The Maze Tracing Robot
A Sample Specification

Chapter 1

The Requirements

1.1 Requirements Specification

By convention, identifiers are italicized, type names end in T and constants are in capital letters. The names of variables are prefixed with either i_, for inputs, or o_, for outputs or s_, for internal state representations.

The software is intended to find the shortest path through a 2-dimensional maze and control the 'draw-bot' (a robot that is capable of moving a pen to mark on paper), such that it traces that path on a picture of the maze.

1.1.1 Pen Position

We represent the location of the draw-bot pen tip using a Boolean, namely o_penDown, to indicate if the pen is touching the maze surface or not, and a pair, $\langle$o_penPos.x, o_penPos.y$\rangle$ of reals, representing the location in the horizontal plane where the pen tip is touching the maze (if o_penDown is `true`) or would touch the maze if lowered (if o_penDown is `false`). The location is specified by the distance, in millimeters, from the respective axis, which are parallel (x = 0) and perpendicular (y = 0) to the front edge of the robot arm base. The extent of the region of interest is defined by the constants MIN_X, MAX_X, MIN_Y and MAX_Y. The origin is the center of the robot base post. The 'home' location of the pen-tip (to which it is returned on initialization of the draw-bot), is $\langle HOME\_X, HOME\_Y\rangle$.

1.1.2 Maze

As illustrated in Figure 1.1, the maze is contained within a

$$M\_WIDTH\,\text{mm} \times M\_HEIGHT\,\text{mm}$$

region of the horizontal plane bounded by the lines $x = -M\_X\_OFFSET$, $y = M\_Y\_OFFSET$, $x = -M\_X\_OFFSET + M\_WIDTH$ and $y = M\_Y\_OFFSET +$

Figure 1.1: Robot and Maze Parameters

M_HEIGHT, which are the external walls of the maze. The 'internal walls' of the maze are segments of the lines $x = -M\_X\_OFFSET + n \times M\_CELL\_SIZE$ mm and $y = M\_Y\_OFFSET + n \times M\_CELL\_SIZE$ mm, where n is an integer (i.e., a square grid with line spacing *M_CELL_SIZE* mm). The endpoints of the walls lie at intersections of these grid lines. Figure 1.2 is a sample maze with dashed lines indicating the possible wall locations.

1.1.3 Computer System

The draw-bot is controlled using a 80386 based PC running MS-DOS 6.0. The computer is equipped with Borland C compiler (version 3.1) and libraries for controlling the robot (`robots.lib`, `robotm.lib` and `robotl.lib`). The maze-tracer software will be expected to compile and run in this environment.

1.1.4 Draw-Bot

The draw-bot is constructed using a Robix^{TM} RCS-6 construction set. It consists of three arms, each of which is controlled by a motor. The first two arms move in the horizontal plane to position the pen and the third arm is used to raise or lower the pen.

1.2 Environment Variables

This section gives the quantities in the environment to be monitored and/or controlled by the system. Note that all environment variables are functions of time.

1.2.1 Inputs

i_mazeWalls : set of **positionT**
: The set of points that make up the walls of the maze. Note that the exterior walls (i.e., the perimeter) are included.

i_mazeStart : **positionT**
: Start position for the maze.

i_mazeEnd : **positionT**
: Finish position for the maze.

i_stopButton : **buttonT**
: The status of the button labeled "stop".

i_homeButton : **buttonT**
: The status of the button labeled "home".

