|  |
| --- |
| Carleton University |
| Plants vs. Zombies Model and Simulation with DEVS |
| SYSC5104 |

|  |
| --- |
| Wang Xiang 100680824  11/2/2010 |

# Introduction:

In this assignment, the game “Plants vs. Zombies” will be modeled based on the concept of DEVS. The conceptual model, system specification, project design, project implementation, and project testing will be presented in this report.

“Plants vs. Zombies” is a very famous and successful flash game released in the recent years. The aim of the game is to grow different kinds of plants to defend zombie intruders. On the other hand, one of its mini-game is about deploy different zombies to penetrate a computer generated plants defence squad. See Figure 1 below for an illustration.



Figure 1: "Plants vs. Zombies" mini-game

The complexity of such game is quite large. Different zombies and plants all have different functions, and zombies needed to be purchased from gathered money (sunflowers) from destroy the plant, and many more. To simplify the problem to become feasible for the assignment purpose, the scope of the project will only deal with a largely simplified scenario of the game.

# Project Definition and Conceptual Design:

This project will focus on modeling a single zombie’s behaviour in the game. For this purpose, a yard of size (10\*5) is modeled with a matrix of size (11\*5). One line of single type plants are pre-grown at (1, \*). The goal of the simulation is to examine if a zombie can reach (0, \*). Having the background of the experimental frame set up, the design of the project will randomly spawn one single zombie on the yard, and it is immediately subject to taking damage from the plants. An illustration is given below:



Figure 2: Modeled game play

Although the scope of this project only models one type of zombie, it is desirable to design the model in a way that can be generalized for all kinds of zombies. Different types of zombies are characterized by their health amount, movement speed, and biting rate. Although some zombies have some special ability such as jump over a plant, that special ability is not modeled in this project. Thus a coupled model named “Single Zombie” would contain three atomic models named “Health”, “Speed”, and “Bite”. In order to model the behaviour of plants shooting bullets, a random generator is implemented with an atomic model. Lastly, an atomic model named “Yard” is in charge of keeping track of zombie’s location, and notifies the “Single Zombie” model for being hit, or should start biting the plant.

Having the scope of the project and the functions of each block understood the structure of the DEVS model is illustrated in Figure 3, and the DEVS formal specifications of each block will be given in next section.

# DEVS Formal Specification and Testing Strategies:

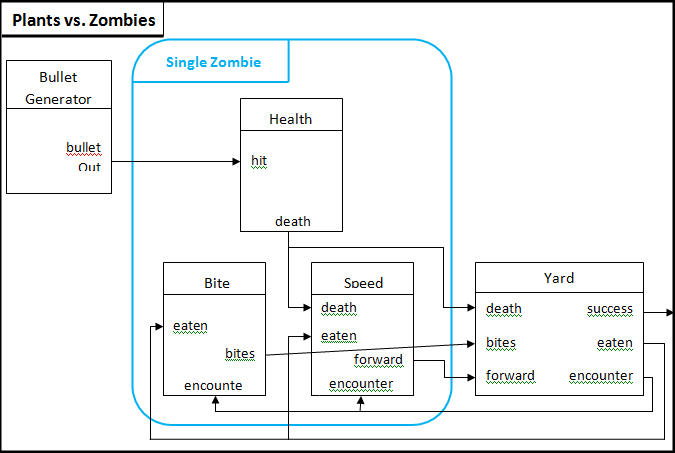


Figure 3: DEVS model structure

## Plants\_vs\_Zombies

Figure 3 shows the DEVS model structure for the project. The top level named “Plants vs. Zombie” takes no input, but outputs a boolean value indicating whether the zombie successfully penetrated the defence line. The formal DEVS specification is given below:

Plants vs. Zombies = <X, Y, D, M, EIC, EOC, select>

X = {};

Y = {success, S[0,1]};

D = {BulletGenerator, SingleZombie, Yard};

M = {BulletGenerator, MsingleZombie, Yard};

EIC = {};

EOC = {(Yard.success, self.out)};

IC = {(BulletGenerator.bulletOut, SingleZombie.hit), (SingleZombie.death, Yard.death), (SingleZombie.bites, Yard.bites), (SingleZombie.forward, Yard.forward)};

