change thickness here

Uppsala University

Introduktion till informationsteknologi

A Bug's Life

Authors:
Anonymous Student Team
Aletta Nylén

1 oktober 2023

Innehåll

1 Inledning

Swarm intelligence är ett koncept som innebär att en grupp av organismer, helt utan en ledare i centrum och endast med begränsad individuell intelligens, i grupp kan lösa komplexa problem. Till exempel består ett försvar av soldater. Varje soldat kan inte göra så mycket själv, men som grupp är soldaterna mycket starkare. Något annat man kan göra är att simulera idrottslag. Till exempel kan man simulera hur fotbollsspelare borde röra sig på en spelplan för att hjälpa varandra och få ut mer av sitt spel. Med avancerade varianter av swarm intelligence kan man räkna ut hur man optimalt kan röra sig för att få så stort övertag som möjligt. Genom att simulera enklare varianter av swarm intelligence kan man dra lärdomar som senare kan användas för att simulera mer komplicerade varianter.

BugsLife är en simulering där myror kan hitta utplacerad mat och sedan ta tillbaka den till sitt bo. Genom att lämna feromoner på vägen kan myrorna sedan hitta tillbaka till maten genom att följa sina egna spår. Systemet är konstruerat så att det är fullständigt concurrent och arbetar helt utan någon supervisor eller tidssynkronisering. Detta har åstadkommits genom att låta varje objekt i simuleringen vara en egen process där all kommunikation mellan processerna sker via meddelandeöverföring. Dessutom använder systemet sig av actor-modellen och är fritt från deadlocks.

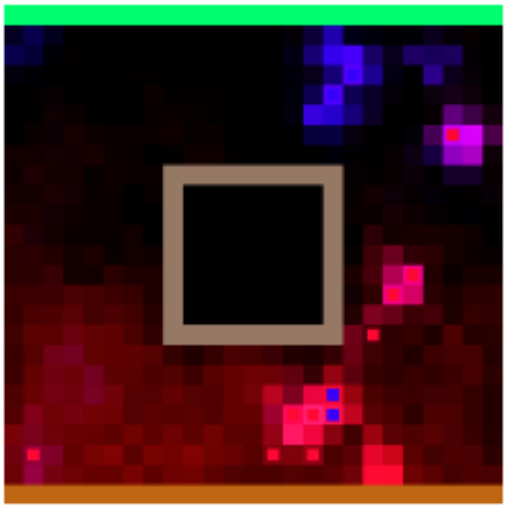
2 Relaterat arbete

Probelmet med den resande säljaren (traveling salesman problem, TSP) handlar om att hitta korsatste vägen mellan ett antal städer i en graf där städerna är noder och kopplingar mellan städerna är kanter. Problemet kan lösas med hjälp av artificiella myror [1]. Man börjar med att placera ut myror i slumpmässigt utvalda städer. Sedan får myrorna utforska grafen genom att gå och släppa ferormoner längs kanterna. Då myrorna undersöker mer och mer av grafen läggs med tiden fler och fler feromoner längs kanterna. Efter ett tag börjar en bra lösning komma fram, inte alltid den optimala men ofta.

3 Översikt över systemet

Användaren startar simuleringen utan input-parametrar och systemet körs sedan automatiskt och kräver ingen interaktion från användaren. Vid första

anblick kan myrkolonin uppfattas som enkel, men komplexiteten i simuleringen ligger i systemarkitekturen och de algoritmer myrorna använder sig av för att effektivisera hämtningen av mat.



Figur 1: En skärmdump av simuleringen. De brun/orangea rutorna är myrbo, de gröna är mat, de bruna/gråa är väggar. De små röda rutorna är myror som letar efter mat och de små blåa rutorna är myror som är på väg hem med mat. I bakgrunden syns feromonerna.

