Richard Zimmerman [U16632404] | Alec Braynen [U41628244]

Two Robots Sensing One Space Ahead with Reasonable Timing Specifications

Specifications

The Robots Will Always Reach the Goal.

A<> r1.Complete
A<> r2.Complete

The statements above confirm that robot 1 and robot 2, *r1 and r2 respectively,* will reach the completed state in every possible state space. The complete state is defined as the destination position for each robot, and it is transitioned to when the robots' *myPosition* variable data matches the *destinationPosition's* coordinates on the grid.

Additionally, because the robots always complete their journey, the robots are not stuck indefinitely on any grid square, except the destination square.

The Robot Does Not Hit Any Obstacles.

A[] gridBlocked[r1.myPosition.posx][r1.myPosition.posy] != 1
A[] gridBlocked[r2.myPosition.posx][r2.myPosition.posy] != 1

The array represented by the variable *gridBlocked* 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 represented by *s* and *t* in *gridBlocked*[*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.

The Robots Do Not Collide With Each Other

A[] r1.myPosition != r2.myPosition

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.

The Robots Do Not Cross the Grid Boundaries.

A[] r1.myPosition.posx >= 0 and r1.myPosition.posx < dimX
A[] r1.myPosition.posy >= 0 and r1.myPosition.posy < dimY
A[] r2.myPosition.posx >= 0 and r2.myPosition.posy < dimX
A[] r2.myPosition.posy >= 0 and r2.myPosition.posy < dimY

In the above statement, the robots' local variable, *myPosition*, which represents the current X and Y coordinates of the robots on the grid are confirmed to never go outside of the grid's boundaries. This effectively means, that the robot always stays and traverses on the grid.

The grid's boundaries are defined with an upper limit of *dimX* and *dimY* for the (x, y) positions respectively. The above statements confirm that the robots' positions are never greater than (*dimX*, *dimY*) or less than (0, 0) which are the boundaries of the grid.

The Robot Cannot Stay in a Grid Block Indefinitely

E<> (r1.idleTimer >= 49 and not r1.Complete)
E<> not(r1.idleTimer >= 50 and not r1.Complete)
E<> (r2.idleTimer >= 49 and not r2.Complete)
E<> not(r2.idleTimer >= 50 and not r2.Complete)

The statements above confirm that the finite upper time limit a robot can stay in a position without moving is 49 clock ticks. The statement: *E<>r1.idleTimer >= 49 and not r1.Complete* means that there exists a trace where the robot has not moved for 49 ticks, and it is still traversing the grid. This statement passes as property satisfied. However, the statement below it: *E<> not(r1.idleTimer >= 50 and not r1.Complete),* means there does not exist a trace where the robot is idle for 50 ticks or more. These two statements act as an upper time limit specification, stating that the robot can remain idle for up to 49 ticks as its upper time limit.

The same constraints apply for robot 2 and its relevant statements.

Analysis

Find the minimum time that the robots use to reach their goal

E<> not(initializationClock <= 168 && (r1.Complete && r2.Complete)) == false

E<> (initializationClock <= 169 && (r1.Complete && r2.Complete)) == true

Using UPPAAL and a special clock that starts running at the same moment in time that the system starts and is never reset, *initializationClock*, we can ask if there exists a trace where both robots are in the Complete state and if the *initializationClock* is less than some value. Using this method, we can determine that a trace exists where the initialization clock is 169 and both robots are in the Complete state. However, there does not exist a trace where the *initializationClock* is 168 or less.

Therefore, we can say that the minimum time for the two robots to complete their goal given the current timing specifications is 169 ticks.