Select: ({BulletGenerator, SingleZombie, Yard}) = BulletGenerator;

({BulletGenerator, SingleZombie}) = BulletGenerator;

({BulletGenerator, Yard}) = BulletGenerator;

({SingleZombie, Yard}) = SingleZombie;

The top model does not require any input and only generates an output of value 1 if zombie intrusion succeeds or a value of 0 if not. There will be no event file supplied to this model. To test this model, many times the simulation should be executed to see when the output is generated and what the output is. One possible mistake of the implementation could be the model outputting twice in one simulation which should not be the case.

## BulletGenerator

The BulletGenerator is an atomic model that generates “bullets” at random time. This mimics the situations where many plants are shooting at the zombie.

BulletGenereator = <X, Y, S, , , , ta>

X = {};

Y = {bulletOut = 1};

S = {active, passive};

(S, e, x) {};

(InternalMessage &msg) {holdIn(active, random\_time);}

λ(InternalMessage &msg) {sendOutput(msg.time(), bulletOut, 1);}

This model does not require an input as well. To examine the correctness of this block is to run the simulation for a period of time and see if the outputs are at random time apart from each other.

## SingleZombie

The SignleZombie block is a coupled model. It models the characteristics of one zombie including its health, speed, and bite rate.

SingleZombie = < X, Y, D, M, EIC, EOC, select >

X = {bulletOut, encounter, eaten};

Y = {death, bites, forward};

D = {Health, Bite, Speed };

M = { Health, Bite, Speed };

EIC = {(self.hit, Health.hit), (self.encounter, Bite.encounter), (self.encounter, Speed.encounter), (self.eaten, Bite.eaten), (self.eaten, Speed.eaten)};

EOC = {(Health.death, self.death), (Bite.bites, self.bites), (Speed.forward, self.forward)};

IC = {(Health.death, Speed.death)};

Select: ({Health, Bite, Speed}) = Health;

({Health, Bite}) = Health;

({Health, Speed}) = Health;

({Bite, Speed}) = Speed;

This coupled model is the most complex model of this project. Many possible input/output combinations can cause many different behaviours of the model. For example, after the zombie is dead, this model should not continue generate the “forward” output anymore. However, the “eaten” input will restart the generation once this input is received. The solution of this problem is however not covered in this block but covered in the Yard model which will be explained later. Therefore, such behaviour will be considered as correct for the SingleZombie block. Thus the event file supplied for testing this block should examine:

* If the “death” output can be generated
* If the “bites” output can be generated at a constant rate
* If the “forward” output can be generated at a constant rate
* If the “death” output is generated, the generation of “forward” output should be stopped
* If the “ate” input can trigger the generation of “forward” output again
* If the “encounter” input can stop the generation of the “forward” output
* If the “encounter” input can trigger the generation of “bites” outputs
* The effects of simultaneous inputs

## Health

Health is then a passive atomic model. It reacts to the input of being hit, and outputs a value 1 after its health reaches 0.

Health = <X, Y, S, , , , ta>

X = {hit = 1};

Y = {death = 1};

S = {active, passive};

(S, e, hit) {

If (health > 0) {health = health – 1;}

}

(InternalMessage &msg) {passivate();}

λ(InternalMessage &msg) {

if (health == 0) {sendOutput(msg.time(), death, 1); //for only one time}

}

This block is only in charge of reducing the health of the zombie. It only generates an output when zombie’s health reaches zero. Therefore, only possible mistake to test this block is to see whether this model keeps on reducing the health after the health reaches zero.

## Speed

The output of Health block is then fed into the Speed block which is an active atomic model. Speed block outputs a value 1 at a constant rate to inform the Yard block to update the zombie’s position. It will stop output upon receiving an input of death and/or encounter signal. It will then restart generating output again after receiving an eaten signal signifying the Zombie has eaten the plant and so able to move forward again.

