Meilenstein Omega

Übersicht:

- Simulator / Ant :

Im Simulator werden alle Threads gestartet. Jeder Thread ruft die run() Methode in Ant Class auf und dann fängt die
Ameise ihre Suche nach Futter mithilfe der forwardMoving() Methode.
Die run() Methode ist zuständig für den Ameisenverlauf und die Terminationsbedingungen.

- AntRunhandler:

In dieser Klasse haben wir ein Teil des forwardMoving gesteuert, die meisten Recorder Signals werden dort geschickt.

- HomewardPathCheck / FoodSearchPathCheck+FoodSearchTrailHandler:

Sind zuständig für die Suche nach dem besten Weg.

- TrailEntry/ ClearingEntry + ClearingEntryhandler:

Diese Klassen beschreiben das Verhalten der Ameise beim Eintreten, Abholen/Droppen der Futter.

- Trail/ Clearing:

enterTrail() / enterClearing()
sind Methoden in Trail/Clearing Klasse, die die passenden Methoden vom ClearingEntry oder TrailEntry aufrufen .

Design Entscheidung:

- Die Idee unseres Designs ist, dass jedes **Trail/Clearing** sein eigenes **ClearingEntry/ TrailEntry** hat, wobei diese als ***Gate*** betrachtet wird .
Eine Ameise kann ein Trail/Clearing nur durch Methoden vom ClearingEntry/TrailEntry betreten.
Auf diese Art und Weise muss eine **Ameise(a),** die ein **Trail (A)** betreten will , ***nicht*** mit einer **Ameise(b)** , die ein **Trail (B)**
betreten will, auf **Locks** ***konkurrieren***.
Das passiert jetzt nur wenn beide Ameisen (a) und
(b) dasselbe Trail (A) betreten wollen. Das gleiche gilt für das Betreten eines Clearings.

- wir haben uns für ***explizite Locks*** und ***conditions*** entscheiden.

jeder Trail/Clearing hat sein eigens lock(für pickupFood/DropFood oder Pheromons update) und sein eigenes Gate (ClearingEntry/TrailEntry) die auch Ihre eigene locks und condition „isSpaceLeft“ .

- wenn eine Ameise ein Trail verlässt, sendet sie ***signalAll*** zu allen anderen Ameisen, die dieses Trail betreten wollen.
Das gleiche gilt für das Betreten eines Clearings.

- Alle Locks bei uns sind mit ***Try/catch/finally*** geschützt. Das heißt ,dass alle locks werden garantiert zurück gegeben.

Nebenläufiges Verhalten:

mehrere Locks / Deadlocks und Data Races:

Die kretischen Stellen in das Projekt ,wo DataRaces/Deadlocks vorkommen können, sind (Das ***Betreten eines Clearing/Trail*** , ***PickUpFood***, ***DropFood***, Das ***Updaten der Pheromone***).
Folglich wird jeder Fall alleine detailliert betrachtet.

- ClearingEntry:

wenn eine ***Ameise (a)*** ein ***(Clearing A)*** betreten möchte, nimmt diese Ameise das ***lock für dieses Clearing*** und danach nimmt sie auch das ***Lock für den Trail***, wo sie sich gerade befindet .
((Dass die Ameise jetzt **2 locks** hält, führt **nicht** zum ***DeadLock*** , Weil solange die Ameise das Clearing nicht betreten kann, ist sie alleine auf dem Trail.
Das heißt die anderen Ameisen , die auf das Lock des Trails warten, bekommen sowieso kein Zugriff darauf, bis die Ameise das Trail verlässt.
Außerdem wenn die Ameise noch **kein Platz** im Clearing findet , wird sie im **waitSet** warten und das Lock des Clearing zurückgeben. Und wenn sie länger als Ihre **Tarnfähigkeit** im waitSet wartet, wird sie **sterben**. In diesem Fall wird auch das **lock des Trails zurückgegeben** mittels unlock() im finally Block.**))**
Nachdem sie das Clearing betreten hat, verlässt sie der Trail, informiert andere Ameisen durch ***signalAll***, dass der Trail wieder ***frei*** ist und gibt **das Lock des Trails wieder zurück** .
Danach fügt Clearing/Trail zur Sequenz und **gibt danach das Lock für das Clearing zurück**.

Dadurch werden hier ***keine DataRaces oder Deadlocks*** vorkommen.