3.1 Concurrency

På grund av den höga nivån av autonomitet kombinerat med separationen mellan de olika processerna används så kallad eventdriven programmering. Eventdriven programmering är när ett program styrs av externa händelser som inte går att kontrollera eller förutse. Ett exempel på det är vid programmering av användargränssnitt, då det inte går att förutspå när eller i vilken ordning användaren kommer att ge input till systemet.

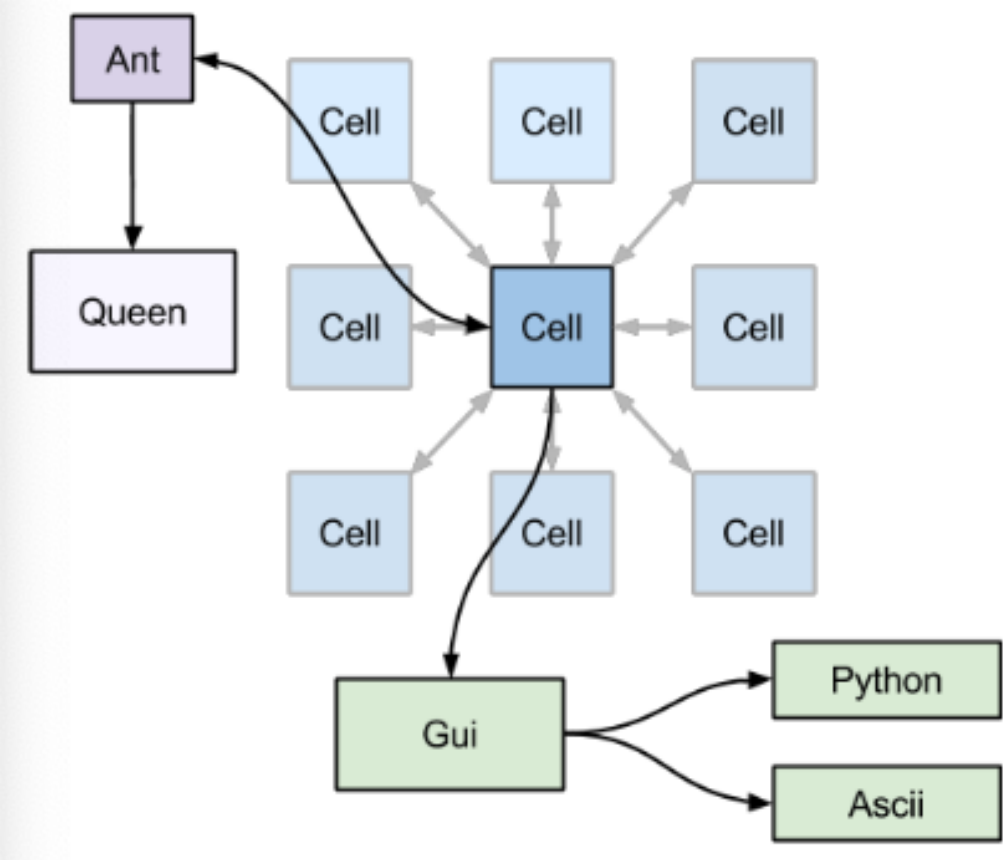
Systemet använder sig av actor-modellen då den ger bra möjligheter att på ett konceptuellt och effektivt sätt implementera eventdriven programmering då den ger en möjlighet att fokusera på en modul i taget utan att behöva ta hänsyn till hur de andra modulerna är implementerade. Det enda som behövs är ett meddelandeprotokoll som alla moduler följer.

Det är viktigt att påpeka att systemet använder sig av asynkron event-

driven programmering. Detta ger upphov till många problem då det inte går att garantera att meddelandena kommer i rätt ordning. Problemen löses med hjälp av meddelandefiltrering, som innebär att en process väntar på ett specifikt meddelande innan den går vidare och buffrar de meddelanden som kvarstår. Varje process behöver då ha ett speciellt tillstånd när den accepterar alla nya förfrågningar.