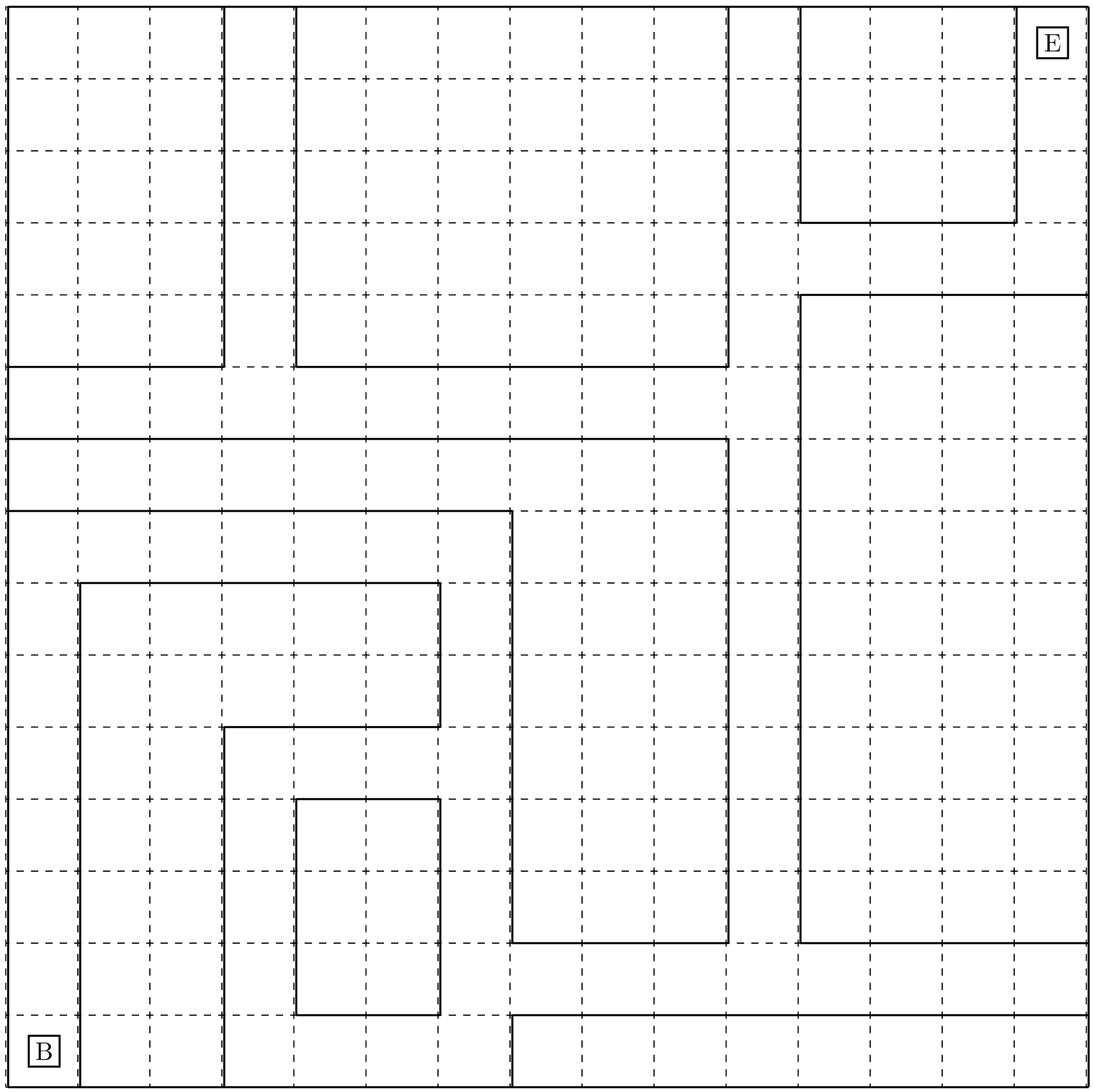

Figure 1.2: Sample Maze

i_backButton : **buttonT**
The status of the button labeled “back”.

i_mazeFile : **string**
The file name passed on the command line.

1.2.2 Outputs

o_penPos : **positionT**
The position of the pen relative to the ‘origin’ $\langle 0, 0\rangle$, which is the center of the robot base post.

o_penDown : **Boolean**
`true` iff the pen is touching the plane containing the maze. Assumed to be initially `false`.

o_powerOn : **Boolean**
`true` iff the robot power is on. Assumed to be initially `false`.

o_message : **string**
The message displayed on the operator console.

1.3 Behavioral Requirements

This section describes the required behavior of the Maze-Tracing Robot in terms of the environmental quantities described in Section 1.2. To aid in understanding, and to help expose students to a variety of formats, the requirements are presented in two forms: Informal and State Machine. These descriptions are intended to describe the same behavior and are in some sense complimentary, since each method has its own strengths and weaknesses.

1.3.1 Informal Description

Safety Requirements

If at any time the stop button is pressed (i_stopButton = *Down*) the robot must stop moving within *RESPONSE_TIME* seconds and must remain stationary until the stop button is released (i_stopButton = *Up*).

When the pen is down (o_penDown = `true`) the pen tip must never come within *WALL_SPACE* mm of a wall point (wall(o_penPos) = `true`).

Messages

Whenever a significant event occurs (i.e., a button is pressed or released, the pen reaches a significant point in its journey, or an error is detected) the software must output a diagnostic message describing the event and the system’s response to it.

Performance

The goal of the program should be to minimize the time between the pen first touching the paper and it being returned to its home position.

Initialization

When the program is started i_mazeFile is read. If an error occurs (e.g. file read failure) or if there is no path through the maze, then an appropriate diagnostic message must be output and the control program must exit without turning on the robot power (o_powerOn = `false`).

Starting

After i_mazeFile has been read, and it has been determined that there is a path through the maze, the robot power must be turned on (which meanso_powerOn = `true`), which initializes the pen to the home position

$$\mathsf{o\_penPos} = \langle \mathit{HOME\_X}, \mathit{HOME\_Y} \rangle$$

with the pen up (o_penDown = `false`). The pen must then be moved to the start position of the maze

$$\mathsf{o\_penPos} \in \mathrm{tol}(\mathsf{i\_mazeStart}).$$

Forward

Once the starting position has been reached (o_penPos $\in$ tol(i_mazeStart)) the pen must be lowered (o_penDown = `true`) and a path traced through the maze to the end (o_penPos $\in$ tol(i_mazeEnd)). When the pen reaches the end of the maze (o_penPos $\in$ tol(i_mazeEnd)) it must be raised (o_penDown = `false`) and returned to the home position.

