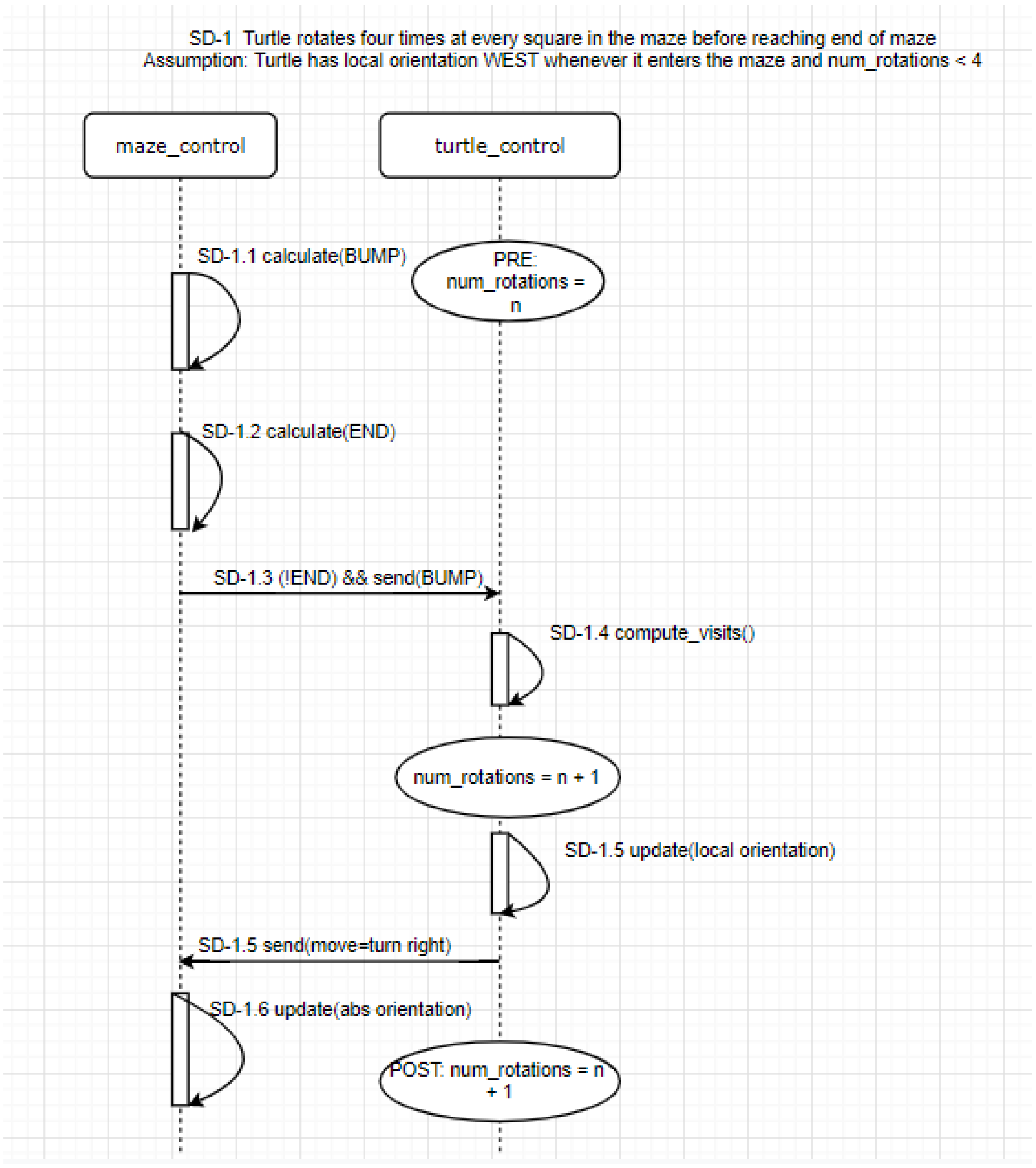
Lavina Chandwani

Project 10: Design Documentation

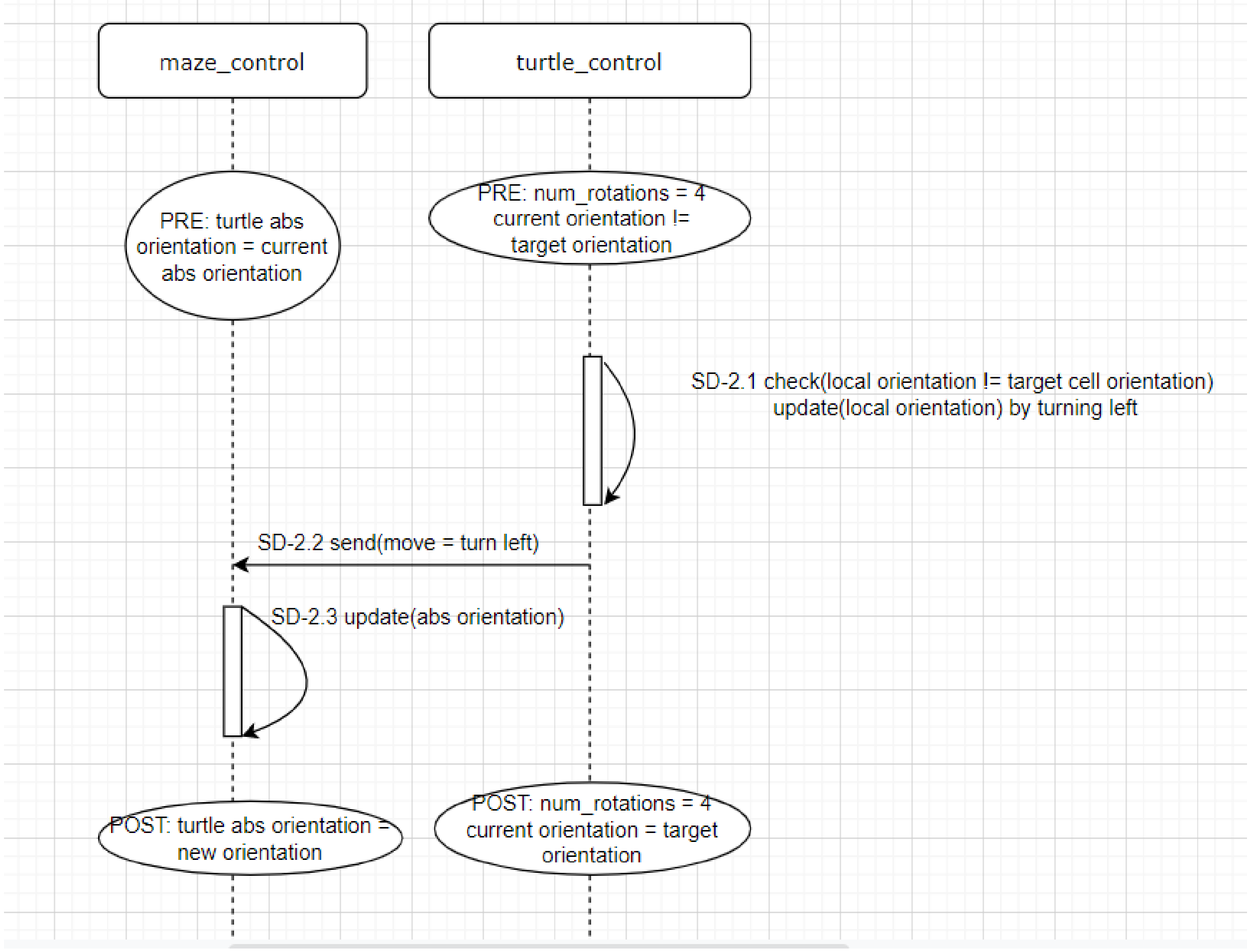
Requirements:

R-1: The turtle shall check if it has reached the end of maze each time it enters a new square and shall stop moving if reaches end of maze.
R-2: The turtle shall complete four rotations by turning right after entering a new square except the end of the maze.
R-3: The turtle shall rotate at most 90 degrees per tick.
R-4: The turtle shall check for a potential wall in the direction it is facing after every right turn.
R-5: The turtle shall keep record of the number of visits made to the adjacent square it faces after every right turn.
R-6: The turtle shall update least number of visits made to any adjacent square after collecting visits to each adjacent square.
R-7: The turtle shall only move in the direction it is facing.
R-8: The turtle shall move forward to the adjacent open square with least number of visits.
R-9: If the turtle is not facing the square with least number of visits, it shall rotate left until it faces that square.
R-10: The turtle shall not move through the walls of the maze.
R-11: The turtle shall not move more than one adjacent square per call to tick.
R-12: The turtle shall update the number of visits made to a square after moving to that square.

