not immediately blocked by the other robot.

The Robot Does Not Cross any Boundaries.

$$A[] \; robotOne.myPosX <= 0 \; and \; robotOne.myPosX < dimX$$

$$A[] \; robotTwo.myPosX <= 0 \; and \; robotTwo.myPosX < dimX$$

$$A[] \; robotOne.myPosY <= 0 \; and \; robotOne.myPosY < dimX$$

$$A[] \; robotTwo.myPosY <= 0 \; and \; robotTwo.myPosY < dimX$$

In the above statement, the robot's local variables, myPosX and myPosY, that represent the current X and Y coordinates of the robot on the grid (respectively) are validated so that the integer value of X or Y is never greater than

In the above statement, robotOne (the only robot) has local variables myPosX and myPosY that represent the current X and Y coordinates of the robot on the grid, respectively.

The Robot Will Eventually Reach the Destination.

A<> (robotOne.Complete and robotTwo.Complete)

The above statement confirms that for every trace, robotOne will reach the Complete state which is only entered if the robot current position (myPosX and myPosY) matches the coordinates of the destination point.
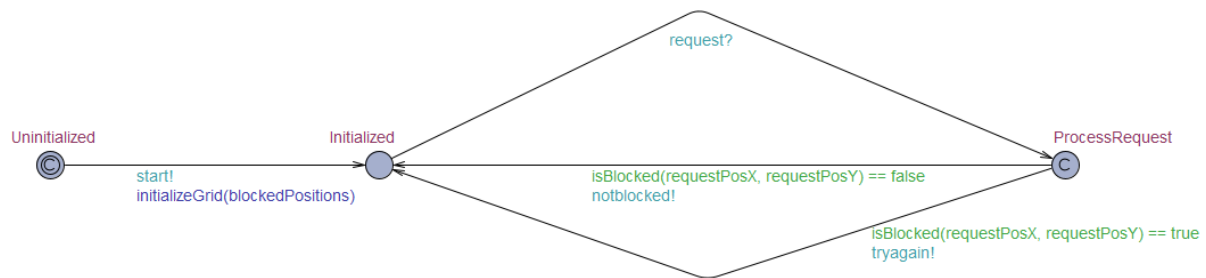
Model Summary

Grid Controller

Figure 1 Model of Grid Controller in UPPAAL

The grid controller starts in a committed state named Uninitialized which, upon system start, is immediately exited and the transition from state Uninitialized to state Initialized is taken. This transition raises a signal on a broadcast channel named *start* and then calls a function that initializes the grid array (the 2d boolean array named *grid* mentioned in *Specifications -> The robot does not hit any obstacles*) which makes sure that the grid's static obstacles are enabled/instantiated.

The grid controller then waits in the Initialized state until it receives a signal on the *request* channel. Upon receiving the request signal, the grid controller enters the committed state ProcessReqeust and in the next "step" of the system, raises a signal on either the *notblocked* or *tryagain* channel which represents whether the grid targeted by the requester is not blocked or blocked, respectively.

Because the Processing state is committed, the grid controller must respond to an active request before any robot in the system is allowed to raise a second request. Effectively, this prevents the robots from moving at the same time to the same space.

directionToTest <= 3
request!
setRequestPosition(1)

SendRequest

isLastPosition(requestPosX, requestPosY)== false
notblocked?
requestPosPreviousX = requestPosX,
requestPosPreviousY = requestPosY

SensingTwo

request!
setRequestPosition(2)

TestMoveTwo

RobotOff

start?

Sensing

tryagain?
directionToTest = directionToTest + 1

isLastPosition(requestPosX, requestPosY)== true
notblocked?
directionToTest = directionToTest + 1

directionToTest > 3

notblocked?
lastPosX = requestPosPreviousX,
lastPosY = requestPosPreviousY,
myPosX = requestPosX,
myPosY = requestPosY,
directionToTest = 0,
myGrid[myPosX][myPosY] = 1

tryagain?
lastPosX = myPosX,
lastPosY = myPosY,
myPosX = requestPosPreviousX,
myPosY = requestPosPreviousY,
directionToTest = 0,
myGrid[myPosX][myPosY] = 1

BackTrack

This State Occurs when a Robot has to backtrack
to a previous position

Move

Complete

myPosX != destinationPosX ||
myPosY != destinationPosY

myPosX == destinationPosX &&
myPosY == destinationPosY

mark backtracked position as blocked

move to last position

update last position

grid[myPosX][myPosY] = true,
myGrid[lastPosX][lastPosY] = true,
tempPosX = myPosX,
tempPosY = myPosY,
myPosX = lastPosX,
myPosY = lastPosY,
lastPosX = tempPosX,
lastPosY = tempPosY,
directionToTest = 0

Figure 2 Model of RobotOne

Please see the document for Part 1 of Project 1 for a description of how the robot works when sensing one-space ahead. The only differences between the 1-sensor robot and 2-sensor robot are the additional states SensingTwo and TestMoveTwo and the corresponding edges.

When the robot receives the first *notblocked* signal from the controller, the robot immediately builds another request to see if it can move two spaces in the same direction as the first successful (notblocked) request. If the target space that is now two spaces away from the robot is free, the robot moves two-spaces to the new target, else the robot only moves one space at a time.