Assignment 3

Amit Binu binua

March 11, 2017

Constants Module

Module

Constants

Uses

N/A

Syntax

Exported Constants

MAX_X = 180 *//dimension in the x-direction of the problem area*
MAX_Y = 160 *//dimension in the y-direction of the problem area*
TOLERANCE = 5 *//space allowance around obstacles*
VELOCITY_LINEAR = 15 *//speed of the robot when driving straight*
VELOCITY_ANGULAR = 30 *//speed of the robot when turing rad*

Exported Access Programs

none

Semantics

State Variables

none

State Invariant

none

Point ADT Module

Template Module

PointT

Uses

Constants

Syntax

Exported Types

PointT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
PointT	real, real	PointT	InvalidPointException
xcrd		real	
ycrd		real	
dist	PointT	real	

Semantics

State Variables

xc: real
yc: real

State Invariant

none

Assumptions

The constructor PointT is called for each abstract object before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

PointT(x, y):

- transition: $xc, yc := x, y$
- output: $out := self$
- exception $exc := ((\neg(0 \leq x \leq \text{Contants.MAX\_X}) \vee \neg(0 \leq y \leq \text{Constants.MAX\_Y})) \Rightarrow \text{InvalidPointException})$

xcrd():

- output: $out := xc$
- exception: none

ycrd():

- output: $out := yc$
- exception: none

dist(p):

- output: $out := \sqrt{(self.xc - p.xc)^2 + (self.yc - p.yc)^2}$
- exception: none

Region Module

Template Module

RegionT

Uses

PointT, Constants

Syntax

Exported Types

RegionT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
RegionT	PointT, real, real	RegionT	InvalidRegionException
pointInRegion	PointT	boolean	

Semantics

State Variables

lower_left: PointT *//coordinates of the lower left corner of the region*
width: real *//width of the rectangular region*
height: real *//height of the rectangular region*

State Invariant

None

Assumptions

The RegionT constructor is called for each abstract object before any other access routine is called for that object. The constructor can only be called once.

Access Routine Semantics

RegionT(p, w, h):

- transition: $lower\_left, width, height := p, w, h$
- output: $out := self$
- exception: $exc := (\neg(0 \leq p.xcrd() \leq Constants.MAX_X - w) \wedge (0 \leq p.ycrd() \leq Constants.MAX_Y - h)) \Rightarrow$ InvalidRegionException

pointInRegion(p):

- output: $out := \exists(i, j : PointT | j.ycrd() \in [lower\_left.ycrd()..height] \wedge j.xcrd() \in [lower\_left.xcrd()..width] : i.dist(j) < TOLERANCE)$
- exception: none

Generic List Module

Generic Template Module

GenericList(T)

Uses

N/A

Syntax

Exported Types

GenericList(T) = ?

Exported Constants

MAX_SIZE = 100

Exported Access Programs

Routine name	In	Out	Exceptions
GenericList		GenericList	
add	integer, T		FullSequenceException, InvalidPositionException
del	integer		InvalidPositionException
setval	integer, T		InvalidPositionException
getval	integer	T	InvalidPositionException
size		integer	

Semantics

State Variables

s: sequence of T

State Invariant

$|s| \leq$ MAX_SIZE

Assumptions

The GenericList() constructor is called for each abstract object before any other access routine is called for that object. The constructor can only be called once.

Access Routine Semantics

GenericList():

- transition: $self.s := <>$
- output: $out := self$
- exception: none

add(i, p):

- transition: $s := s[0..i-1] || < p > || s[i..|s|-1]$
- exception: $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{FullSequenceException} \mid i \notin [0..|s|] \Rightarrow \text{InvalidPositionException})$

del(i):

- transition: $s := s[0..i-1] || s[i+1..|s|-1]$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{InvalidPositionException})$

setval(i, p):

- transition: $s[i] := p$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{InvalidPositionException})$

getval(i):

- output: $out := s[i]$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{InvalidPositionException})$

size():

- output: $out := |s|$
- exception: none

Path Module

Template Module

PathT is GenericList(PointT)

Obstacles Module

Template Module

Obstacles is GenericList(RegionT)

Destinations Module

Template Module

Destinations is GenericList(RegionT)

SafeZone Module

Template Module

SafeZone extends GenericList(RegionT)

Exported Constants

MAX_SIZE = 1

Map Module

Module

Map

Uses

Obstacles, Destinations, SafeZone

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
init	Obstacles, Destinations, SafeZone		
get_obstacles		Obstacles	
get_destinations		Destinations	
get_safeZone		SafeZone	

Semantics

State Variables

obstacles : Obstacles
destinations : Destinations
safeZone : SafeZone

State Invariant

none

Assumptions

The access routine init() is called for the abstract object before any other access routine is called. If the map is changed, init() can be called again to change the map.