Reverse

If at any time while the path is being traced the "back" button is pressed (i_backButton = *Down*) the Draw-bot is required to reverse the direction of its tracing within *RESPONSE_TIME* seconds and begin to re-trace its path back to the beginning (o_penPos $\in$ tol(i_mazeStart)). It should continue to re-trace its path only as long as the "back" button is held down—when it is released the Draw-bot should continue in the forward direction. If, while reversing, it reaches the start position it should stop there until either the "back" button is released or the "home" button is pressed.

Home

If at any time while the path is being traced (in either direction) the "home" button is pressed (i_homeButton = *Down*) the Draw-bot is required to stop tracing within *RESPONSE_TIME* seconds, raise the pen (o_penDown = `false`) and return to the home position, without making any further marks.

Done

When the pen has been returned to the home position, the power must be turned off (o_powerOn = `false`) and the system must exit.

1.3.2 State Machine

This section gives an alternate presentation of the the Maze-tracer requirements by defining a Finite State Machine using the notation found in Section 3.7 of [**?**].

State variables

s_mode : { *Init*, *Starting*, *Forward*, *Holding*, *Reverse*, *Home*, *Done* }
    The system mode.

s_holdRet : { *Starting*, *Forward*, *Reverse*, *Home* }
    The system mode to return to when the "stop" button is released.

s_holdPos : **positionT**
    The position in which the pen is to be held.

s_holdDown : **Boolean**
    The value of o_penDown when the stop button was pressed.

Initial State

s_mode := *Init*

Transitions and Outputs

Table 1.1 describes the state transition function, Table 1.2 describes the o_message event-output relation and Table 1.3 describes the o_penPos, o_penDown and o_powerOn condition-output relation. The performance goal, which does not appear in this description, is to minimize the time between the transition to s_mode = Forward and s_mode = Done.

The predicate reverse is true if the pen back-traces the path to the start which it came. We do not give a formal definition for the predicate reverse.

1.4 Definitions

This section defines types, functions and constants used in the requirements specification.

Table 1.1: Transition Function

Condition		New state
s_mode	Input	
Init	Error opening or reading i_mazeFile	s_mode := *Done*
	$\neg$connected(i_mazeStart, i_mazeEnd)	s_mode := *Done*
	connected(i_mazeStart, i_mazeEnd)	s_mode := *Starting*
Starting	o_penPos $\in$ tol(i_mazeStart)	s_mode := *Forward*
	i_stopButton = *Down*	s_mode := *Holding* s_holdRet := *Starting* s_holdPos := o_penPos s_holdDown := o_penDown
Forward	o_penPos $\in$ tol(i_mazeEnd)	s_mode := *Home*
	i_stopButton = *Down*	s_mode := *Holding* s_holdRet := *Forward* s_holdPos := o_penPos s_holdDown := o_penDown
	i_homeButton = *Down* $\wedge$ i_stopButton = *Up*	s_mode := *Home*
	i_backButton = *Down* $\wedge$ i_stopButton = *Up* $\wedge$ i_homeButton = *Up*	s_mode := *Reverse*
Holding	i_stopButton = *Up*	s_mode := s_holdRet
Reverse	o_penPos $\in$ tol(i_mazeStart)	s_mode := *Holding* s_holdRet := *Forward* s_holdPos := o_penPos s_holdDown := o_penDown
	i_stopButton = *Down*	s_mode := *Holding* s_holdRet := *Reverse* s_holdPos := o_penPos s_holdDown := o_penDown
	i_homeButton = *Down* $\wedge$ i_stopButton = *Up*	s_mode := *Home*
	i_backButton = *Up* $\wedge$ i_stopButton = *Up* $\wedge$ i_homeButton = *Up*	s_mode := *Forward*
Home	o_penPos = $\langle HOME\_X, HOME\_Y \rangle$	s_mode := *Done*
	i_stopButton = *Down*	s_mode := *Holding* s_holdRet := *Home* s_holdPos := o_penPos s_holdDown := o_penDown
Done	`true`	system exit

Table 1.2: o_message Event-Output Function

Condition		o_message
s_mode	Input	
Init	Error opening or reading i_mazeFile	appropriate diagnostics
	$\neg$connected(i_mazeStart, i_mazeEnd)	“No path found, nothing to do.”
	connected(i_mazeStart, i_mazeEnd)	“Path found, starting tracing.”
Holding	i_stopButton $= Up$	“Stop button released, resuming.”
Forward	o_penPos $\in$ tol(i_mazeEnd)	“End of maze reached, returning to home position.”
	i_homeButton $= Down$ $\wedge$ i_stopButton $= Up$	“Home button pressed, returning to home position.”
	i_backButton $= Down$ $\wedge$ i_stopButton $= Up$ $\wedge$ i_homeButton $= Up$	“Back button pressed, reversing direction.”
Reverse	o_penPos $\in$ tol(i_mazeStart)	
	i_homeButton $= Down$ $\wedge$ i_stopButton $= Up$	“Home button pressed, returning to home position.”
	i_backButton $= Up$ $\wedge$ i_stopButton $= Up$ $\wedge$ i_homeButton $= Up$	“Back button released, resuming forward tracing.”
Home	o_penPos $= \langle HOME\_X, HOME\_Y \rangle$	“Home position reached, terminating.”
-	i_stopButton $= Down$	“Stop button pressed, holding.”