3.2 Systemdesign

Systemet är uppbygt i moduler där alla moduler arbetar helt asynkront och all kommunikation mellan modulerna sker via meddelanden.



Figur 2: En schematisk överblick över systemarkitekturen

3.2.1 Ant

Ant-modulen är den modul som implementerar myran. Ant-modulen innehåller de funktioner som krävs för att myran skall kunna fatta beslut över hur den ska agera baserat på hur dess omgivning ser ut. Detta gör den genom att skicka förfrågningar till den cell myran befinner sig på och sedan använda svaret för att autonomt fatta ett beslut om vilken handling som är lämpligast. Under normal drift så tar myran aldrig emot några förfrågningar från någon annan process, endast svar från den cell som den befinner sig i.

Ant-modulen är utrustad med funktionalitet så att myran kan analysera den information som ges av grannskapet till cellen den står i. Informationen gör det möjligt för myran att fatta ett beslut, baserat på vad myrans omgivning består av. Myror i det verkliga livet kommunicerar med varandra genom olika typer av feromoner, detta för att meddela var maten finns. Det är viktigt att myrorna kan inspektera sin omgivning så att den kan fatta ett beslut baserat på vad andra myror har upptäckt i omgivningen. Finns det feromoner som indikerar till myran att det finns föda i närheten kan myran ta ett beslut att röra sig i den riktningen och leta rätt på födan. Myrorna använder sig även av feromoner för att hitta hem till sitt näste.

3.2.2 Cell

Cell-modulen representerar cellerna där varje cell är en ruta i det rutnät som utgör världen där myrorna lever. Cellen är det objekt som innehåller den relevanta informationen för systemet. En cell innehåller information om vad det är för typ av cell, om cellen exempelvis representerar ett myrbo, om det finns mat på cellen, om det står en myra på cellen samt intensiteten för dem olika feromonerna på cellen. Förutom att lagra information hanterar cellen också meddelanden. Cellen tar emot och besvarar förfrågningar från myran som står på cellen, tar emot, besvara och skickar förfrågningar till cellerna i dess direkta grannskap samt skickar meddelanden till GUI-modulen. Den enda gång en cell kan skicka förfrågningar till en annan cell är om cellen fått en förfrågan att skicka en förfrågan.

3.2.3 Message_Buffer

Message Buffer är den modul som hanterar kommunikationen av meddelanden mellan processerna. Modulen exporterar receiver-funktionen som är central för att använda den typ av eventdriven programmering som används.

Receiver-funktionen används för att hantera deadlocks och filtrering av meddelandekön.

3.2.4 GUI

GUI-modulen är implementerad i Python och Erlang med hjälp av ErlPort. Modulen tar emot och behandlar meddelanden innehållandes cellernas tillstånd från cellerna. Informationen omvandlas till Pythons syntax via Erlport och skickas till en Python-instans som hanterar den grafiska representationen.

3.2.5 Grid_Init

Modulen används för att bygga upp världen samt placera alla celler och länka ihop dessa. Grid Init modulen tilldelar även alla celler dess initiala attributer samt skapar, placerar och startar alla myrorna. Grid init initierar även en Queen-process. Myrorna skickar statistik till Queen-processen, som används för debugging och prestandamätning.

4 Implementation

Systemet implementerades i Erlang då det är ett väletablerat och stabilt språk som är byggt kring actor-modellen och meddelandeöverföring. Utöver Erlang används Python för den grafiska delen. För att kunna kommunicera mellan de två språken används ErlPort [2].

4.1 GUI

GUI-modulen initieras med att skapa en lista motsvarande hela griden som fylls med tomma atomer. GUI-modulen körs sedan i en main loop som tar emot meddelanden från cell-modulerna, bearbetar dessa och fyller på den listan. När GUI-modulen tar emot ett meddelande från en cell används en algoritm för att uppdatera listan. Listan skickas via ErlPort till Python där den ritas upp grafiskt.