Speed = <X, Y, S, , , , ta>

X = {encounter, death, eaten};

Y = {forward = 1};

S = {active, passive, stop };

(S, e, encounter, death, eaten) {

If (receive encounter input) {stop = true;}

If (receive death input) {stop = true;}

If (receive eaten input) {stop = false;}

}

(InternalMessage &msg) {holdIn(active, constant\_time);}

λ(InternalMessage &msg) {

if (!stop) {sendOutput(msg.time(), forward, 1); }

}

To test this block, it is important to test if the “stop” field is at conflict upon receiving simultaneous inputs. For example, if receiving both the death and eaten input at the same time, this model should not set the “stop” to true first then to false.

## Bite

The Bite block is again another atomic model that models the bite rate of a particular zombie. It will start generating “bites” output at a constant rate upon receiving an “encounter” input. It then will then stop generating output upon receiving an “eaten” input because the zombie has already eaten the plant.

Bite = <X, Y, S, , , , ta>

X = {encounter, eaten};

Y = {bites = 1};

S = {active, passive, start };

(S, e, encounter, eaten) {

If (receive encounter input) {start = true;}

If (receive eaten input) {stop = false;}

};

(InternalMessage &msg) {holdIn(active, constant\_time);}

λ(InternalMessage &msg) {

if (start) {sendOutput(msg.time(), bites, 1); }

}

The testing strategy is similar to the speed block. It needs to verify that upon receiving simultaneous inputs, the “start” state does not flip. Also need to see if it generates output at a constant rate.

## Yard

Lastly, the Yard block is an atomic model that captures the positions of plants and the zombie. It informs the SingleZombie block that the zombie has came next to a plant. Due to the simplicity of this project design, this block also in charge of keeping track of the health of the plant which is in the same row as the zombie. The block will inform the SingleZombie block after the zombie drained the plant’s health. Also, this block is the final stage of the design. It outputs a “success = 1” signal if the zombie is able to reach the end of the yard (left most side), or outputs “success = 0” if the zombie is killed halfway.

Yard = <X, Y, S, , , , ta>

X = {forward, death, bites};

Y = {encounter, eaten, success};

S = {active, passive, [x-cord, y-cord], plantHealth};

(S, e, forward, death, bites) {

If ((receive forward input) && (x-cord > 0)) {x-cord -= 1;}

If (receive eaten input) {stop = false;}

};

(InternalMessage &msg) {holdIn(active, constant\_time);}

λ(InternalMessage &msg) {

if (start) {sendOutput(msg.time(), bites, 1); }

}

To test this block, a few test cases should be used separately. First, test if the block is able to arrive at the “success” state. Second, test if the block can deal with simultaneous inputs, especially for receiving simultaneous “death” and “forward” inputs, also for simultaneous “bite” and “forward” inputs. There are many other possibilities of simultaneous inputs and input/internal state conflict. However, these conflicts should be solved if other blocks behave correctly.

# Testing of the project implementation

In this section the testing cases and their results will be presented. Explanations will also be given for each result for its behaviour.

## BulletGenerator

This block does not require any input, thus no event file is supplied. Simulation result is provided below:

00:00:00:000 bullet 1

00:00:02:000 bullet 1

00:00:11:000 bullet 1

00:00:11:000 bullet 1

00:00:14:000 bullet 1

00:00:19:000 bullet 1

00:00:21:000 bullet 1

00:00:25:000 bullet 1

00:00:29:000 bullet 1

00:00:29:000 bullet 1

00:00:36:000 bullet 1

00:00:41:000 bullet 1

00:00:48:000 bullet 1

00:00:50:000 bullet 1

00:00:57:000 bullet 1

As shown, the outputs are apart from each other with a random time interval. Therefore this block behaves as desired.

## Health

//Health.ev

00:00:08:00 bullet 1

00:00:15:00 bullet 1

00:00:17:00 bullet 1

00:00:20:00 bullet 1

00:00:23:00 bullet 1

00:00:25:00 bullet 1

00:00:40:00 bullet 1

00:00:42:00 bullet 1

00:00:48:00 bullet 1

00:00:50:00 bullet 1

00:00:58:00 bullet 1

//HealthOUT.out

00:00:50:000 dead 1

The health of zombie is set to 10. As seen the output is generated at 00:00:50:000 which is upon receiving the 10th bullet. Also, there is no output generated again at 00:00:58:000. These behaviours are desired which proves this block functions correctly.