Table 1.3: o_penPos, o_penDown and o_powerOn Condition-Output Function

s_mode	o_penPos |	o_penDown =	o_powerOn =
Init	o_penPos $= \langle HOME\_X, HOME\_Y \rangle$	`false`	`false`
Starting	`true`	`false`	`true`
Forward	$\neg$wall(o_penPos)	`true`	`true`
Reverse	$\neg$wall(o_penPos) $\wedge$ reverse(o_penPos)	`true`	`true`
Holding	o_penPos = s_holdPos	s_holdDown	`true`
Home	`true`	`false`	`true`
Done	o_penPos $= \langle HOME\_X, HOME\_Y \rangle$	`false`	`false`

1.4.1 Types

pathT = sequence of tuples of $\langle s, f : \mathbf{positionT} \rangle$

positionT = tuple of $\langle x : [MIN\_X \ldots MAX\_X], y : [MIN\_Y \ldots MAX\_Y] \rangle$

buttonT = $\{ \ Up, \ Down \ \}$

1.4.2 Functions

connected : **positionT** × **positionT** → **Boolean**
connected(b, e)

$$\doteq \left\{ \exists p_i \ \in \mathbf{postitionT} \ \ p_0 = b \wedge p_n = e \wedge \forall t \quad 0 \le t \le 1 \ \ \neg\mathrm{wall}(tp_i + (1-t)p_{i+1})^1 \right\}$$

tol : **positionT** → set of **positionT**
tol(p)

$$\doteq \left\{ q \in \mathbf{postitionT} \middle| \left(\sqrt{(q.x - p.x)^2 + (q.y - p.y)^2} \le POS\_TOL \text{ mm} \right) \right\}$$

wall : **positionT** → **Boolean**
wall(p)

$$\doteq (\exists q_1, q_2 \in \text{i\_mazeWalls}) \, (||t \ q_1 - (1-t)q_2|| \le WALL\_SPACE \text{ mm})$$

1.4.3 Constants

Table 1.4 lists the constants used in this specification, their informal interpretation and their range of values. Your software should be able to be easily changed to accommodate changes in these values within the specified ranges. The actual values of these constants will be provided late in the term.

[1] $tp_i + (1-t)p_{i+1} \quad 0 \le t \le 1$ is the line connecting p_i and p_{i+1}

Table 1.4: Constants

Name	Possible Values	Interpretation
MAX_X	[0...500]	Maximum valid x co-ordinate, millimeters.
MIN_X	[-500...0]	Minimum valid x co-ordinate, millimeters.
MAX_Y	[0...500]	Maximum valid y co-ordinate, millimeters.
MIN_Y	[-500...0]	Minimum valid y co-ordinate, millimeters.
HOME_X	$[MIN\_X \ldots MAX\_X]$	x location of pen 'home' position, millimeters.
HOME_Y	$[MIN\_Y \ldots MAX\_Y]$	y location of pen 'home' position, millimeters.
M_X_OFFSET	$[1 \ldots MAX\_X - M\_WIDTH]$	x distance of maze from origin, millimeters.
M_Y_OFFSET	$[1 \ldots MAX\_Y - M\_HEIGHT]$	y distance of maze from origin, millimeters.
M_WIDTH	$[M\_CELL\_SIZE \ldots MAX\_X]$	Width of maze, millimeters.
M_HEIGHT	$[M\_CELL\_SIZE \ldots MAX\_Y]$	Height of maze, millimeters.
M_CELL_SIZE	[4 ... 25]	Width/Height of a maze cell, millimeters.
RESPONSE_TIME	[2 ... 15]	Maximum delay before responding to a button, seconds.
MAX_TIME	[60 ... 300]	Maximum time allowed to trace the maze, seconds.
WALL_SPACE	$[1 \ldots \frac{M\_CELL\_SIZE}{2}]$	Minimum distance between the pen and walls, millimeters.
POS_TOL	$[1 \ldots \frac{M\_CELL\_SIZE}{2}]$	Maximum tolerance on locating the start and end positions, millimeters.

1.5 Software Interface

This section describes how the Maze-tracer control software interfaces with the operator and the robot by giving the relationship between the variables described in Section 1.2 and quantities available to the software.

1.5.1 Inputs

Maze

The values of the i_mazeStart, i_mazeEnd and i_mazeWalls are read from the text file whose name is given by i_mazeFile. Since the maze is constructed from lines in a grid as described in Section 1.1.2, points are represented by the index of the grid lines (integers). The first two lines of the file contain pairs that give the location of i_mazeStart and i_mazeEnd, respectively, which are taken to be the middle of the 'cell' with the given point as its lower left corner, i.e., if the first line of the file contains "`1 3`" then i_mazeStart is located at

$$\Big\langle M\_X\_OFFSET + M\_CELL\_SIZE + \frac{M\_CELL\_SIZE}{2}$$

$$M\_Y\_OFFSET + 3M\_CELL\_SIZE + \frac{M\_CELL\_SIZE}{2}\Big\rangle.$$

The remaining lines each contain four integers representing the endpoints of a wall. For example a line containing "`0 8 7 8`" indicates that all the points from

