Question 3: Ant Colony

# Structure of the program

I have implemented the Ant Colony by creating two loops: *ant\_loop* that handles messages received by one ant process and *colony\_loop* that handles the colony process. Inside the *ant\_loop* I kept the current coordinates of the and as a tuple *{X,Y}* , one Boolean that says if pen is up or down, the value of the current angle, the whole picture that was drawn by the ant while having the pen up and the process id of the colony process to which it belongs. Inside the colony\_loop process I have chosen to keep a list with coordinates of the living ants, a counter for the number of ants that died and nother list containing the pictures of the ants that died.

# Implementation for the interaction with ants processes

I have made that inside of the functions *forward*, *left*, *right* and *setpen* to send asynchronous messages to the process as described in the exam, even if it has proven to have some race condition problems as stated. I could also implement by making *rpc* calls to the ant process and also return *{ok}* in case of wrong values passed to the ant but I have chosen to follow the requirements of the exam. Since it was not specified I have chosen to set the value false for the boolean pen that is set to the ant. So the pen has to be set up in order to have the ant draw something when moved.

In case it happens that an ant to die because of wrong values passed to functions *forward*, *left*, *right* or *setpen* I have chosen to simply send an asynchronous message to the colony that holds the ant and simply no longer call the loop of that function when the handling of the message ends. Like this, the process exits the loop and stops. I know it was better to use a separate process to handle the ant and colony processes that die or get blasted but I did not have much time to make such a complex implementations. And however it has proven that even this simple way to solve this point is effective and correct. If I use the function is\_process\_alive(Pid) before and after the ant process has to stop, I get true before and false after I stop the process by doing as I described.

In case of using the clone function for an ant process it starts a recursive function called *clones* that sends a message to the colony process with information needed by it to create a new ant process: coordinates and angle so the clones have these values similar to these. For this purpose I have also created a function *new\_ant\_clone(C,{X,Y},Angle)* to create a new ant process with these values set.

# Implementation for the interaction with colony processes

For the interaction with the colony I have chosen to only keep a list with process ids of the ants that are alive and in case that one ant has to die, the colony concatenates its picture to the list *Graveyard* which is also returned in case of using the function that requests the picture drawn by the dead ants. I also have a counter *NrDeadAnts* which is incremented each time a message *{From, {ant\_died, DeadPicture}}* reaches the colony process.

For the implementation of *blast* function I have chosen that the colony process that has to be stopped is at first calling a recursive function *blastAllAnts* which takes the list of ids that the colony has and asks each ant process to stop in the same manner as in case of ants that die because of errors. After this function ends, then the current colony process is also stopped by simply not calling the *colony\_loop* function again.

Since I have the process Ids, the number of dead ants and graveyard already memorized by the colony process then the functions requesting these where easy to implement by just returning the information that is asked.

For the implementation of function *picture* I have chosen to also call a recursive function that requests the *Picture* from each ant with process id stored in *LivingAnts*.

# Testing

I have done several tests to check the functions are working properly by using Unit testing. In order to avoid the issues caused by race conditions when calling the asynchronous function one after another I have used the function timer:sleep(1) which makes the function to sleep for 1 millisecond which I have noticed it is enough for that the previous operation is finished by the ant process.

* The first two tests are just checking if the functions for starting a new colony process and a new ant process are working properly
* The next 3 tests are checking that by spawning multiple ants, their corresponding colony process is returning the correct number of ants. In 2 of these tests I have made that some of the ants to die by sending wrong parameters to each of the functions *forward*, *left*, *right* and *setpen* and also checked if the number of dead ants is returned correctly
* I have done one test where I have moved an ant and also rotated it in order to check if the final position is the one that was expected.
* One test that checks the functionality of cloning that starts two processes assigned to same colony and makes multiple spawns for them. At the end it is checked if the total number of ants in the colony is the one that was expected.
* Two tests which prove that colony and processes are stopped properly when killing or blasting them by checking the result offered by *is\_process\_alive()* function before and after they are killed.
* 2 tests which check if there is data saved into Picture of an ant if the pen is set up or down. I wanted to make sure that the function setpen works properly.

Inside of the file *svg.erl* I have also created a function that takes the code returned by the *pictureToSvg* function and writes it into a file so it was easier for me to visualize the evolution of ants.

# Conclusion

I have chosen to solve the challenges of this question in a simple and quite effective manner. Perhaps it is not entirely correct but the functionality of the program is working very well even like this. I have done tests to prove that functions are working properly and that the framework is quite robust in case of that some processes have to stop. Perhaps more error handling for wrong parameters or unexpected failure of a process could have been a good improvement to my implementation. If I had time I would have created another loop process that is controlling the servers for ants and colony.