Model Summary

Grid Controller

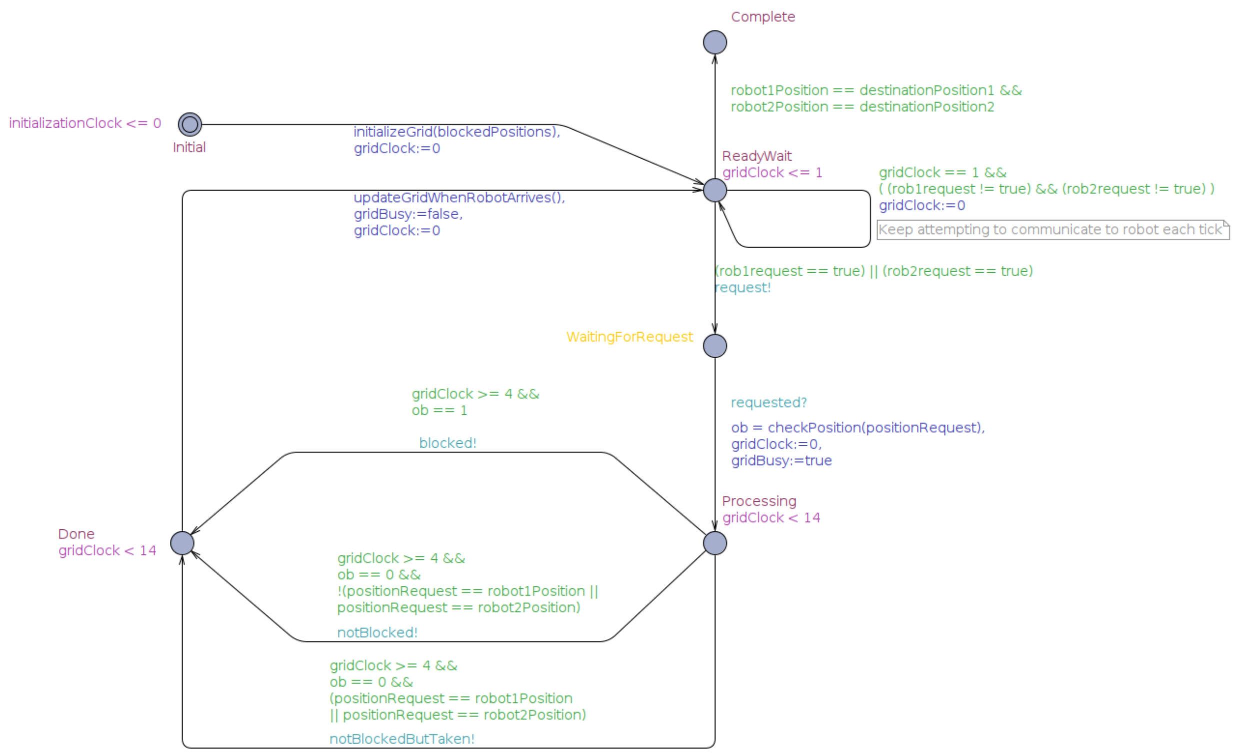

Complete
robot1Position == destinationPosition1 &&
robot2Position == destinationPosition2
initializationClock <= 0
Initial
initializeGrid(blockedPositions),
gridClock:=0
ReadyWait
gridClock <= 1
gridClock == 1 &&
((rob1request != true) && (rob2request != true))
gridClock:=0
Keep attempting to communicate to robot each tick
updateGridWhenRobotArrives(),
gridBusy:=false,
gridClock:=0
(rob1request == true) || (rob2request == true)
request!
WaitingForRequest
requested?
ob = checkPosition(positionRequest),
gridClock:=0,
gridBusy:=true
gridClock >= 4 &&
ob == 1
blocked!
Processing
gridClock < 14
Done
gridClock < 14
gridClock >= 4 &&
ob == 0 &&
!(positionRequest == robot1Position ||
positionRequest == robot2Position)
notBlocked!
gridClock >= 4 &&
ob == 0 &&
(positionRequest == robot1Position
|| positionRequest == robot2Position)
notBlockedButTaken!

Figure 1 Model of Grid Controller in UPPAAL

The grid controller starts in the initial state called ***Initial****.* ***Initial*** has a state invariant "*initializationClock <= 0"* which when invalidated as the *initializationClock* increases, causes the grid controller to transition to its ***ReadyWait*** state, initializing the grid and setting the *gridClock* to 0 in the process. The *gridClock* is the clock variable that imposes timing specifications on the grid controller and its states.

