SOFTWARE ENGINEERING

BUG WORLD SPECIFICATION AND DESIGN

Pratham Shah
Ayam Banjade
Harshit Mutha
Haris Muratovic
Timur Pashakulov

GENERAL INFORMATION

PURPOSE

The purpose of this document is to implement a Web environment for the simulation of a bug game to observe two swarms of bugs (e.g., red and black bugs) with their own instruction set in a hunting competition (for food) lasting a specific amount of time.

SCOPE

Bug World is an implementation of the above-mentioned Web game that attempts to subdivide the game into:

- GUI: screen and interaction management
- Assembler: generating an executable bug program from assembler input
- Simulator: the engine executing the bug codes and updating its world state

Additional functionality includes:

- **to_string():** method defined in every class, which outputs the current internal object state in human-readable format
- **rand(n)**: a pseudo-random nonnegative integer in the range *[0, n-1]*

The goal is to run an experiment, also called tournament, which consists of two runs where the two bugs compete. They switch starting positions on the second run. Two runs are made to alleviate the advantage of any particular landscape.

During the entire experiment, the state of the world will be constantly changed, and these changes need to be able to be viewed in the GUI and the logging ability must also be provided.

To conduct unique tournaments, an ability to define the environment and a specific amount of time along with additional options and functionality must also be implemented.

IMPLEMENTATION

TECHNOLOGIES

- *User-Interface:* for the user interface of Bug World the technologies used are
  - HTML: helps build structured web pages / documents and directly introduce content
  - CSS: helps style and present the web page and its content
  - JavaScript: helps add dynamic capabilities, further functionality, and interaction on the web page

- *Application:* for the application of Bug World the technologies used are:
  - JavaScript

ARCHITECTURE

As mentioned above, Bug World will be subdivided into three main layers:

1. GUI

Manages the screen, accepts user input and displays responses by invoking assembler and simulator.

For **inputs** it will receive:

- a world map file (.*world* file),
- two bug assembler source code files (.*buggy file).* The bugs do not need to be called red and black.

Various additional options can also be inputted:

- the number of ticks / iterations / cycles (**mandatory**)
- various optional controls like log output, state information every *i*-th cycle, etc. (**optional**)

Every tick, it will **output** a visual representation of:

- the current tick
- remaining ticks
- world map
- log output (if enabled)
- a summary of current tournament status (amount of undetected food; for red and black bugs: food brought home, bugs killed / remaining, etc.)

There should also be a way to load just one bug assembler program and show the machine code generated otherwise the errors.

2. Simulator

Generated or created from three files: the world file (*.world* file), red bug file (*.bug* file), and black bug file (*.bug* file). It parses bug code, provides defaults, and holds everything together. It consists of two main modules:

- **Engine**: defines operational semantics of the bug machine language. The simulation lasts for n cycles; every cycle the simulator is logged if logging is activated. The simulation, at the end, responds with statistics about the state of the world.
- **Tournament**: As mentioned above, it creates a two-game competition in each environment, called a tournament. The swarms will switch starting positions in the second game. Here, a bug will receive 2 points if it wins, a draw will result in 1 point and a loss gets 0 points.

The run function receives file names for the environment world and the bugs, characterized by their machine instructions. The result is the number of points each bug has won in the tournament.

3. Assembler

Reads assembler code (*.buggy* file) (for correct grammar see **Appendix**), translates to bug code / machine instruction (for correct syntax of instruction set see **Appendix**) for the simulator and emits it onto the GUI. Primarily, it provides a parser read for assembler instructions. It does two passes over the assembler code:

- **First pass:** assigns addresses to labels (*address resolution*)
- **Second pass:** translates assembler instruction to machine instruction. Notably, a ***goto*** that follows an instruction can be eliminated by making the target of ***goto*** the successor of the instruction. If ***goto*** is the first instruction, implement it using ***flip***.

The ***assemble()*** method implements both passes.

The labels should have a symbol table; the labels are denoted by a state (integer). Parser reads line by line from input file until *EOF* in a loop – and pattern matches over the list of tokens. *Exception:* **cond**, which may have 1-2 tokens. As **cond** is last element of sense – the remaining tokens can just be passed to **cond**.