## Speed

//Speed.ev

00:00:20:00 enct 1

00:00:30:00 ate 1

00:00:38:00 enct 1

00:00:40:00 ate 1

00:00:45:00 enct 1

00:00:50:00 dead 1

00:00:50:00 ate 1

//SpeedOUT.out

00:00:05:000 fwd 1

00:00:07:000 fwd 1

00:00:09:000 fwd 1

00:00:11:000 fwd 1

00:00:13:000 fwd 1

00:00:15:000 fwd 1

00:00:17:000 fwd 1

00:00:19:000 fwd 1

00:00:30:000 fwd 1

00:00:32:000 fwd 1

00:00:34:000 fwd 1

00:00:36:000 fwd 1

00:00:40:000 fwd 1

00:00:42:000 fwd 1

00:00:44:000 fwd 1

00:00:50:000 fwd 1

00:00:52:000 fwd 1

00:00:54:000 fwd 1

00:00:56:000 fwd 1

00:00:58:000 fwd 1

00:01:00:000 fwd 1

The Moving speed of the zombie is set to 00:00:02:000 apart, and the initialization time is set to 00:00:05:000. Therefore the first “fwd” output appears at that time, and keeps outputting at a constant rate. It is then stopped outputting after 00:00:19:000 because there is an “enct” input received at 00:00:20:000. It then restarts to generate outputs at 00:00:30:000 upon receiving an “ate” signal indicating the plant has been eaten and the zombie is able to move again. These behaviours are as expected.

However, at 00:00:50:000, two inputs are fed into the model. One will trigger the model to restart generation of outputs, and one will not. In the code implementation, it was designed to handle this kind of situation such that the model should NOT generate output. It is however failed in this test. The reason is nonetheless obvious. Computer is a sequential machine which will not have simultaneous inputs. Whichever input is given the last will be the dominating one. Thus here the “ate” signal is given the last and that triggered the regeneration. This result is actually expected as explained in class that there cannot really be simultaneous events at the same time. Therefore the overall performance of the model is still successful.

\*Note: simultaneous inputs will NOT be tested in the later models.

## Bite

//Bite.ev

00:00:08:00 enct 1

00:00:15:00 ate 1

00:00:20:00 enct 1

00:00:30:00 ate 1

00:00:32:00 ate 1

00:00:33:00 enct 1

00:00:38:00 enct 1

00:00:45:00 ate 1

//BiteOUT.out

00:00:08:000 bit 1

00:00:10:000 bit 1

00:00:12:000 bit 1

00:00:14:000 bit 1

00:00:20:000 bit 1

00:00:22:000 bit 1

00:00:24:000 bit 1

00:00:26:000 bit 1

00:00:28:000 bit 1

00:00:33:000 bit 1

00:00:35:000 bit 1

00:00:37:000 bit 1

00:00:38:000 bit 1

00:00:40:000 bit 1

00:00:42:000 bit 1

00:00:44:000 bit 1

The bite rate is set to 00:00:02:000 apart from each other. As seen upon receiving an “enct” input at 00:00:08:000, the model starts to generate output at that constant rate. It stops generating outputs after 00:00:14:000 because the “ate” signal is received at 00:00:15:00. For later in time, the results show that duplicated “enct” and “ate” inputs does not change the behaviour of the model. Therefore, this model functions correctly.

## SingleZombie

In this coupled model, three previous atomic models are put together to examine their functions together.