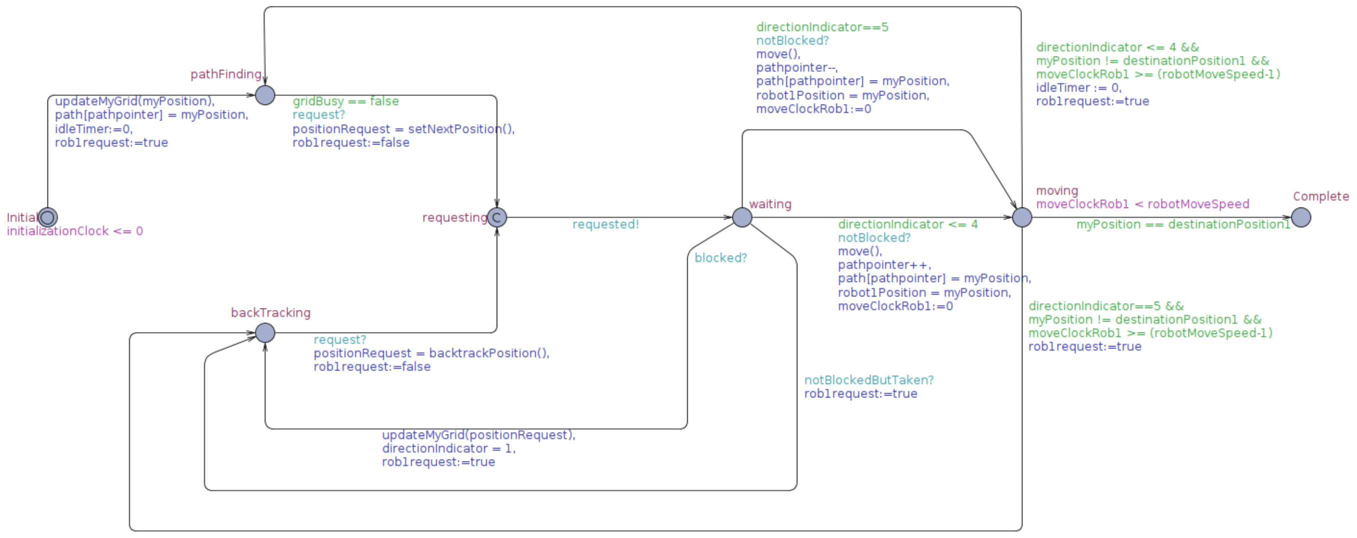
In the ***ReadyWait*** state, the invariant is "*gridClock <= 1."* The ***ReadyWait*** state has three edges, one that loops back to the ***ReadyWait*** state and resets the *gridClock* to 0*,* another that transitions the grid controller to its ***WaitingForRequest*** state, and another that transitions the grid controller to its complete or finished state. The looping edge keeps the grid controller busy-waiting to communicate with a robot, while the other edge synchronizes with the robots to enter the processing states of the system (which is *Requesting* for the robot and ***WaitingForRequest*** for the grid.)

In the ***WaitingForRequest*** state, the grid controller waits until it receives a *requested!* signal from a robot and when it does it transitions to the ***Processing*** state. During the transition, the grid controller checks the position requested by the robot against its 2D array, sets the boolean variable *ob* to the status of the position requested by the robot (*true or false for blocked or not blocked***)**, sets the boolean value *gridBusy* to true (to indicate the grid is currently processing a request) and resets the *gridClock* to 0.

The grid controller timing specifications for processing are defined as *4<= gridClock < 14*. This means the grid controller takes up to 14 ticks and at least 4 ticks to process a request and return the signal response to the robot. The grid controller returns *blocked*! if the position requested is blocked by an object, *notblocked*! if the position requested is available, and *notBlockedButTaken*! if the position requested is taken by another robot. Each of these edges transitions the grid controller to its ***Done*** state.

The ***Done*** state has one edge that transitions back to the ***ReadyWait*** state. When that transition is taken, the grid checks to see if a robot has reached its destination position (it will update the grid if so), sets *gridBusy* to false, and resets the grid clock back to 0.