DESCRIPTION

WORLD

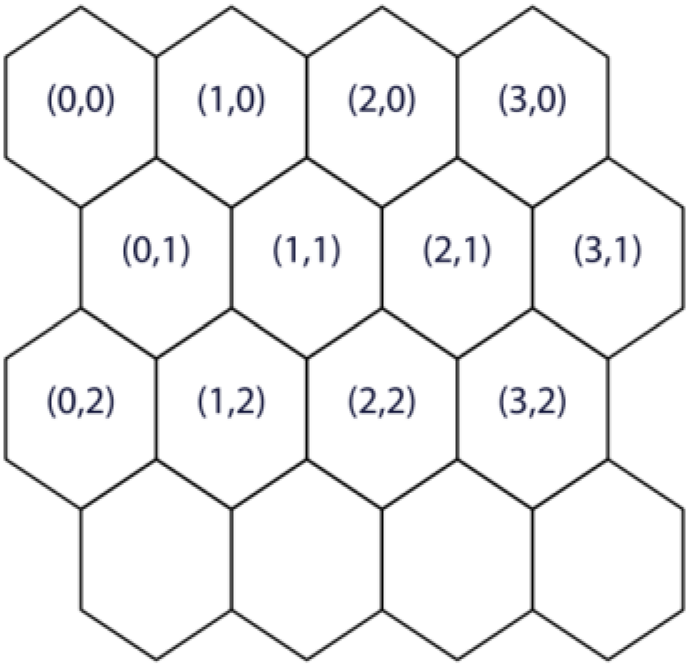
2D flat environment of contiguous hexagonal cells containing bases / nest of swarm for food depositing; bugs; food, and obstacles. The world simulation proceeds in ticks – which executes instructions for each bug and updates its and the world state accordingly.

BUGS

In this world, two types of bugs compete against each other to collect the most food at a specific time. Bugs can also leave traces (path bug has taken) and up to 6 markers. A marker has an *id*, *color,* and *position*. Both can be viewed by the enemy, but the enemy cannot view the marker id.

CELLS

Each cell is identified through its *(x,y)* position. The top-left corner cell has position *(0,0)*, coordinates increase to bottom and right (see example below). A cell is either free or obstructed. A free cell can contain at most one bug, a non-negative number of food packages and markers from each swarm.



DEBUGGING

The state of the world can be written into log file (either icfp log or sopra log). A stats function returns a record of statistics per bug colony: the number of alive bugs, and the amount of food (e.g., bits) placed on the colony’s base.

Requirement Analysis

USER REQUIRMENTS

1. User must provide valid .word file.
2. User must provide valid .buggy files.
3. User must be able to provide options for simulation.
4. User must be able to change number of ticks(cycles).