//SingleZombie.ev

00:00:30:00 enct 1

00:00:40:00 ate 1

00:00:50:00 bulletOut 1

00:00:52:00 bulletOut 1

00:00:54:00 bulletOut 1

00:00:56:00 bulletOut 1

00:00:58:00 bulletOut 1

00:01:00:00 bulletOut 1

00:01:02:00 bulletOut 1

00:01:04:00 bulletOut 1

00:01:06:00 bulletOut 1

00:01:08:00 bulletOut 1

00:01:10:00 bulletOut 1

00:01:14:00 ate 1

00:01:20:00 enct 1

00:01:34:00 ate 1

//SingleZombieOUT.out

00:00:05:000 fwd 1

00:00:07:000 fwd 1

00:00:09:000 fwd 1

00:00:11:000 fwd 1

00:00:13:000 fwd 1

00:00:15:000 fwd 1

00:00:17:000 fwd 1

00:00:19:000 fwd 1

00:00:21:000 fwd 1

00:00:23:000 fwd 1

00:00:25:000 fwd 1

00:00:27:000 fwd 1

00:00:29:000 fwd 1

00:00:30:000 bit 1

00:00:32:000 bit 1

00:00:34:000 bit 1

00:00:36:000 bit 1

00:00:38:000 bit 1

00:00:40:000 fwd 1

00:00:42:000 fwd 1

00:00:44:000 fwd 1

00:00:46:000 fwd 1

00:00:48:000 fwd 1

00:00:50:000 fwd 1

00:00:52:000 fwd 1

00:00:54:000 fwd 1

00:00:56:000 fwd 1

00:00:58:000 fwd 1

00:01:00:000 fwd 1

00:01:02:000 fwd 1

00:01:04:000 fwd 1

00:01:06:000 fwd 1

00:01:08:000 dead 1

00:01:14:000 fwd 1

00:01:16:000 fwd 1

00:01:18:000 fwd 1

00:01:20:000 bit 1

00:01:22:000 bit 1

00:01:24:000 bit 1

00:01:26:000 bit 1

00:01:28:000 bit 1

00:01:30:000 bit 1

00:01:32:000 bit 1

00:01:34:000 fwd 1

00:01:36:000 fwd 1

00:01:38:000 fwd 1

00:01:40:000 fwd 1

00:01:42:000 fwd 1

00:01:44:000 fwd 1

00:01:46:000 fwd 1

00:01:48:000 fwd 1

00:01:50:000 fwd 1

00:01:52:000 fwd 1

00:01:54:000 fwd 1

00:01:56:000 fwd 1

00:01:58:000 fwd 1

00:02:00:000 fwd 1

As shown, this model starts generating “fwd” output at 00:00:05:000 after some initiation time set in the Speed model. No input is needed to trigger this event to happen. At 00:00:30:00, the “enct” signal is received. Two events happened after receiving this input. First is that SignleZombie model stops generating “fwd” output, and second it starts generating the “bit” outputs. This signifies the zombie has come next to a plant, so it can no longer move forward but instead start biting the plant.

At 00:00:40:000, the “ate” input is received signifying the zombie has finished eating the plant and thus nothing stops the zombie to move forward any more. Thus, the SingleZombie model restarts generating the “fwd” outputs but no longer generating “bit” outputs.

While the zombie keeps moving forward, starting at 00:00:50:000, the zombie is under attack from the plant. After been hit 10 times, at 00:01:08:000, the zombie is dead and SingleZombie model generates a “dead” output. Again, this model does not generate duplicated “dead” output at 00:01:10:000 upon receiving another bullet because the zombie has already dead.

Although the zombie is dead by now, this model restarts generating “fwd” and “bit” outputs again after receiving “enct” and “ate” inputs afterwards. This behaviour is correct here because the Yard model should prevent sending these inputs after the death of zombie. Therefore, the coupled model SingleZombie is proven to be working as well.

## Yard

To test this model, several separate event files are supplied to the same model to study its behaviour in different situations. The first event file focuses on testing if the model can arrive at success state; second one focuses on the model’s behaviour upon receiving a death input from the SingleZombie model; third one focuses on testing the model’s behaviour upon receiving different combination of inputs.