Robot



pathFinding
updateMyGrid(myPosition),
path[pathpointer] = myPosition,
idleTimer:=0,
rob1request:=true
gridBusy == false
request?
positionRequest = setNextPosition(),
rob1request:=false
Initial
initializationClock <= 0
requesting
C
requested!
waiting
blocked?
directionIndicator==5
notBlocked?
move(),
pathpointer--,
path[pathpointer] = myPosition,
robot1Position = myPosition,
moveClockRob1:=0
directionIndicator <= 4 &&
myPosition != destinationPosition1 &&
moveClockRob1 >= (robotMoveSpeed-1)
idleTimer := 0,
rob1request:=true
moving
moveClockRob1 < robotMoveSpeed
Complete
myPosition == destinationPosition1
directionIndicator <= 4
notBlocked?
move(),
pathpointer++,
path[pathpointer] = myPosition,
robot1Position = myPosition,
moveClockRob1:=0
directionIndicator==5 &&
myPosition != destinationPosition1 &&
moveClockRob1 >= (robotMoveSpeed-1)
rob1request:=true
backTracking
request?
positionRequest = backtrackPosition(),
rob1request:=false
notBlockedButTaken?
rob1request:=true
updateMyGrid(positionRequest),
directionIndicator = 1,
rob1request:=true

Figure 2 Model of Robot

The robot "starts" in its initial state called ***Initial**.* This state has an invariant "*initializationClock <= 0*". This invariant is invalidated when the *initializationClock* starts incrementing and this causes the robot to transition to the next state called *pathfinding.* During this transition, the robot's local grid array is updated (its initial position is marked as visited), which contains a record of all the spaces the robot has visited, and the path history array *path* has the robot's initial position loaded into it. The *idleTimer* clock variable is reset to 0 and the boolean *rob1request* is set to true. If *rob1request* is set to true, the grid controller knows that it needs to attempt a new request as there is a robot waiting.

In the ***pathfinding*** state, there is one exit transition to the state ***requesting**.* This edge is guarded by the condition *"gridBusy == false",* which means that the edge cannot occur unless the grid controller is free to process, and it has a synchronization listening channel called *request*, that listens for the request synchronization from the grid controller. Once this transition happens, the robot sets the *positionRequest* variable to the next position the robot would like to traverse to (set by the *setNextPosition* function), and the *rob1request* variable to false. Once in the ***requesting*** state the robot raises the synchronization channel called *requested!* and transitions to the ***waiting*** state.

The ***waiting*** state has 4 edges that can be taken. One is the movement edge to the ***moving*** state with the guard *directionIndicator <= 4*. This edge occurs when the robot is in a progressive pathfinding mode and is attempting to visit a new position. The synchronization channel *notblocked?* raised by the grid controller indicates that the requested position was not blocked. When this edge is taken, the robot moves to the ***moving*** state, updates its local grid array(marks position as visited), stores the new position in its path history array, and the movement clock *moveCLockRob1* is reset to 0.

Another is the movement edge to the ***moving*** state with the guard *directionIndicator == 5*. This edge occurs when the robot is backtracking and reading positions from its path history array to backtrack to. The synchronization channel *notblocked?* raised by the grid indicates the backtracking position requested was not blocked. When this edge is taken the robot moves to the ***moving*** state (backtracks to its previous position), decrements its path history pointer and the movement clock *moveClockRob1* is reset to 0.

The other edge occurs when the grid controller raises the *notBlockedButTaken* signal which indicates that the requested position is not blocked but is currently taken by another robot. When this edge occurs the boolean *rob1Request* is set to true and the robot transition to its ***backtracking*** state.

The final possible edge from the ***waiting*** state is the one that occurs when the grid controller raises the *blocked* signal, indicating that the requested position is blocked. When this occurs the robot transitions to the ***backtracking*** state, the blocked position is marked in the local grid and the *directionIndicator* variable is reset to 1. The indicator is reset because the robot can no longer continue to traverse in the direction where the blocked signal occurred.

In the ***backtracking*** state, the robot follows logic that identifies the next direction it should attempt to traverse in. It will iterate through all of the possible directions, checking if there is a possible position the robot can visit that it has not visited yet. If there is, it sets the next position to that position and sends the request to the grid controller. If there is not valid position, the *directionIndicator* is set to 5 which indicates that the robot is backtracking through its path history, and sets the request to the controller to the previous position the robot was in.