$$\langle M\_X\_OFFSET\ ,\ M\_Y\_OFFSET + 8M\_CELL\_SIZE\rangle$$

to

$$\langle M\_X\_OFFSET + 7M\_CELL\_SIZE$$

, $M\_X\_OFFSET + 8M\_CELL\_SIZE\rangle$, inclusive, are in i_mazeWalls. The boundaries of the maze region are also considered to be 'walls'. The following is a sample input file describing the maze appearing in Figure 1.2, with start point in the lower left corner, and end point in the upper right.

```
0 0
14 14
1 0 1 7
3 0 3 5
4 1 4 4
4 1 6 1
6 1 6 4
4 4 6 4
7 0 7 1
7 1 15 1
7 2 10 2
11 2 15 2
7 2 7 8
10 2 10 9
```

```
11 2 11 11
3 5 6 5
6 5 6 7
6 7 1 7
0 8 7 8
0 9 10 9
0 10 3 10
4 10 10 10
3 10 3 15
4 10 4 15
10 10 10 15
11 11 15 11
11 12 14 12
11 12 11 15
14 12 14 15
```

Note that there are several possible files to represent the same maze. Not only can the walls be listed in any order, but it is possible to describe a segment as one continuous segment or several shorter ones. Also note that in any line of the file the endpoints can appear in either order.

Buttons

The values of the buttons are read using the following access programs of the appropriate `robot.lib` library.

```
short i_homeButton();      /* Return 1 if Home button pressed 0 else */
short i_stopButton();      /* Return 1 if Stop button pressed 0 else */
short i_reverseButton();   /* Return 1 if Reverse button pressed 0 else */
```

1.5.2 Outputs

Pen Position

The pen position is controlled by manipulating the Draw-bot arms using the routines in the appropriate `robot` library to set pen position. The following access program controls the pen position.

```
short o_penPos(int x,int y);   /*  Move Pen to position x, y
  Returns 0 if OK, <>0 if ERROR */

short o_penDown(int pen);   /* Move Pen down pen=1, move Pen up pen=0
       Returns 0 if OK, <>0 if ERROR  */
```

Power

The motor power is turned on or off using the following access program.

```
short o_power(int pow);     /* Turn Power on pow=1, turn Power off pow=0
       Returns 0 if OK, <>0 if ERROR */
```

Before the Draw-bot can be used it must be initialized using the following access program.

```
short o_init(void);       /* Call at the Beginning to initialize
   returns 0 if status OK, 1 if error */
```

Message

Status and diagnostic messages are output using the `o_message` function of the library!

1.6 Expected Changes

The software should be designed to make it relatively easy to accommodate any of the following classes of changes.

- Changes to the geometry of the robot such that the mapping from a position (i.e., $\langle x, y \rangle$ pair) to the robot inputs is different.
- Changes to the interface to the robot.
- Changes to the format of the maze input file.
- Changes to any constant value within the given ranges.

Chapter 2

The Design

2.1 Module Guide : Maze Tracing Robot

In the following we propose a modularization for our robot project. The modularization is illustrated in 2.1

Module Name:	maze_storage
Prefix:	- ms_
Service:	- stores the maze
Secret:	- how the maze is stored
Module Name:	path_storage
Prefix:	- ps_
Service:	- stores the shortest path
Secret:	- how the path is stored
Module Name:	load_maze
Prefix:	- lm_
Service:	- loads the maze
Secret:	- where and how the maze file is read in
Module Name:	find_path
Prefix:	- fp_
Service:	- finds the shortest path through the maze
Secret:	- the algorithm for finding the shortest path
Module Name:	control
Prefix:	- cn_
Service:	- controls the movement of the arm
Secret:	- how the arm moves from position to position and how the buttons are checked
Module Name:	geometry
Prefix:	- gm_
Service:	- handles geometric positioning of the arm
Secret:	- how the calculations from cell coords to robot coords are performed
Module Name:	hardware

Prefix: - hw_
Service: - handles hardware aspects of arm (movement and button checking)
Secret: - how it interfaces with the robot
Module Name: types_constants
Service: - provides standard variable types and constants to modules
Secret: - how the data structures are defined and constants defined and calculated

2.2 maze_storage : MIS

Imported Data Types: `cell` `Boolean`
Imported Constants: `NUM_X_CELLS` `NUM_Y_CELLS`
Exported Functions

NAME	INPUT	OUTPUT	EXCEPTION
ms_init			
ms_set_maze_start	cell		ms_not_initialized ms_cell_out_of_range
ms_set_maze_end	cell		ms_not_initialized ms_cell_out_of_range
ms_get_maze_start		cell	ms_not_initialzed ms_no_start
ms_get_maze_end		cell	ms_not_initialized ms_no_end
ms_set_wall	cell,cell		ms_not_initialized ms_not_valid_wall
ms_is_connected	cell,cell	`Boolean`	ms_not_initialized ms_cell_out_of_range ms_not_neighbours

State Variables

`maze` : `set of tuple` < `cell`, `cell` >
`start` : `cell`
`end` : `cell`
`is_init` : `Boolean`