För den grafiska renderingen i projektet används Pythonbiblioteket Pygame. Pygame är ett bibliotek av färdigutvecklade moduler till Python designade för att enkelt kunna skapa spel med en enkel grafisk implementation. Pygames design gör att det är enkelt att använda på alla plattformar som har stöd för Python.

4.2 Ant-modulen och Cell-modulen

Ant-modulen och Cell-modulen är väldigt lika varandra i sin implementation. Båda modulerna består av en initialiseringsfunktion som används för att starta processer. Funktionen inväntar de meddelanden som är nödvändiga för att starta processerna. Detta kan till exempel vara ett meddelande om att myran har placerats korrekt och att cellerna har fått sitt grannskap definierat.

Cellerna och myrorna går efter ett meddelande till sin mainfunktion där de inväntar nya förfrågningar. Om det inte finns några obehandlade meddelanden kommer myrorna att agera spontant. När cellerna och myrorna får ett inkommande meddelande kommer de att anropa en funktion specifikt för den typen av meddelande. Dessa funktioner kan skicka och ta emot meddelanden själva med hjälp av meddelandefiltreringen. När en förfrågan eller annat meddelande har behandlats kommer processen återgå till sin mainfunktion. Om ett felaktigt eller otillåtet meddelande inkommer under någon del av exekveringen så kommer hela systemet att krascha.

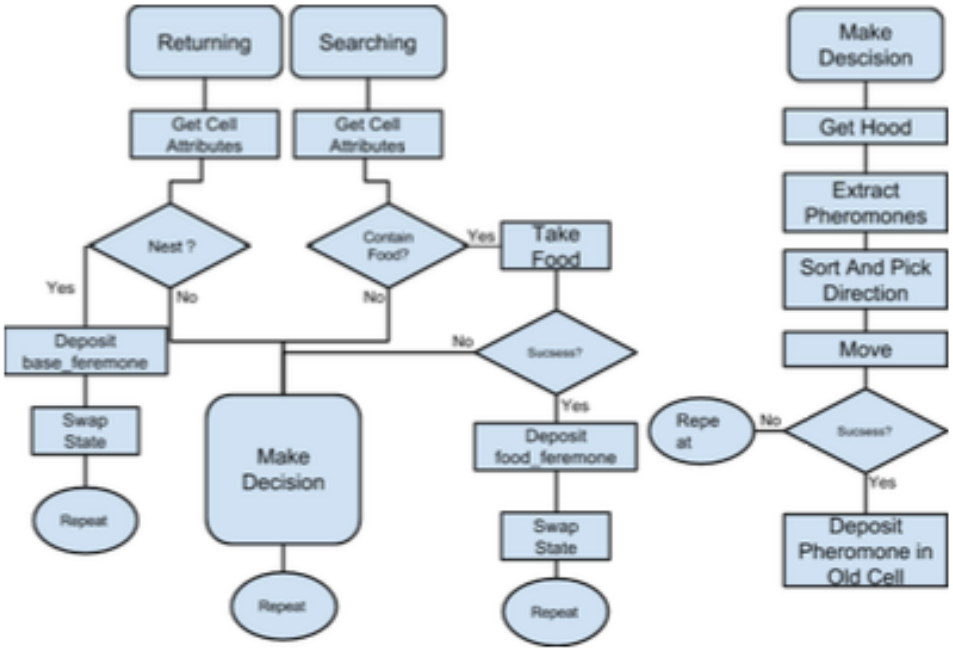
Cellerna kommer vid varje inkommen förfrågan, beroende av cellens tillstånd, automatiskt genomföra en uppdatering av dess feromonnivåer. Detta sker som en funktion av den faktiska tiden (wall-time) som har gått sedan cellen senast uppdaterades.