SYSTEM REQUIRMENTS

1. User must have JavaScript enabled.
2. User must have a Browser.
3. User must have an Internet connection.

FUNCTIONAL REQUIRMENTS

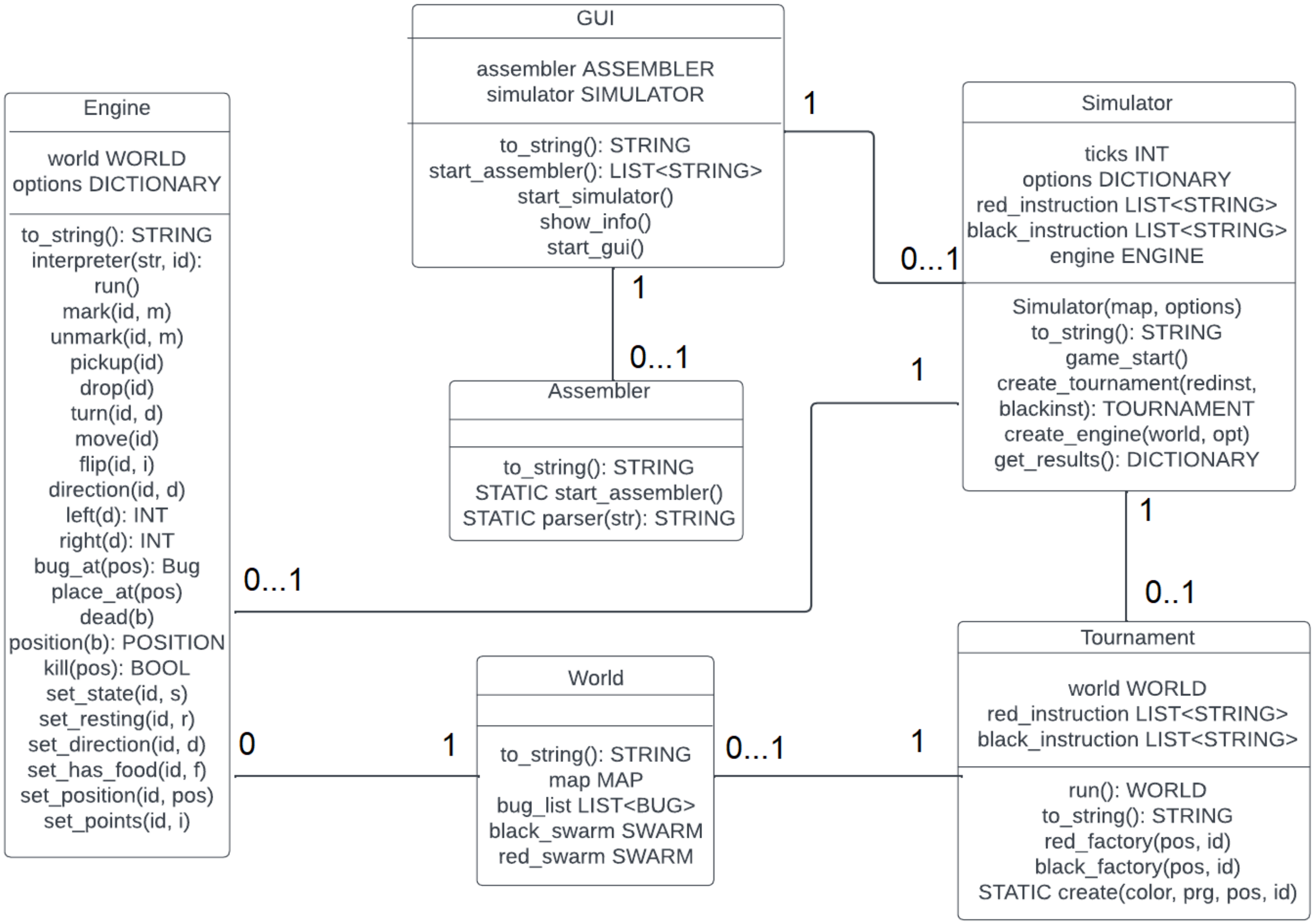
1. The simulation must have 2 bug's nests and two bug types.
2. The bug must be able to make actions: to move, turn, pick up and drop food, set and unset markers, kill other bugs and check cell or sense for one of these parameters: “Friend”, “Foe”, “FriendWithFood”, “FoeWithFood”, “Food”, “Rock”, “Marker” int, “FoeMarker”, “Home”, “FoeHome”.
3. For every tick, instruction sets must be executed sequentially based on the Bugs’ id (ascending).
4. The bug can make only one action per tick.
5. The bug must rest for 14 ticks after making an action.
6. The simulation must provide logs if asked by options (see **Appendix**)
7. The simulation must show a leaderboard after the last tick.
8. By default, simulation will run for 1000 ticks.
9. The bug behavior must be described as finite automata.
10. The bug must be able to tell whether the cell is free.
11. The assembler must notify the user if *.buggy* file is not valid.
12. The assembler must notify the user if *.world* file is not valid.
13. The maximum value of markers for each nest must be 6.
14. The bug can carry at most one food package at a time.
15. Each cell can have only non-negative number of food packages.
16. Absolute direction in the world is encoded as an integer

17. Each position has up to six neighbours
18. The bug can only change its direction left or right relative to its current heading
19. The bug can cannot view the value of adversarial markers
20. On creation, the bug must face direction 0, carry no food, be in state 0 and need no resting (0).
21. All opcode and symbolic constants need to be mapped to single-byte numbers. State requires two bytes to allow for more than 255 states.