Access Function Semantics

`ms_init()`

Transition: `maze` :=<>
`start` :=<>
`end` :=<>
`is_init` := `true`

`ms_set_maze_start(c:cell)`

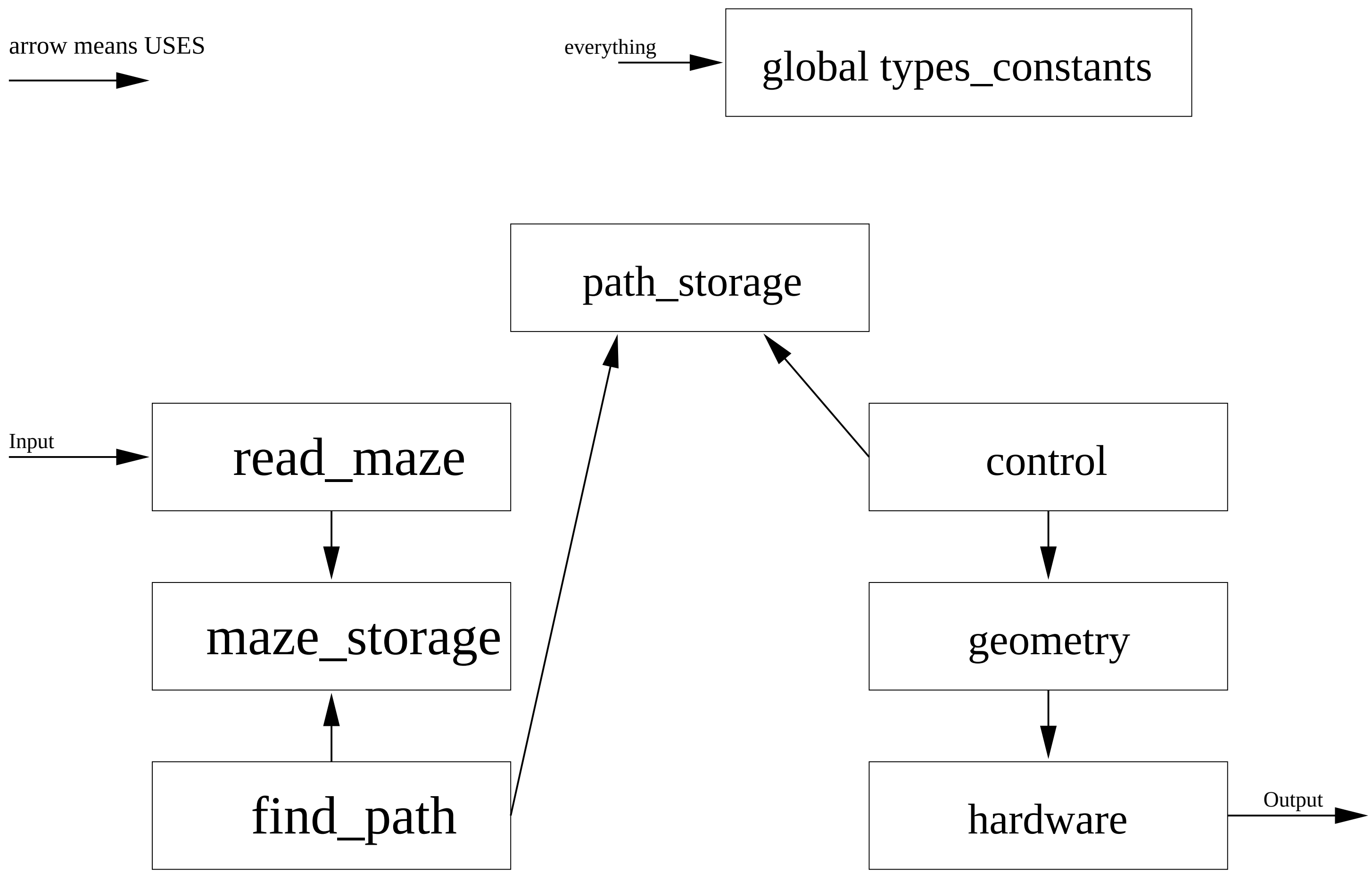

Figure 2.1: Robot Modules

Exception: $\neg$is_init $\Rightarrow$ ms_not_initialized
$\neg$(cell_in_range(c)) $\Rightarrow$ ms_cell_out_of_range
Transition: start := c

`ms_set_maze_end(c:cell)`
Exception: $\neg$is_init $\Rightarrow$ ms_not_initialized
$\neg$(cell_in_range(c)) $\Rightarrow$ ms_cell_out_of_range
Transition: end := c

`cell ms_get_maze_start()`
Exception: $\neg$is_init $\Rightarrow$ ms_not_initialized
$\neg$ms_set_maze_start $\Rightarrow$ ms_no_start
Output: start

`cell ms_get_maze_end()`
Exception: $\neg$is_init $\Rightarrow$ ms_not_initialized
$\neg$ms_set_maze_end $\Rightarrow$ ms_no_end
Output: end

`ms_set_wall(c1,c2:cell)`
Exception: $\neg$is_init $\Rightarrow$ ms_not_initialized
(wall is point) $\vee$ (wall is diagonal) $\vee$ (wall is out of range)
$\Rightarrow$ ms_not_valid_wall
Transition: maze := maze $||$ $<$ c1, c2 $>$