Access Routine Semantics

init(o, d, sz):

- transition: $obstacles, destinations, safeZone := o, d, sz$

- exception: none

get_obstacles():

- output: *out* := *obstacles*
- exception: none

get_destinations():

- output: *out* := *destinations*
- exception: none

get_safeZone():

- output: *out* := *safeZone*
- exception: none

Path Calculation Module

Module

PathCalculation

Uses

Constants, PointT, RegionT, PathT, Obstacles, Destinations, SafeZone, Map

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
is_validSegment	PointT, PointT	boolean	
is_validPath	PathT	boolean	
is_shortestPath	PathT	boolean	
totalDistance	PathT	real	
totalTurns	PathT	integer	
estimatedTime	PathT	real	

Semantics

Access Routine Semantics

is_validSegment(p_1, p_2):

- output: $out := \forall(q, r : PointT | (q.dist(p_1) + q.dist(p_2) = p_1.dist(p_2)) \wedge (r.dist(q) \leq$ Constants.TOLERANCE) : $\forall(i : \mathbb{N} | i \in [0..Map.obstacles\_.size() - 1] :$ $\neg((Map.obstacles\_.getval(i)).pointInRegion(r))))$
- exception: None

is_validpath(p):

- output: $out := \exists(q : PointT \wedge n : \mathbb{N} | n \in [0..p.size() - 1] \wedge (q.dist(p.getval(n)) +$ $q.dist(p.getval(n+1)) = p.getval(n).dist(Map.Path\_.getval(n+1))))$ $\forall(i, j, k : \mathbb{N} \wedge safe : Map.safeZone \wedge dest : Map.destinations | i \in [0..safe.size() -$ $1] \wedge safe.getval(i).pointInRegion(p.getval(0)) \wedge safe.getval(i).pointInRegion(p.getval(p.size() -$ $1)) \wedge j \in [0..dest.size() - 1] \wedge k \in [1..p.size() - 1] \wedge dest.getval(j).pointInRegion(p.getval(0)) \wedge$ $dest.getval(j).pointInRegion(p.getval(k)) : p.isvalidSegment(p.getval(k-1), p.getval(k)))$

- exception: none

is_shortestPath(p):

- output: $out := \forall(x : PathT | is\_validPath(x) : totalDistance(p) \geq totalDistance(x))$
- exception: none

totalDistance(p):

- output: $out := +(i : \mathbb{N} | i \in [0..p.size()-1] \wedge i < p.size()-2 : p.getval(i).dist(p.getval(i+1)))$
- exception: none

totalTurns(p):

- output: $out := +(j : \mathbb{N} | j \in [0..p.size()-3] \wedge (p.getval(j+1).dist(p.getval(j)) + p.getval(j+2).dist(p.getval(j+1))) \neq p.getval(j+2).dist(p.getval(j)) : 1)$
- exception: none

estimatedTime(p):

- output: $out := (Constants.VECLOCITY\_LINEAR \times totalDistance(p)) + (+(i : \mathbb{N} | i \in [0..p.size()-3] : Constants.VELOCITY\_ANGULAR \times angle(p.getVal(i), p.getVal(i+1), p.getVal(i+2))))$
- exception: none

LOCAL FUNCTIONS:

angle($p1, p2, p3$):

- output: $out := \arccos(((p3.ycrd()-p2.ycrd()) \times (p2_ycrd()-p1.ycrd()) + ((p3.xcrd()-p2.xcrd()) \times (p2.xcrd()-p1.xcrd())) / (p3.dist(p2) \times p2.dist(p1)))$
- exception: none

Critique for this assignment's interface

For the most part, this assignment's interface is very well described and unambiguous. Therefore, these specifications are reliable.
Howver, I think it would be better if some local functions were introduced for some methods in some modules, in order to make it easy to read.
Also, I think some methods like *is_validSegement* and *is_isvalidPath* should have exceptions that check whether the given input is of appropriate object. This will make the specification more reliable and verfiable.
These exceptions can be implemented by adding a getter method in the modules that don't have one. The getter method will return the parameters and these returned values can be used to check whether it is of the appropriate method or not.