NON-FUNCTIONAL REQUIRMENTS

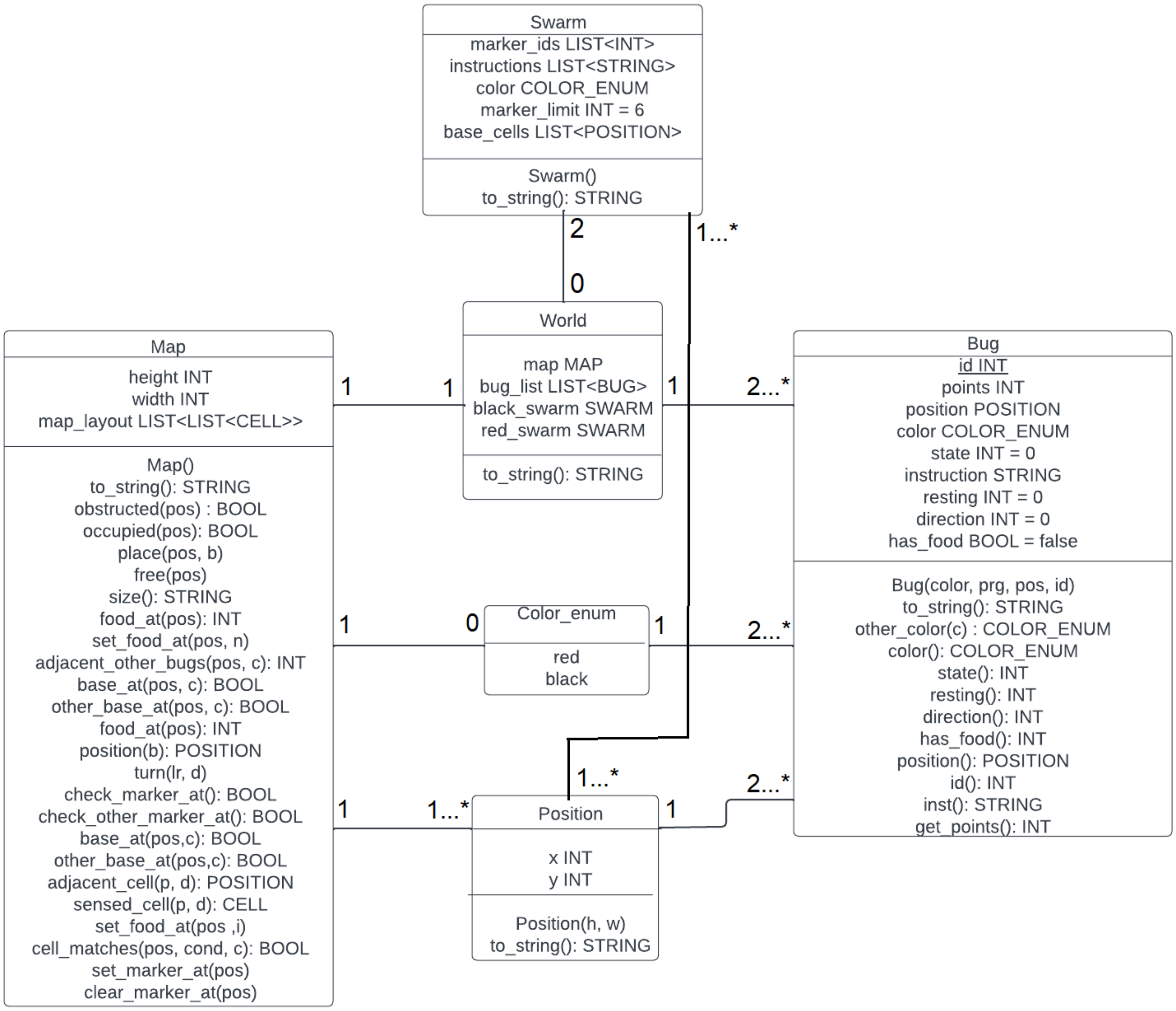
1. The Website must support N users at the same time.