`Booleanms_is_connected(c1,c2:cell)`
Exception: $\neg$is_init $\Rightarrow$ ms_not_initialized
$\neg$(cell_in_range(c1)) $\Rightarrow$ ms_cell_out_of_range
$\neg$(cell_in_range(c2)) $\Rightarrow$ ms_cell_out_of_range
$\neg$(neighbour(c1, c2)) $\Rightarrow$ ms_not_neighbours
Output: $\exists$(wall between c1 and c2)

2.3 path_storage : MIS

Imported Data Types: cell
Boolean

Imported Constants: NUM_X_CELLS
NUM_Y_CELLS
MAX_NUM_CELLS

Exported Functions

NAME	INPUT	OUTPUT	EXCEPTION
ps_init			
ps_add_to_path	cell		ps_not_initialized
ps_get_next	Integer	cell	ps_not_initialized ps_index_out_of_range
ps_get_prev	Integer	cell	ps_not_initialized ps_index_out_of_range
ps_get_curr	Integer	cell	ps_not_initialized ps_index_out_of_range

State Variables

`path` : `sequence of cell`
`index` : `Boolean`
`is_init` : `Boolean`

Access Function Semantics

`ps_init()`

Transition: $\mathtt{path} := <>$
$\mathtt{index} := -1$
$\mathtt{is\_init} := \mathtt{true}$

`ps_add_to_path(c:cell)`

Exception: $\neg\mathtt{is\_init} \Rightarrow \mathtt{ps\_not\_initialized}$
Transition: $\mathtt{path} := \mathtt{path}||\mathtt{c}$
$\mathtt{index} := \mathtt{index} + 1$

`cell ps_get_next(pos:Integer)`

Exception: $\neg\mathtt{is\_init} \Rightarrow \mathtt{ps\_not\_initialized}$
$(\mathtt{pos}\ <\ 0 \vee \mathtt{pos}\ >\ \mathtt{index} - 2) \Rightarrow \mathtt{ps\_index\_out\_of\_range}$

Output: $\mathtt{path}[\mathtt{pos} + 1]$

`cell ps_get_prev(pos:Integer)`

Exception: $\neg\mathtt{is\_init} \Rightarrow \mathtt{ps\_not\_initialized}$
$(\mathtt{pos}\ <\ 1 \vee \mathtt{pos}\ >\ \mathtt{index} - 1) \Rightarrow \mathtt{ps\_index\_out\_of\_range}$

Transition: $\mathtt{pos} := \mathtt{pos} - 1$
Output: $\mathtt{path}[\mathtt{pos} - 1]$

`cell ps_get_curr(pos:Integer)`

Exception: $\neg\mathtt{is\_init} \Rightarrow \mathtt{ps\_not\_initialized}$
$(\mathtt{pos}\ <\ 0 \vee \mathtt{pos}\ >\ \mathtt{index} - 1) \Rightarrow \mathtt{ps\_index\_out\_of\_range}$

Output: $\mathtt{path}[\mathtt{pos}]$

2.4 load_maze : MIS

Imported Data Types: `cell`
`String`

Imported Access Functions: `ms_init`
`ms_set_start`
`ms_set_end`
`ms_set_wall /* from maze_storage */`
`read_cell`

Exported Functions

NAME	INPUT TYPE	OUTPUT TYPE	EXCEPTION
lm_load_maze	`String`		lm_file_error

State Variables

`f : file`

Access Function Semantics

`lm_load_maze(filename : String)`

Exception: error opening, reading, file format $\Rightarrow$ `lm_file_error`

Transition:
```
f := open(filename)
ms_set_maze_start(read_cell)
ms_set_maze_end(read_cell)
until end of f do
  ms_set_wall(read_cell, read_cell)
od
```

2.5 find_path : MIS

Imported Data Types: `cell`, `Boolean`

Imported Constants: `NUM_X_CELLS`, `NUM_Y_CELLS`

Imported Access Functions: `ms_get_maze_start`, `ms_get_maze_end`, `ms_is_connected`, `ps_add_to_path`

Exported Functions

NAME	INPUT	OUTPUT	EXCEPTION
fp_find_path		Boolean	

State Variables

`path:sequence of cell`

Access Functions

`Boolean fp_find_path()`

Output: $\exists \mathtt{path},\ \mathtt{path}[0] = \mathtt{ms\_get\_maze\_start}() \wedge$
$\mathtt{path}[|\mathtt{path}| - 1] = \mathtt{ms\_get\_maze\_end}() \wedge$
$(\forall \mathtt{i}, 0 \leq \mathtt{i} \leq |\mathtt{path}| - 2, \mathtt{ms\_is\_connected}(\mathtt{path}[\mathtt{i}], \mathtt{path}[\mathtt{i}+1])) \wedge$
$(\forall \mathtt{i}, 0 \leq \mathtt{i} \leq |\mathtt{path}| - 2, \mathtt{ps\_add\_to\_path}(\mathtt{path}[\mathtt{i}]))$