4.2.1 Myrans algoritm

Myrans beslut baseras på en väldigt enkel algoritm. Myran kan vara i två olika tillstånd `searching_for_food` och `returning_with_food`. När myran letar efter mat, har tillståndet `searching_for_food`, så kommer den att undersöka cellen den står i, om det finns mat i cellen så kommer myran att försöka plocka upp maten. Om myran lyckas plocka upp mat kommer den att byta tillstånd till `returning_with_food`, om myran misslyckas med att plocka upp mat så kommer den att fortsätta leta efter mat i andra celler.

När myran letar efter mat så kommer den att be cellen den står i att skicka tillbaka information om alla celler i dess grannskap. Myran kommer sedan att studera sitt grannskap och sortera cellerna efter hur mycket `food_feremone` varje cell innehåller och sedan genom *rank-selection* att välja den riktingen den skall gå i. Rank-selection innebär att den med en förutbestämd sannolikhet p kommer att gå till den cellen med det högsta antalet feromoner. Om den inte väljer den riktningen så kommer den att gå till den cellen med näst högst riktning med samma sannolikhet p.

Myran kommer efter varje genomförd förflyttning att släppa feromoner på den cellen där den tidigare var. Då myran letar efter mat kommer den att släppa `base_feremone` och då den går tillbaka med maten så kommer den att släppa `food_feremone`. Då myran går tillbaka med mat så kommer den att följa en snarlik algoritm men istället leta efter den högsta koncentrationen av `base_feremone`.



Figur 3: Flowchart över myrans algoritm

4.2.2 Grid_Init

Grid init är den modul som bygger upp världen. Det första som den modulen gör är att den startar alla cell-processer. Det krävs att GUI-modulen är startad, dess PID kommer att skickas med till alla celler. Den kommer sedan att skicka ut `set_cell_attribute` medelanden till alla dessa celler med cellernas attribut. Grid_init kommer sedan att initiera en *Queen*-process, starta alla myror och försöka placera ut dem på de celler där de skall starta. När alla myror har blivit utplacerade så kommer Grid_init modulen att skicka `start_ant` medelanden till myrorna vilket kommer att starta simuleringen.

4.3 Concurrency

Då concurrencyn i systemet uteslutande bygger på actor-modellen och message passing så har systemet tre definierade klasser av meddelanden.

- Enkelrikatade meddelanden på formen
  `{Pid,{Type,Payload}}` eller `{Pid,Type}`
  Enkelriktade meddelanden är meddelanden som inte kommer att resultera i att något svar inkommer.

- Förfrågningar på formen
  `{Pid,Reference,Payload}`
  Alla förfrågningar resulterar i att processen blockerar och inväntar svarsmeddelanden.

- Svarsmeddelanden på formen
  `{Pid,Reference,Request_Reference,{Type,Reply_Payload}}`
  Svar är de meddelanden som skickas som svar på förfrågningar.

För alla meddelanden gäller att `Pid` är process id:t hos den process som skickar meddelandet, `Payload` är vad meddelandet faktiskt innehåller och `Reference` är en referens som den skickande processen ger meddelandet. `Request_Reference` är en referens från en förfrågning som sedan skickas med svaret. På så sätt kan systemet veta vilken förfrågan svaret svarar på.

Då systemet faller under *asynkron eventdriven programmering* så kan vissa problem uppstå då det inte är säkert vilka och i vilken ordning meddelanden kommer hanteras. Lösningen på detta är att använda olika tillstånd. Processens tillståndet dikterar vilka meddelanden som processen kommer att hantera. Sedan passeras meddelandena genom ett meddelandefilter som returnerar rätt meddelande (baserat på meddelandets unika tag/referens) och alla andra meddelanden som inkommer läggs på en buffer så att de senare kan hanteras.

Detta leder till att koden blir kortare och lättare att underhålla. Det finns dock ett krav på att alla processer måste ha ett eller flera tillstånd då de accepterar nya meddelanden. I detta tillstånd kommer meddelanden som ligger på buffern att hanteras först och när buffern är tom kommer de nya meddelandena att hanteras.