CLASS DIAGRAMS

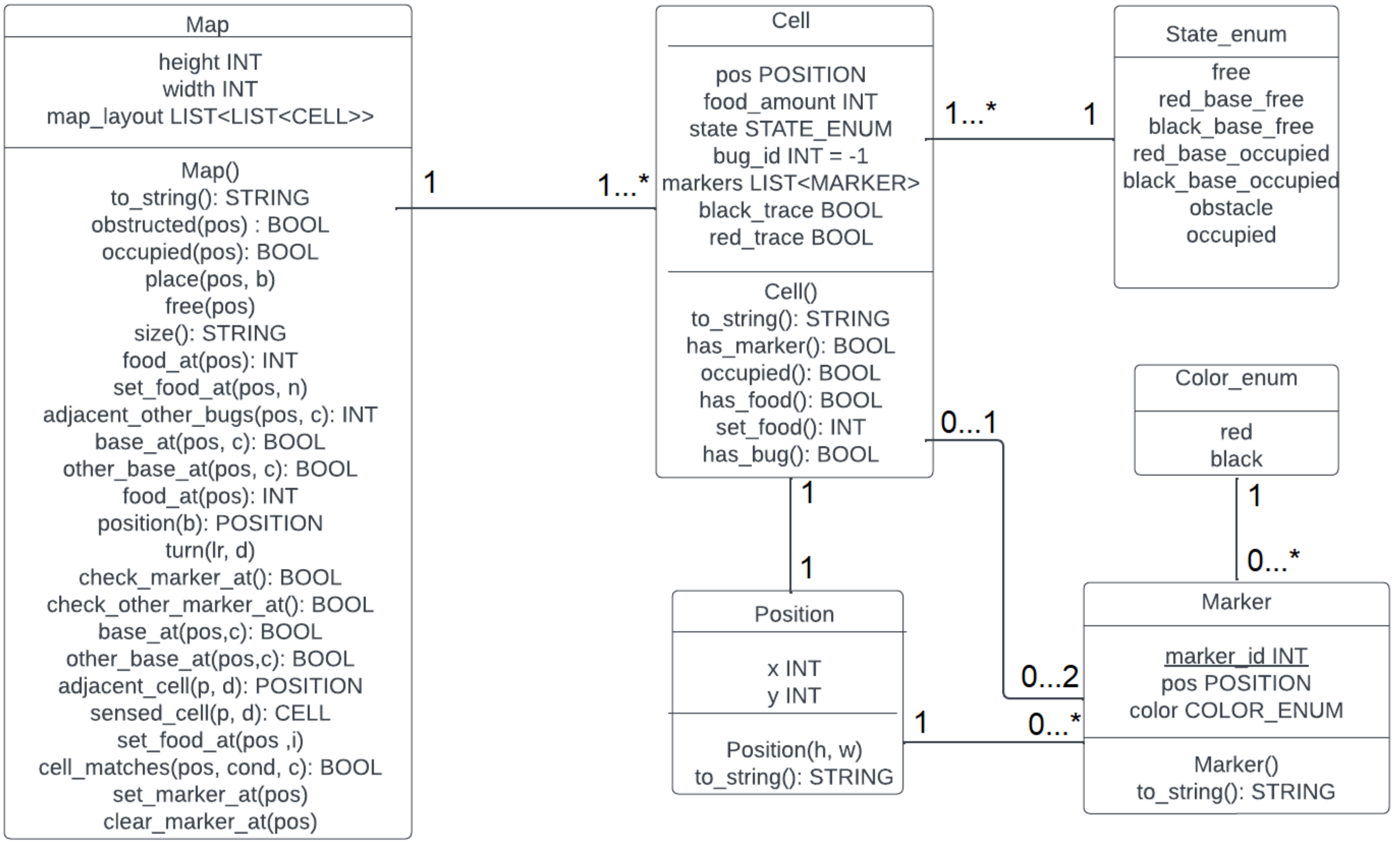


The GUI is the main class which will essentially start everything. First, the assembler and the simulator get created by the GUI. The former actually starts the assembler, while the latter commences the game creation, and creates the engine and the tournament. Then, the tournament creates the world which then gets passed on to the engine, which allows it to proceed further

.



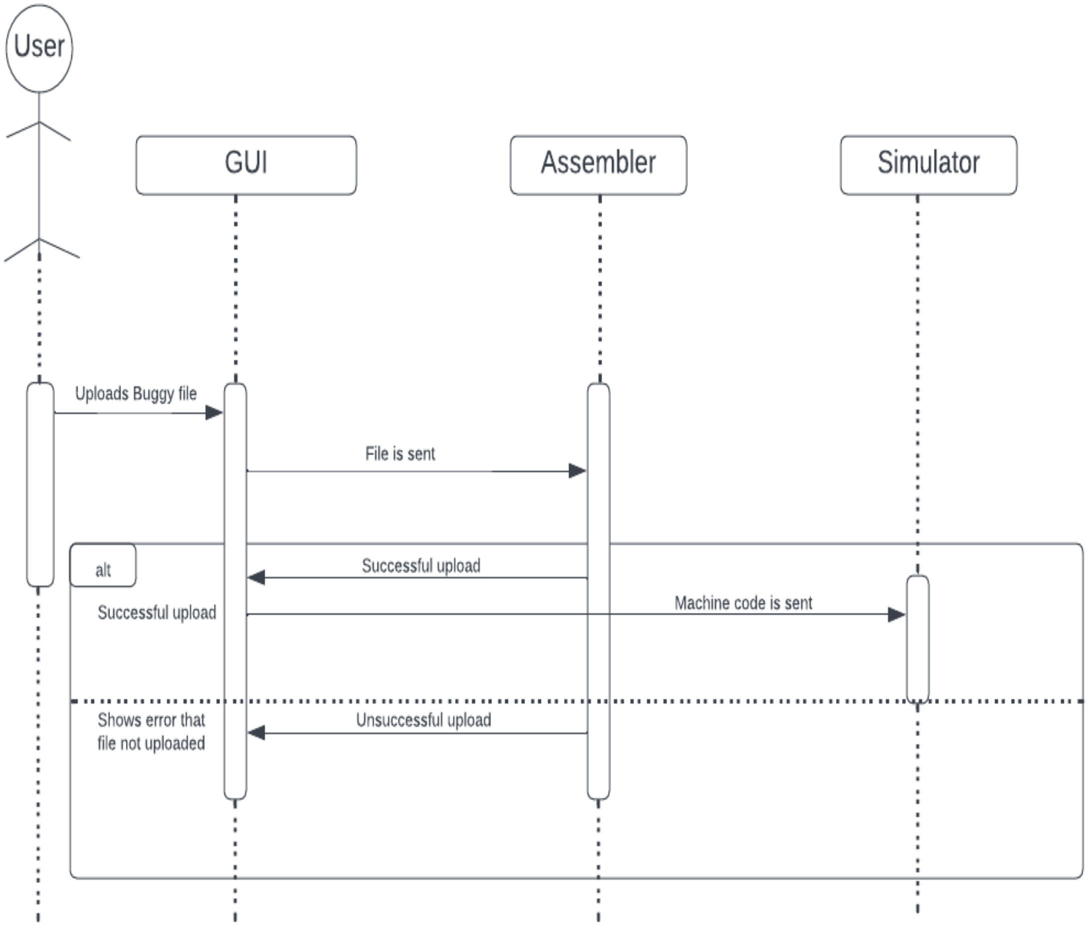
The World class from the previous diagram is shown here as well. It contains two swarms, many bugs, and just one map. A color enumeration is displayed also – it was a better choice than, say, a string containing the color, since only two colors can be given: red and black. The Position class is contained within the map, bug and swarm.