The ***moving*** state has 3 possible edges that can be taken. One edge, is the transition to ***completed***, which occurs when the guard condition that the robot has arrived at its destination position is met. Another edge occurs when the *directionIndicator <= 4*, the robot is not at its destination position, and the movement clock *moveClockRob1* value is greater than or equal to the specified time variable *robotMoveSpeed* (which defines how long a robot can take to move). When the direction indicator is less than 4, this indicates that the robot is in pathfinding mode. That is why this edge leads to the ***pathfinding*** state. The other edge occurs when the *directionIndicator == 5*, the robot is not at its destination position and the movement clock is greater than or equal to the specified time variable *robotMoveSpeed*. This edge leads to the ***backtracking*** state.

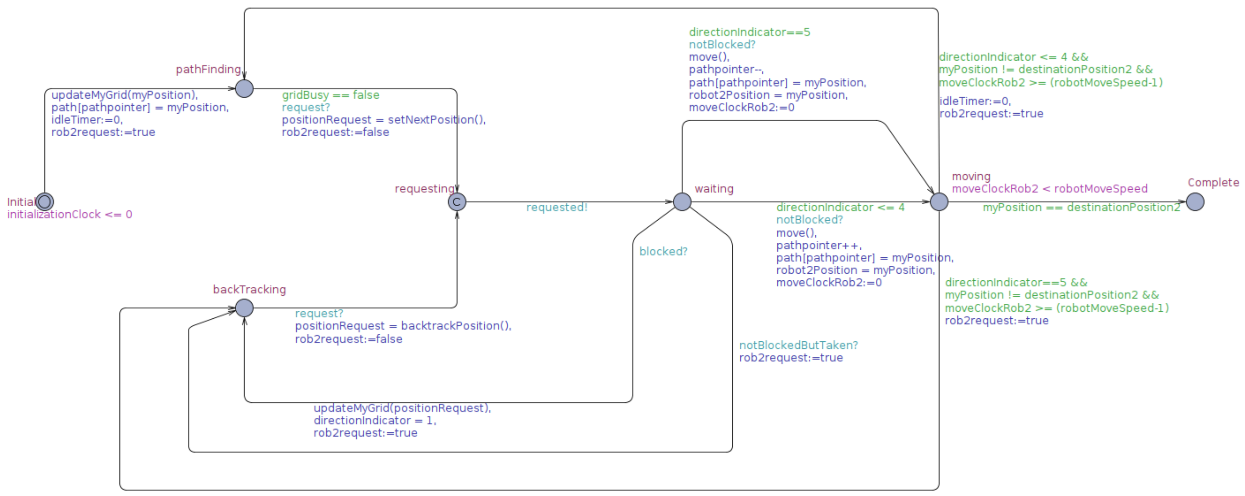

pathFinding
updateMyGrid(myPosition),
path[pathpointer] = myPosition,
idleTimer:=0,
rob2request:=true
Initial
initializationClock <= 0
gridBusy == false
request?
positionRequest = setNextPosition(),
rob2request:=false
requesting
C
requested!
waiting
directionIndicator==5
notBlocked?
move(),
pathpointer--,
path[pathpointer] = myPosition,
robot2Position = myPosition,
moveClockRob2:=0
directionIndicator <= 4 &&
myPosition != destinationPosition2 &&
moveClockRob2 >= (robotMoveSpeed-1)
idleTimer:=0,
rob2request:=true
moving
moveClockRob2 < robotMoveSpeed
Complete
myPosition == destinationPosition2
directionIndicator <= 4
notBlocked?
move(),
pathpointer++,
path[pathpointer] = myPosition,
robot2Position = myPosition,
moveClockRob2:=0
directionIndicator==5 &&
myPosition != destinationPosition2 &&
moveClockRob2 >= (robotMoveSpeed-1)
rob2request:=true
blocked?
backTracking
request?
positionRequest = backtrackPosition(),
rob2request:=false
notBlockedButTaken?
rob2request:=true
updateMyGrid(positionRequest),
directionIndicator = 1,
rob2request:=true

Figure 3 Model of Robot 2

The model of robot 2 is similar to that of robot one. However, the variables rob1request, robot1Position, moveclockRob1 and destinationPosition1 are rob2request, robot2Position, moveclockRob2 and destinationPosition2 respectively.