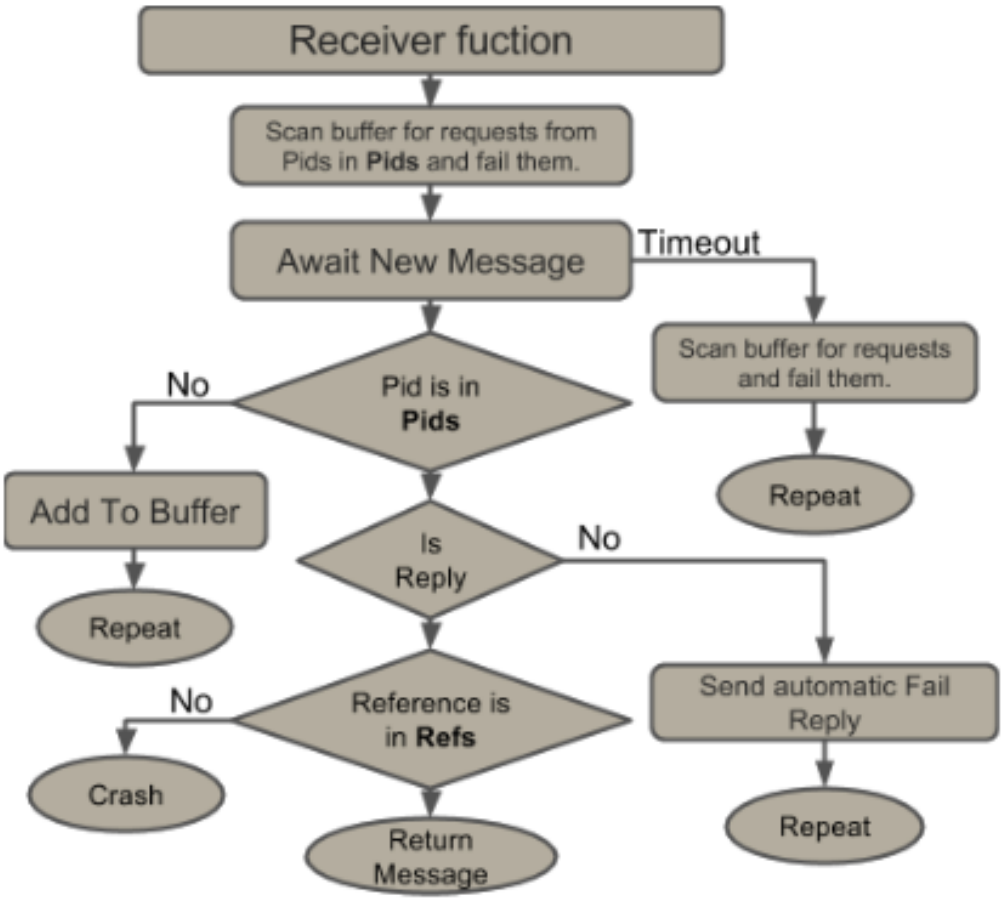
4.3.1 Deadlocks

För att lösa problemet med distribuerade deadlocks använder systemet en metod som är väldigt simpel men mycket effektiv. Metoden löser alla deadlocks helt automtiskt och kräver inte att några rollbacks måste genomföras eller analys av det globala tillståndet med en Wait-For-graf [3]. Metod kräver inte heller att några probe meddelanden skickas mellan processerana som i Chandy-Misra-Haas algoritmen [4]. Dock så kräver metoden ett antal krav på systemet för att fungera.

- Alla förfrågningar kan misslyckas på ett väldefinierat sätt.
- Alla processer kommer att invänta ett svar efter en förfrågan har skickats.
- Från det att ett svar på en förfrågan har inkommit kommer processen alltid att inom en finit tid drivas till ett tillstånd där den accepterar inkommande förfrågningar.

Kraven tillåter att en process som väntar på ett svar fortfarande kan hantera vissa meddelanden och skicka nya förfrågningar till processer. De tillåter även att man implementerar en prioritering av olika typer av förfrågningar eller olika typer av processer.