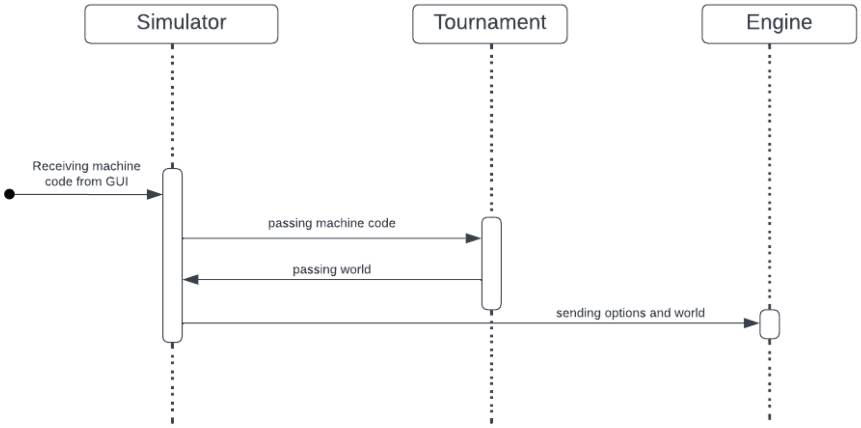
Similarly to the aforementioned diagram, the Map class shows up once again in this diagram. It contains a 2D List made up of cells. Cells have a state enum, a list of markers, and a position. A marker also has a position and color enum.

SEQUENCE DIAGRAM

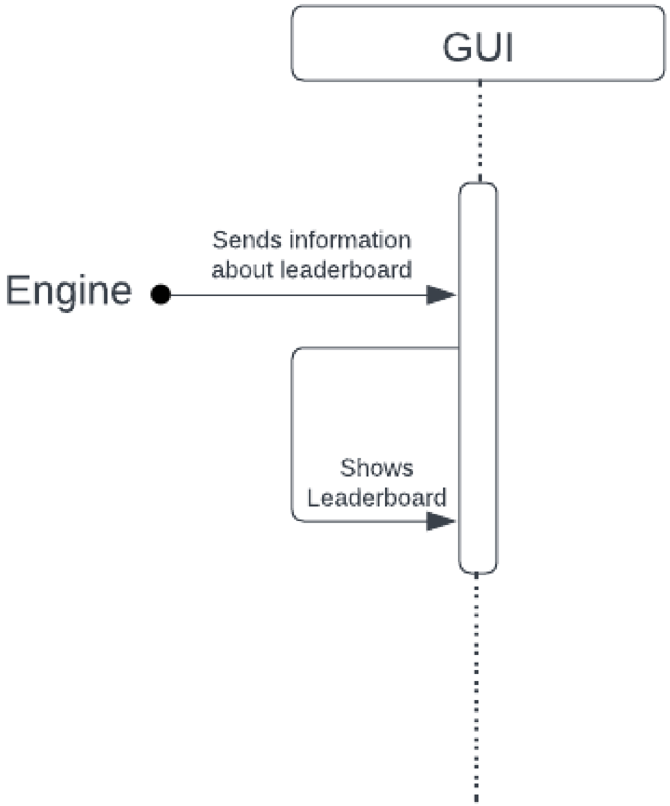
Sequence diagrams display the flow of order which is when and how the objects interact with each other



The user first uploads .buggy files in the GUI. Then the GUI sends the file to assembler which converts the .buggy files into the bug instructions (assembler code to machine code), if the conversion was successful then the user gets a message on the GUI saying that the upload was successful and the machine code is sent to the simulator, however if the conversion was not successful then the user sees an error message on the GUI.