### Case I:

//Yard.ev

00:00:05:00 fwd 1

00:00:07:00 fwd 1

00:00:09:00 fwd 1

00:00:11:00 fwd 1

00:00:13:00 fwd 1

00:00:15:00 fwd 1

00:00:17:00 fwd 1

00:00:19:00 fwd 1

00:00:21:00 fwd 1

00:00:23:00 fwd 1

00:00:25:00 fwd 1

00:00:27:00 fwd 1

00:00:30:00 fwd 1

//YardOUT.out

00:00:17:000 enct 1

00:00:19:000 win 1

The zombie is generated at (8, 1) on the yard. Shown above, the model generates an “enct” output at 00:00:17:000 because at such time there were 7 “fwd” signals been received and the zombie has arrived at (1, 1). By the definition of the project, once the zombie reaches (1, \*) the zombie is next to the plant. Thus an “enct” signal is outputted. At this moment, the zombie should stop moving forward and start biting the plant. However those mechanisms are dealt in the SingleZombie model explained above. Here the zombie is forced to move forward again. Thus at 00:00:19:000 the zombie reaches (0, 1) which in the game means the zombie obtained the brain, and thus succeeded. Only one “win = 1” output is generated which is what desired.

### Case II:

//Yard1.ev

00:00:09:00 fwd 1

00:00:11:00 dead 1

00:00:13:00 fwd 1

00:00:15:00 fwd 1

00:00:17:00 fwd 1

00:00:19:00 fwd 1

00:00:21:00 fwd 1

00:00:23:00 fwd 1

00:00:25:00 fwd 1

00:00:27:00 fwd 1

00:00:30:00 fwd 1

00:00:32:00 fwd 1

00:00:33:00 fwd 1

00:00:35:00 bit 1

00:00:37:00 bit 1

00:00:38:00 bit 1

00:00:40:00 bit 1

00:00:42:00 bit 1

00:00:45:00 bit 1

00:00:46:00 bit 1

//YardOUT\_1.out

00:00:11:000 win 0

This time, the zombie is generated at (5, 3) on the yard. After moving forward once, the zombie is dead at 00:00:11:000. An output of “win = 0” is generated signifying the intrusion is failed. After that, no matter how many “fwd” or “bit” inputs are received, the model does not generate any output because the zombie is dead.

### Case III:

//Yard2.ev

00:00:05:00 fwd 1

00:00:07:00 bit 1

00:00:09:00 bit 1

00:00:11:00 bit 1

00:00:13:00 bit 1

00:00:15:00 bit 1

00:00:17:00 bit 1

00:00:19:00 bit 1

00:00:21:00 dead 1

00:00:23:00 fwd 1

00:00:25:00 fwd 1

00:00:27:00 fwd 1

00:00:30:00 fwd 1

00:00:32:00 fwd 1

00:00:34:00 fwd 1

00:00:36:00 fwd 1

00:00:38:00 fwd 1

00:00:40:00 fwd 1

00:00:42:00 fwd 1

00:00:46:00 fwd 1

//YardOUT\_2.out

00:00:15:000 ate 1

00:00:21:000 win 0

This time, the zombie is generated at (5, 2) on the yard. As shown this model generates an “ate” output at 00:00:15:000 after the plant has been bitten for 5 times and thus eaten. Then the zombie is dead right after at 00:00:21:000, so an output of “win = 0” is generated. Afterward, no matter how many “fwd” inputs are received, there is no “enct” nor “win = 1” signal is generated.

To conclude from three test cases for this Yard model, this model has been implemented successfully.

## Plants\_vs\_Zombie

This is the overall top model for the project. This coupled model does not require inputs, thus no event file is supplied. A few runs of simulation were performed to see if the model can generate the outputs correctly.

Run I: Zombie at (1, 3) 00:00:05:000 succ 1

Run II: Zombie at (1, 1) 00:00:05:000 succ 1

Run III: Zombie at (8, 1) 00:00:25:000 succ 1

At this point, the zombie is able to survive even when it is deployed at (8, 1). Thus the zombie is too powerful for the simulation to generate a “succ = 0”. Thus, the moving speed of the zombie has been modified from 00:00:02:000 to 00:00:05:000. Also, the health of the zombie is reduced from 10 to 5. After these modifications, the following intrusion failure is obtained:

Run VI: Zombie at (9, 2) 00:00:13:000 succ 0

# Conclusion:

The overall design and implementation of this project has been successfully achieved. There are many improvements and expansions can be added to this project. For example, the model for plants has not been implemented; the model for the Yard can be largely improved to support multiple zombies and plants. However, under the constrain of project deadline and project complexity, This project has been implemented successfully.