- TrailEntry:

Im Prinzip ist das Betreten eines Trails **ähnlich** zum Betreten von Clearings. Das einzige **Unterschied** hier ist , dass die Ameise das ***Lock des Clearing*** wo sie sich gerade befindet erst dann **beansprucht, nachdem** sie das l**ock des Trails**, wo sie hin will, **bekommen** hat, und ein **freien Platz** gefunden hat, damit sie das Lock des Clearings nicht umsonst hält und die Anderen Ameisen ,die das Clearing betreten wollen, nicht aufhält.
Nachdem sie ein Platz im Trail gefunden hat, ***nimmt sie die 2 Locks,*** betritt das Trail, verlässt das Clearing und gibt **signalAll** . danach **gibt sie das Lock des Clearings zurück ,**sie updatet die Pheromone und **gibt das Lock des Trails zurück**.
Die **locks** werden also **nicht ewig** genommen . Und wenn sie terminieren muss oder irgendwas unerwartet passiert, werden die locks Dank ***finally Block*** auf jeden Fall zurückgegeben

Hier werden auch ***keine DataRaces oder Deadlocks*** vorkommen.

- getOrSetFood:

Diese Methode packt „hasFood“, „pickupFood“ und „placeFood“ zusammen.
Sie ist **durch Locks geschützt.**
Das bedeutet ,dass auf die Variable „Food“ nicht mehr als eine Ameise gleichzeitig zugreifen kann.

DataRaces sind dadurch vermieden. ***DeadLock*** kann auch nicht passieren

- getOrSetHill/ FoodPheromone:

Durch diese Methoden kann eine Ameise den ***Food- und HillPheromone entweder lesen***
oder schreiben.
Sie ist **durch Locks geschützt.**
Die Locks versichern uns, dass nur eine Ameise gleichzeitig auf die „Pheromone“ zugreifen kann.

DataRaces sind dadurch vermieden. ***DeadLock*** kann auch nicht passieren

Terminierung:

1)

Alle Methoden ,die für das betreten eines Clearing/Trail, das updaten eines Pheromons oder das aufheben/absetzen von Futter zuständig sind, geben ein **„boolean wert**“ zurück. Nun bevor die Ameise einen von diesen Schritten macht, überprüft sie schnell Ihre **Interrupt Status** und , ob ***genüg Futter aufgesammelt*** wurde . Wenn einer der beiden Fälle auftritt, wird sie die **locks zurückgeben , SignalAll verteilen** und **„False“ zurückgeben**, Welches als nicht erfolgreiche Operation in der „forwardMoving()“ Methode verstanden wird. Dort wird dann die Suche beendet und **ein Exception geworfen** , der am Ende der „Ant.run()“ **gefangen wird**. Danach ***terminiert*** die Ameise.

2)

Auch wenn eine Ameise merkt ,dass sie **Interrupted wurde**, während sie im **WaitSet** wartet, dann wird die **geworfene Exception** durch ein **catch** gefangen. Dann wird die Ameise **hinter sich Aufräumen**(Clearing/Trail verlassen usw.) und ein „**return False**“ zurückgeben , welches dann weiter wie bei punkt 1) behandelt wird.

3)

Eine Ameise ist im **Ameisenhaufen und alle Trails sind mit MaP** beschriftet. Diesen Fall wird einmal in „Ant.forwardMoving()“ Überprüft und ein **Exception** zum **Terminieren** wird dann geworfen ,der ebenfalls in run() **gefangen** wird. Danach ***terminiert*** die Ameise

4)

wenn die Ameise ***entdeckt und gefressen*** wurde, weil sie auf einen Trail mehr Zeit als ihre Tarnfähigkeit verbracht hat, werden die **Locks zurückgegeben , SignalAll verteilt** und ein **„False**“ zurückgeben .

Dann wird auch ein **Exception** geworfen und in run() **gefangen**.

Danach ***terminiert*** die Ameise.

5)

Wenn eine Ameise die ***letzte FutterEinheit*** zum Ameisenhaufen gebracht hat, terminiert sie glücklich in „Ant.run()“ .

Der ***Simulator*** wartet bis alle Ameisen terminiert haben, dann beendet er die Simulation und terminiert.

Falls er aber beim warten Interrupted wurde , dann beendet er die Simulation und terminiert sofort, wie verlangt. Die Ameisen die zu diesem Punkt noch nicht terminiert haben , merken dann beim nächsten Schritt ,dass sie vom Simulator in der „main“ Methode interrupted sind, und terminieren wie oben erklärt.