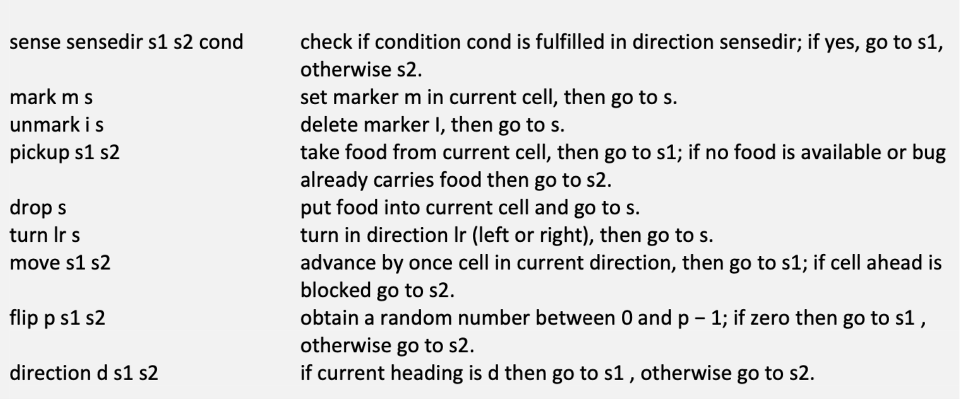
The GUI sends the machine code (the instructions for bugs, and .world file) to the simulator, which then passes it to the tournament to create a world. Then this world is passed to the engine class, which then will start a simulation.



After the simulation has ended, the engine sends the score of the bugs to the GUI, which then displays it to the user in the form of a scoreboard/leaderboard.

APPENDIX

INSTRUCTION SET

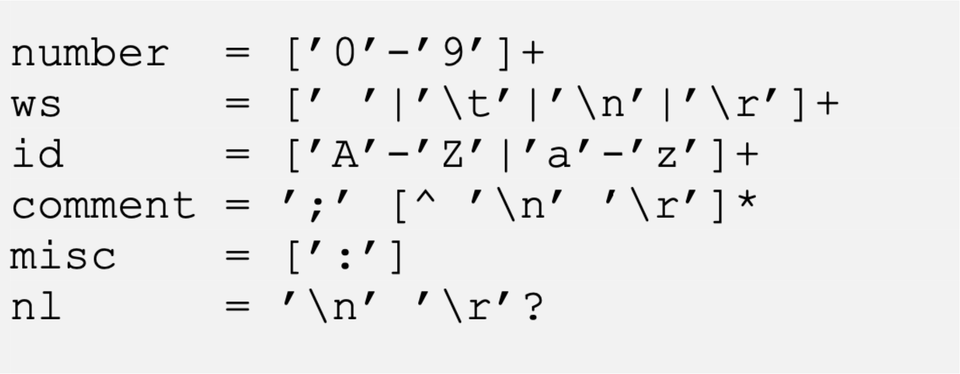


sense sensedir s1 s2 cond	check if condition cond is fulfilled in direction sensedir; if yes, go to s1, otherwise s2.
mark m s	set marker m in current cell, then go to s.
unmark i s	delete marker I, then go to s.
pickup s1 s2	take food from current cell, then go to s1; if no food is available or bug already carries food then go to s2.
drop s	put food into current cell and go to s.
turn lr s	turn in direction lr (left or right), then go to s.
move s1 s2	advance by once cell in current direction, then go to s1; if cell ahead is blocked go to s2.
flip p s1 s2	obtain a random number between 0 and p − 1; if zero then go to s1 , otherwise go to s2.
direction d s1 s2	if current heading is d then go to s1 , otherwise go to s2.

ASSEMBLER GRAMMAR

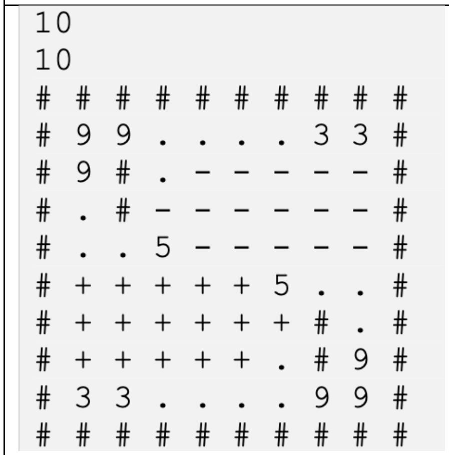
```
program ::= instruction+
instruction ::=
     | “Sense” dir cond “then” label “else” label
     | “Mark” int “then” label
     | “Unmark” int “then” label
     | “PickUp” “then” label “else” label
      | “Drop” next
     | “Turn” leftright
     | “Move” “then” label “else” label
     | “Flip” int “then” label “else” label
     | “Direction” int “then” label “else” label
     | string “:”
label ::= string | number
dir ::= “Here” | “LeftAhead” | “RightAhead” | “Ahead”
leftright ::= “Left” | “Right”
cond ::=
    | “Friend”
     | “Foe”
| “FriendWithFood”
     | “FoeWithFood”
     | “Food”
     | “Rock”
     | “Marker” int
     | “FoeMarker”
     | “Home”
     | “FoeHome”
```