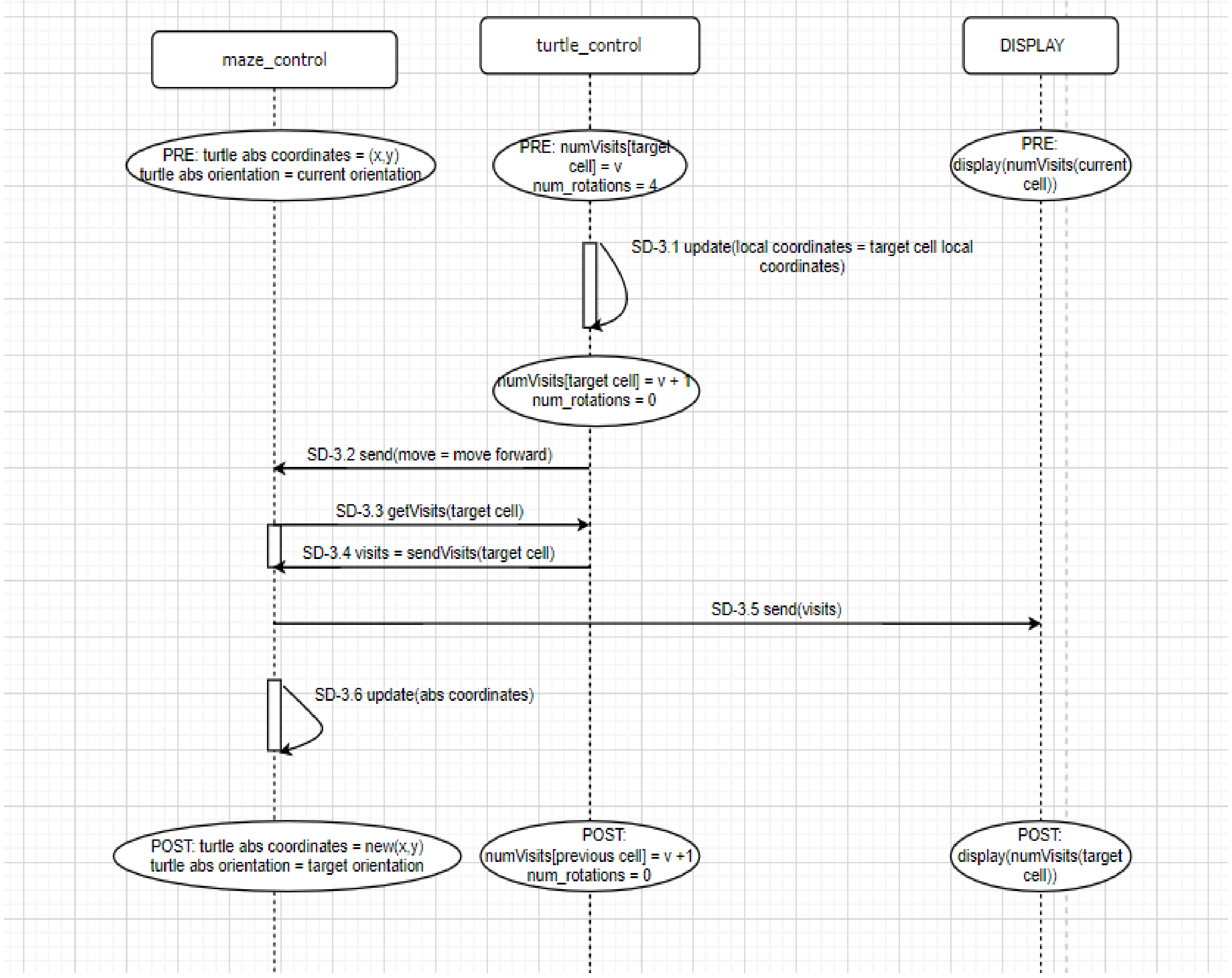
Sequence Diagrams:

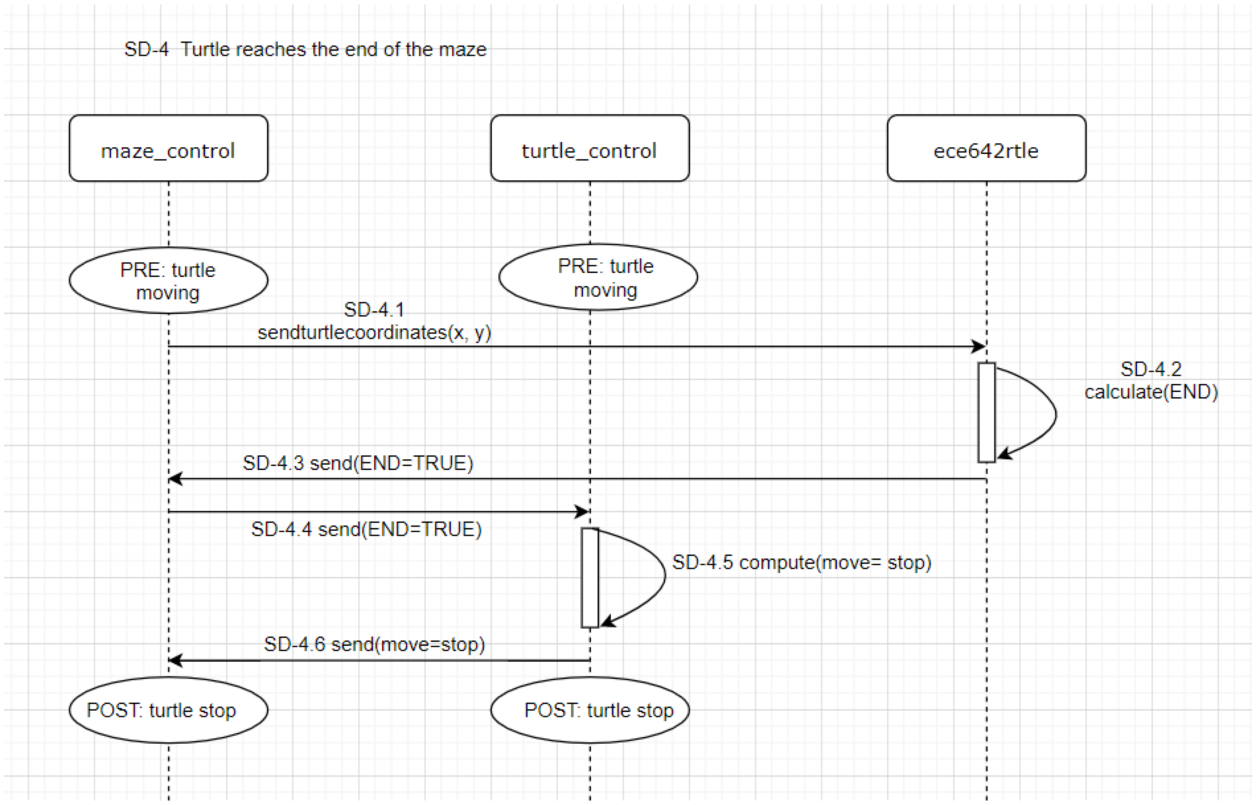

SD-1 Turtle rotates four times at every square in the maze before reaching end of maze
Assumption: Turtle has local orientation WEST whenever it enters the maze and num_rotations < 4
maze_control
turtle_control
SD-1.1 calculate(BUMP)
PRE: num_rotations = n
SD-1.2 calculate(END)
SD-1.3 (!END) && send(BUMP)
SD-1.4 compute_visits()
num_rotations = n + 1
SD-1.5 update(local orientation)
SD-1.5 send(move=turn right)
SD-1.6 update(abs orientation)
POST: num_rotations = n + 1

SD-2 Turtle rotates left until its local orientation matches target orientation to move forward in the maze
Assumption: target orientation is available to turtle from compute_target_orientation() function