2.6 hardware : MIS

Imported Data Types: `Boolean`, `button`

Imported Access Functions: `o_init`, `o_power`, `o_penDown`, `o_penPos`, `i_stopButton`, `i_homeButton`, `i_backButton`

Exported Functions

NAME	INPUT TYPE	OUTPUT TYPE	EXCEPTION
hw_init			hw_init_error
hw_power	`Integer`		hw_not_initialized hw_power_error
hw_pen	`Integer`		hw_not_initialized hw_power_not_on hw_pen_error
hw_move	`Integer`,`Integer`		hw_not_initialized hw_power_not_on hw_move_error
hw_check		button	hw_not_initialized hw_power_not_on hw_button_error

State Variables

`stop_flag` : `Boolean`
`pwr_flag` : `Boolean`
`is_init` : `Boolean`

Access Function Semantics

`hw_init()`

Exception: $\texttt{o\_init}() \neq 0 \Rightarrow \texttt{hw\_init\_error}$
Transition: $\texttt{is\_init} := \texttt{true}$

`hw_power(power:Integer)`

Exception: $\neg\texttt{is\_init} \Rightarrow \texttt{hw\_not\_initialized}$
$\texttt{o\_power}(\texttt{power}) \neq 0 \Rightarrow \texttt{hw\_power\_error}$
Transition: if (power)
then $\texttt{pwr\_flag} := \texttt{true}$
else $\texttt{pwr\_flag} := \texttt{false}$

`hw_pen(pen:Integer)`

Exception: $\neg\texttt{is\_init} \Rightarrow \texttt{hw\_not\_initialized}$
$\neg\texttt{pwr\_flag} \Rightarrow \texttt{hw\_power\_not\_on}$
$\texttt{o\_penDown}(\texttt{pen}) \neq 0 \Rightarrow \texttt{hw\_pen\_error}$
Transition: $\texttt{o\_penDown}(\texttt{pen})$

`hw_move(x,y:Integer)`

Exception: $\neg\texttt{is\_init} \Rightarrow \texttt{hw\_not\_initialized}$
$\neg\texttt{pwr\_flag} \Rightarrow \texttt{hw\_power\_not\_on}$
$\texttt{o\_penPos}(\texttt{x},\texttt{y}) \neq 0 \Rightarrow \texttt{hw\_move\_error}$
Transition: $\texttt{o\_penPos}(\texttt{x},\texttt{y})$

`button hw_check(x,y:Integer)`

Exception: $\neg\texttt{is\_init} \Rightarrow \texttt{hw\_not\_initialized}$
$\neg\texttt{pwr\_flag} \Rightarrow \texttt{hw\_power\_not\_on}$
Transition: if i_stopButton
$\texttt{stop\_flag} := \texttt{true}$

Output: STOP if i_stopButton
HOME if i_homeButton
BACK if i_backButton

2.7 geometry : MIS

Imported Data Types: cell
Boolean
button

Imported Access Functions: hw_check
hw_move

Exported Functions

NAME	INPUT TYPE	OUTPUT TYPE	EXCEPTION
gm_go	cell		

Access Function Semantics

gm_go(c:cell)

Transition: if hw_check $\neq$ STOP, hw_move(convert(dest))

2.8 control : MIS

Used External Data Types: Boolean, button, cell

Used External Modules & Functions: hardware, path_storage, maze_storage, geometry

Exported Functions

NAME	INPUT TYPE	OUTPUT TYPE	EXCEPTION
cn_execute			

State Variables

mode : {init, start, forward, reverse, home, done}
back_flag : Boolean
pos : Integer

State Transformations

MODE	CONDITION	ACTION	NEW MODE
init	lm_file_error		mode := done
	¬`fp_find_path`		mode := done
	fp_find_path	hw_init() hw_power(ON) hw_pen(UP)	mode := starting pos := 0 back_flag := FALSE
starting	hw_check() = NONE	gm_go(ps_get_curr(pos)) hw_pen(down)	mode := forward
	hw_check() = STOP		mode := starting
forward	back_flag = TRUE		back_flag := false
	hw_check() = NONE	gm_go(get_next(pos))	pos := pos + 1 mode := forward
	hw_check() = STOP		mode := forward
	hw_check() = BACK	gm_go(ps_get_prev(pos))	pos := pos - 1 back_flag := TRUE mode := reverse
	hw_check() = HOME		mode := home
	o_penPos = ms_get_end ∧ back_flag = FALSE		mode := home
reverse	back_flag = TRUE hw_check() = NONE	 gm_go(ps_get_next(pos))	 pos := pos + 1 mode := forward
	hw_check() = STOP		mode := reverse
	hw_check() = BACK	gm_go(ps_get_prev(pos))	pos := pos - 1 mode := reverse
	hw_check() = HOME		mode := home
home	hw_check() ≠ STOP	hw_pen(UP)	mode := home
	hw_check() = STOP		mode := home
	o_penPos = HOME		mode := done
done		hw_power(OFF)	quit program

2.9 types_constants : MIS

Exported Types: `cell = tuple (x:Integer,y:Integer)`
`Boolean= {TRUE,FALSE}`
`String= sequence of char`
`button = set of {STOP,HOME,BACK,NONE}`

Exported Constants: `UP`
`DOWN`