SYMBOL CLASSES



```
number  = [’0’-’9’]+
ws      = [’ ’|’\t’|’\n’|’\r’]+
id      = [’A’-’Z’|’a’-’z’]+
comment = ’;’ [^ ’\n’ ’\r’]*
misc    = [’:’]
nl      = ’\n’ ’\r’?
```

TEST CASES

Valid **.***world* file	Invalid **.***world* file
 10 10 # # # # # # # # # # # 9 9 . . . . 3 3 # # 9 # . - - - - - # # . # - - - - - - # # . . 5 - - - - - # # + + + + + 5 . . # # + + + + + + # . # # + + + + + . # 9 # # 3 3 . . . . 9 9 # # # # # # # # # # # Expected reaction from system: *Accepted world file.*	10 -10 # # # # # # # # # # * . . . . . . 9 8 9 - .- + . .+ $ $ $ # # # # # # # # # # Expected reaction from system: *Invalid world file! Please make sure to use valid dimensions, characters and properties.*

Valid .*buggy* file	Invalid .*buggy* file
search:	search:
sense ahead food else walk	sense ahead food else run
move else search	move else search
pickup else search	jump else search
goto home	goto albania
walk:	walk:
flip 3 else searchright	flip 3 else 3
turn left	turn behind
goto search	goto tokyo
searchright:	Ulqin:
flip 2 else searchstraight	move up 3
turn right	turn left
goto search	
searchstraight:	searchstraight:
move else walk	demarcus cousins the 3rd
goto search	goto search
home:	home:
sense ahead home else	sense ahead school else
walkhome	walkschool
move else home	move else home
drop	drop
goto search	goto search
walkhome:	walkhome:
flip 3 else walkhomeright	bababooey
turn left	Flip 3 else bababooey
goto home	Turn righeft
walkhomeright:	walkhomeright:
flip 2 else	dab 3 else home
walkhomestraight	walkhomestraight:
turn right	turn back
goto home	goto homie
walkhomestraight:	walkhomestr8:
move else walkhome	move else walk to Dhangadhi
goto home	Goto Kathmandu

Expected output by system:	Expected output by system:

Machine code (passed onto simulator):
`sense ahead 1 3 food ;[0]`
`move 2 0 ;[1]`
`pickup 8 0 ;[2]`
`flip 3 4 5 ;[3]`
`turn left 0 ;[4]`
`flip 2 6 7 ;[5]`
`turn right 0 ;[6]`
`move 0 3 ;[7]`
`sense ahead 9 11 home ;[8]`
`move 10 8 ;[9]`
`drop 0 ;[10]`
`flip 3 12 13 ;[11]`
`turn left 8 ;[12]`
`flip 2 14 15 ;[13]`
`turn right 8 ;[14]`
`move 8 11 ;[15]`
	ERROR! Invalid goto. Invalid grammar. Please fix.

LOG FILE FORMAT

```
    After cycle 0...
    ========== cell ========== ======= bug ======
            b b
            a i                    cbd
    pos pos s t  red    black      oii
     x   y  e  s  marks  marks  id  ltr state rest
    === === = == ====== ====== === === ===== ====
    ...
    001 007 r 02 543__0 5_3___
    002 007 r 03 5___1_ 5_3210
    003 007 r 00 __3__0 __3210 010 r_4 00033 0000
    004 007 r 00 _4__10 54_210 005 rX3 00002 0000
    005 007 r 00 __321_ 5___1_
    006 007 _ 02 _4_2__ 54__1_ 012 r_4 00019 0000
    007 007
    008 007 _ 09 5__2__ 5__2_0
    ...
```

BIBLIOGRAPHY

https://peter-baumann.org//Courses/SoftwareEngineering/index.php

https://lucid.app/

Bug World Description – Peter Baumann

Bug World Assembler Manual - Peter Baumann

Bug World Simulator Manual – Peter Baumann