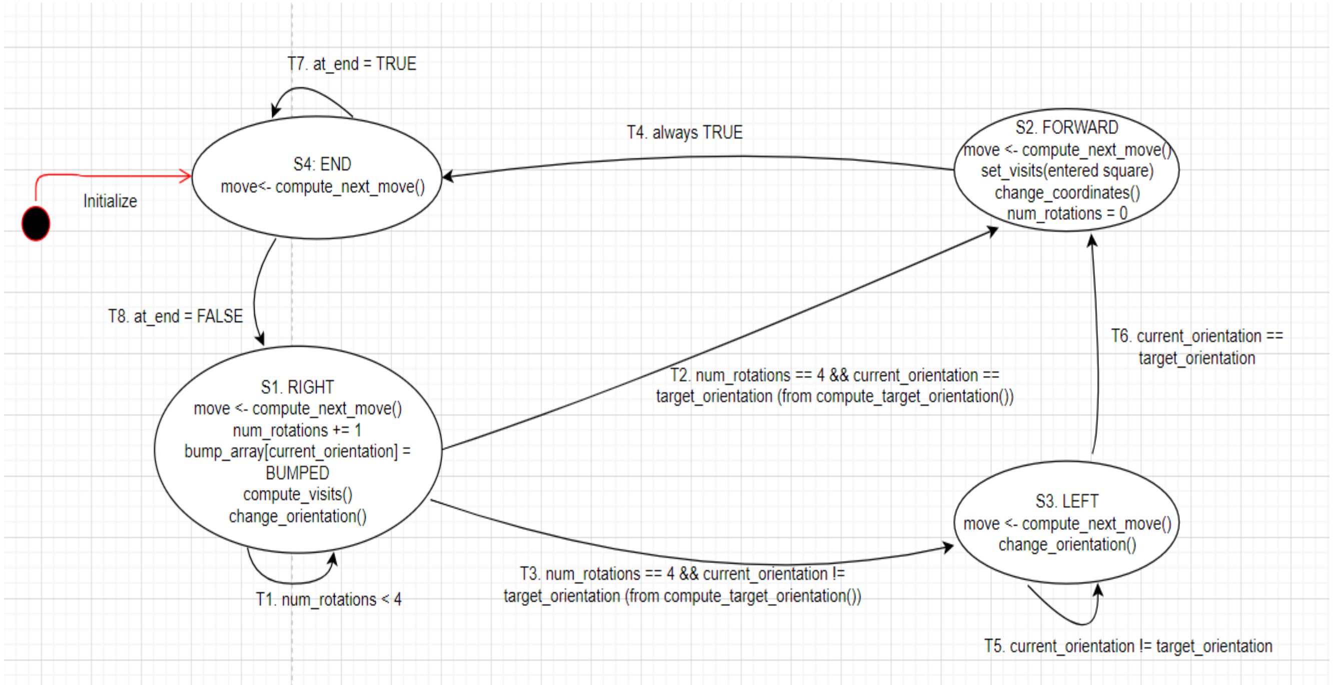

maze_control
turtle_control
PRE: turtle abs orientation = current abs orientation
PRE: num_rotations = 4 current orientation != target orientation
SD-2.1 check(local orientation != target cell orientation) update(local orientation) by turning left
SD-2.2 send(move = turn left)
SD-2.3 update(abs orientation)
POST: turtle abs orientation = new orientation
POST: num_rotations = 4 current orientation = target orientation

SD-3 Turtle's current orientation matches target orientation and moves forward
Assumption: turtle's local orientation matches target orientation and hence there are no changes in abs orientation since turtle next move is move forward.


maze_control
turtle_control
DISPLAY
PRE: turtle abs coordinates = (x,y)
turtle abs orientation = current orientation
PRE: numVisits[target cell] = v
num_rotations = 4
PRE: display(numVisits(current cell))
SD-3.1 update(local coordinates = target cell local coordinates)
numVisits[target cell] = v + 1
num_rotations = 0
SD-3.2 send(move = move forward)
SD-3.3 getVisits(target cell)
SD-3.4 visits = sendVisits(target cell)
SD-3.5 send(visits)
SD-3.6 update(abs coordinates)
POST: turtle abs coordinates = new(x,y)
turtle abs orientation = target orientation
POST: numVisits[previous cell] = v +1
num_rotations = 0
POST: display(numVisits(target cell))



Statechart:


T7. at_end = TRUE
T4. always TRUE
S4: END
move<- compute_next_move()
S2. FORWARD
move <- compute_next_move()
set_visits(entered square)
change_coordinates()
num_rotations = 0
Initialize
T8. at_end = FALSE
T6. current_orientation == target_orientation
T2. num_rotations == 4 && current_orientation == target_orientation (from compute_target_orientation())
S1. RIGHT
move <- compute_next_move()
num_rotations += 1
bump_array[current_orientation] = BUMPED
compute_visits()
change_orientation()
S3. LEFT
move <- compute_next_move()
change_orientation()
T1. num_rotations < 4
T3. num_rotations == 4 && current_orientation != target_orientation (from compute_target_orientation())
T5. current_orientation != target_orientation

Statechart: Turtle's movement logic
target_orientation is calculated by compute_target_orientation() function, while computing the target_orientation turtle doesn't turn right or left or move forward so it doesn't fit in any state in the statechart but in the code it is done in RIGHT when num_rotations become 4.