När en process väntar på svar från en annan process så kommer den att efter en viss förutbestämd tid (timeout), skicka automatiska fail-svar till alla inkomna förfrågningar som ligger på dess meddelande-buffer. Detta kommer att repeteras tills att ett svar har inkommit.



Figur 4: En schematisk överblick över hur meddelandefiltreringen och deadlockhanteringen fungerar. **Refs** är en lista med de referenser från de förfrågningar meddelanden som har skickats. **Pids** är en lista med de Pids som förfrågningarna har skickats till.

5 Avslutning

BugsLife är en simulering av en myrkoloni. Systemet är fullständigt concurrent och drabbas inte av deadlocks. Detta gör att simuleringen känns naturlig och smidig vilket i sin tur gör den intuitiv att använda.

5.1 Diskussion och slutsats

Systemet är speciellt då det endast simluerar en myrkoloni. Detta gör att man kan studera skillnader i hur myrorna beter sig i en miljö där de har respektive inte har konkurrens.

Metoden för att undvika deadlocks upptäcker ofta falska deadlocks, det vill säga att den kommer att tro att det finns ett deadlock när det inte gör det. Detta i sin tur leder till att systemet avvisar vissa förfrågningar som det inte ska. För att fixa detta måste timeout pararmetern vara lagomt lång. En timeout som är för lång kommer innebära att deadlocks kommer att ligga kvar för länge innan de upptäcks och delar av simuleringen kommer att lagga. En för kort timeout kommer att leda till att väldigt många förfrågningar kommer att avisas i onödan vilket också kan påverka systemets prestanda. Om metoden att hitta deadlocks inte avvisade meddelanden felaktigt skulle systemet vara snabbare.

Som vidareutveckling skulle man till exempel kunna använda sig av flera olika typer av mat. Beroende på vilken typ av mat det var skulle myrorna till exempel kunna skicka ut mer feromoner eller en annan typ av feromoner. Något annat man kan lägga till är fientliga insekter eller en konkurrerande myrkoloni. Detta skulle vara användbart då man får en mer realistisk simulering. Detta är mer likt simuleringar av till exempel soldater som rör sig mot en motståndare. Man kan också göra simuleringen interagerbar i realtid. Detta skulle innebära att man kan simulera yttre faktorer myrorna inte har någon kontroll över. Till exempel skulle man kunna simulera att en sten ramlade någonstans. Om det finns ett nytt hinder skulle myrorna behöva lista ut hur man tar sig runt det.

Om stystemet skulle utvecklas vidare för att simluera soldater finns en del etiska aspekter att ta hänsyn till. Simuleringarna skulle kanske ge övertag i krig för den som använde det. Ifall simuleringen är felaktig finns det risk att många soldater dör då det som hände i simuleringen var annorlunda från verkligheten. Att använda systemet för att dra slutsatser om hur djur agerar skulle innebära att man inte behöver testa på verkliga djur. Detta i sin tur skulle leda till mindre djurförsök. Dock så är det inte säkert att verkliga djur skulle bete sig likadant som i simuleringen. Detta skulle innebära att man publicerar falska resultat. Att publicera falska resultat är ett problem av flera orsaker, till exempel förlorar både författaren och utgivaren trovärdighet.

Referenser

[1] M. Dorigo and L. M. Gambardella, "Ant colonies for the travelling salesman problem," *BioSystems*, vol. 43, no. 2, pp. 73–81, 1997.

[2] D. Vasilev, "Erlport documentation," http://erlport.org/docs/, besökt 2023-09-30.

[3] P. Krzyzanowski, "Distributed deadlock," 2012, tillgänglig på https://www.cs.rutgers.edu/ pxk/417/notes/deadlock.html.

[4] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Transactions on Computer Systems (TOCS)*, vol. 1, no. 2, pp. 144–156, 1983.