Statechart Variable Table:

Name	Type	Range	Description
bumped	Input	{TRUE, FALSE}	Tells the turtle if it has bumped the wall it faces
at_end	Input	{TRUE, FALSE}	Tells the turtle if it has solved the maze/reached the end of the maze
move	Output	{turn_left, turn_right, move_forward}	Next move of the turtle
bump_array	Internal variable	{(TRUE, FALSE), (TRUE, FALSE), (TRUE, FALSE), (TRUE, FALSE)}	Stores bump for every local orientation of turtle
num_rotations	Internal variable	{0,1,2,3,4}	Number of times turtle rotates in a cell to collect cell visits to adjacent cells
least_visit_count	Internal variable	{0,1,…..,MAX_VISITS} MAX_VISITS = 100 (in my algorithm)	Least number of visits made to an adjacent cell
next_visit_count	Internal variable	{(0,1,….,MAX_VISITS), (0,1,….,MAX_VISITS), (0,1,….,MAX_VISITS), (0,1,….,MAX_VISITS)}	Array that stores visits made to adjacent cells for each local orientation of turtle
current orientation	Internal variable	{LEFT, UP, RIGHT, DOWN}	Current local orientation of turtle
target orientation	Internal variable	{LEFT, UP, RIGHT, DOWN}	Target local orientation of turtle to make a move forward
visits[NUM_SQUARES][NUM_SQUARES] NUM_SQAURES = 23 (in my algorithm)	Internal variable	{(0,….,MAX_VISITS), …….(0,….,MAX_VISITS)} (23 times)	Number of visits of each square in the maze

Compute Function Designs:

1. compute_visits: This function stores the bump value and the number of visits made to each adjacent square it faces after turning right four times. This helps the turtle decide which square has least number of visits to move forward in the maze.

Input: least_visit_count, next_visit_count_array[], ort

least_visit_count is set to a high value (MAX_VISITS here) when turtle enters a new square, it is updated as the minimum element in next_visit_count_array[] after every right turn.
next_visit_count_array[] has four elements, each element denotes the number of visits made to an adjacent square for a particular local orientation of the turtle.
next_visit_count_array[0] is the visit count of the adjacent square and bump_array[0] is bumped for the adjacent square the turtle faces when turtle's local orientation is WEST.
next_visit_count_array[1] is the visit count of the adjacent square and bump_array[1] is bumped for the adjacent square the turtle faces when turtle's local orientation is NORTH.
next_visit_count_array[2] is the visit count of the adjacent square and bump_array[2] is bumped for the adjacent square the turtle faces when turtle's local orientation is EAST.
next_visit_count_array[3] is the visit count of the adjacent square and bump_array[3] is bumped for the adjacent square the turtle faces when turtle's local orientation is SOUTH.
next_visit_count_array's element is set to MAX_VISITS when bumped=TRUE for the adjacent square the turtle is facing. Compute function is called only when bumped=FALSE.
ort is current local orientation of the turtle.

Psuedocode:

```
procedure compute_visits(ort, least_visit_count, next_visit_count_array[])
    next_visit_count_array[ort] is visits[adjacent square local x coordinate][ adjacent square local x coordinate]
//visits[][] is a 23x23 file static array that keeps number of visits to each square in a 11x11 maze
    endif
    if least_visit_count is more than next_visit_count_array[ort]
        least_visit_count is next_visit_count_array[ort]
    endif
end procedure
```

2. compute_target_orientation: The turtle should be facing the square that it wants to move to next in the maze so the turtle should change its local orientation if it is not facing the square. This function computes the local orientation of the turtle for its next move forward. Since each element of next_visit_count_array denotes the visit count of the adjacent square in a particular local orientation of the turtle, matching it with least_visit_count will give target orientation of the turtle.
Inputs: next_visit_count_array[], least_visit_count
Output: target_ort is the local orientation of turtle to move forward

Psuedocode:

```
procedure compute_target_orientation(next_visit_count_array[], least_visit_count)
    if least_visit_count equals next_visit_count_array[0]
        then target_ort is WEST
    else if least_visit_count equals next_visit_count_array[1]
        then target_ort is NORTH
    else if least_visit_count equals next_visit_count_array[2]
        then target_ort is EAST
    else if least_visit_count equals next_visit_count_array[3]
        then target_ort is SOUTH
    else error
```

```
        endif
        return target_ort
end procedure
```

2. compute_next_move: The turtle calls this function in every state to get the next move of that state. It is decided based on what state the turtle is currently in. The state of turtle is accessible to this function since it is a file static variable, current_state.
Output: next_move of the turtle
Psuedocode:

```
procedure compute_next_move()
        if current_state equals RIGHT
                then next_move is TURN_LEFT
        else if current_state equals FORWARD
                then next_move is MOVE_FORWARD
        else if current_state equals LEFT
                then next_move is TURN_LEFT
        else if current_state equals END
                then next_move is STOP_MOVING
        else error
        endif
        return next_move
end procedure
```

Traceability:

a. Sequence Diagrams to Requirements

Traceability matrix	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
SD1		X	X	X	X	X						
SD2			X						X			
SD3							X	X		X	X	X
SD4	X											

b. Statechart to Requirements:

Traceability matrix	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
S1		X	X	X	X	X						
S2							X	X		X	X	X
S3			X						X			
S4	X											
T1		X										
T2		X					X	X				
T3		X					X	X	X			
T4	X											
T5							X	X	X			
T6							X	X				

T7	X											
T8	X											
compute_visits()					X	X		X				
compute_target_orientation()								X				
compute_next_move()	X	X						X	X			

Monitor Peer Review Checklist:

1. Code compiles clean with extensive warning checks (e.g. MISRA C rules)
2. Commenting: top of file, start of function, code that needs an explanation
3. Style is consistent and follows style guidelines
4. Proper modularity, module size, use of .h files and #includes
5. No orphans (redundant, dead, commented out, unused code & variables)
6. Conditional expressions evaluate to a boolean value; no assignments
7. Parentheses used to avoid operator precedence confusion
8. All switch statements have a default clause; preferably an error trap
9. Single point of exit from each function
10. Loop entry and exit conditions correct; minimum continue/break complexity
11. Conditionals should be minimally nested (generally only one or two deep)
12. SCC or SF complexity less than 10 to 15
13. Use const and inline instead of #define; minimize conditional compilation
14. Avoid use of magic numbers (constant values embedded in code)
15. Use strong typing (includes: sized types, structs for coupled data, const)
16. Variables have well chosen names and are initialized at definition
17. Minimum scope for all functions and variables; essentially no globals.
18. Concurrency issues (locking, volatile keyword, minimize blocking time)
19. Input parameter checking is done (style, completeness)
20. Error handling for function returns is appropriate
21. Null pointers, division by zero, null strings, boundary conditions handled
22. Floating point use is OK (equality, NaN, INF, roundoff); use of fixed point
23. Buffer overflow safety (bound checking, avoid unsafe string operations)

<u>All Reviewers:</u>

24. Code matches correct functionality
25. Code as simple, obvious, and easy to review as possible
26. Monitors cover all logical invariants
27. ROS_WARN() used to print out violations, and ROS_INFO() for relevant information
28. There are no violations for the monitors on the turtle code

Monitor Peer Review Sheet:

Peer Review Checklist 18-642		
(Substitute forms may be used as long as they are comparable)		
Group#	1	
Date:	22-Nov	
Artifact:	monitors/*	
Scribe:	Lavina Chandwani	
Leader:	Deepak Pallerla	
Time spent:	25 minutes	
Issue#	**Issue Description**	**Status**
1	Checklist Issue 2: All monitors: No comments added for each interrupt/helper function	Fixed
2	Checklist Issue 9: Line 32-39 Turn Monitor: Multiple return in the helper function getOrientation()	Fixed
3	Checklist Issue 14: Line 35-45 Tick Monitor: Magic number used	Fixed
4	Checklist Issue 24: Face Monitor: Incorrect Functionality, pose interrupt and bump interrupt	Fixed
5	Checklist Issue 28: Violations occur in attend and tick monitors	Not a Bug
Status Key:	Fixed	
	Not Fixed	
	Not a Bug (false alarm)	

Monitor Peer Review Defect Status:

Issue 5 of Peer Review is not a bug since it is a timing issue in both attend and tick monitors.
In atend monitor the atend function gets called before pose is updated so it gives a violation.
In tick monitor the count of bump/visit/pose randomly gets a violation in tick but that is also a timing issue.