Intelligent Design

Processorienterad programmering (1DT049) våren 2012. Slutrapport för grupp 2

Emil Vikström - 880728-0493
Patrik Broman - 810416-7138

Erik Samuelsson - 890112-0413

Illustration: John Gould - *The mammals of Australia*

Innehållsförteckning

Inledning

Målet med projektet är att testa evolutionen som koncept. Tanken var att programmet ska simulera några populatier av olika arter som konkurrerar om föda, slåss och parar sig. Det visade sig dock att vi var tvungna att begränsa våra ambitioner betydligt. I projektet används könlös förökning och det finns bara en art. Detta är dock tillräckligt för att få fungerande evolution. För detta krävs egentligen bara två saker, och det är slumpmässiga förändringar och naturligt urval.

Initialt hade vi tänkt ha ett webgränssnitt, men av tidsskäl har vi tvingats ordna ett enkelt CLI.

Intelligent design

Programmet simulerar ett ekosystem med ett antal djur. I varje steg i simuleringen försöker varje djur utföra 10 handlingar: *äta, föröka sig, döda ett annat djur.* Djuret styr själv, genom sina gener, vilka händelser som ska prioriteras (i procent). Djuret har även ett antal nedärvda egenskaper: kroppstemperatur, attackstyrka, försvarsstyrka och maxålder. Både prioriteringar och egenskaper ärvs från föräldern med små slumpmässiga mutationer när djuret föds. Mutationerna kan gå lika långt uppåt som nedåt, vilket gör att om egenskaperna går åt ett visst bestämt håll så har vi ett naturligt urval.

Djuret har också en energimätare. Energin minskar i varje steg beroende på den sammanlagda attack- och försvarsstyrkan. Energin minskar även av att djurets kroppstemperatur ligger för långt ifrån världens utomhustemperatur (utomhustemperaturn ändras slumpmässigt inom ett intervall). När energin når noll dör djuret.

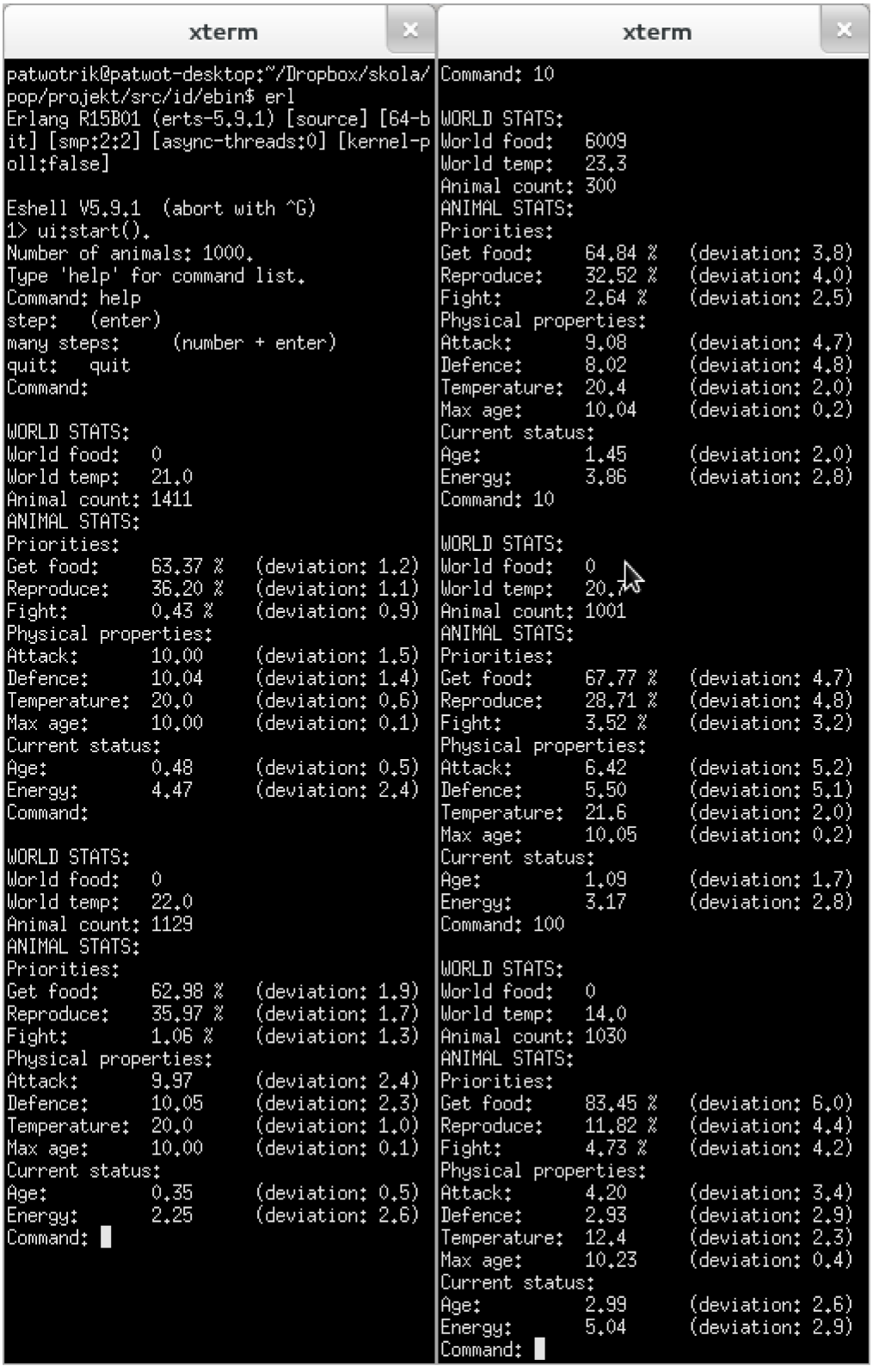
Energin ökar genom att lyckas äta mat från världen (World). World har dock begränsat med mat och en begränsad tillväxt. Ett annat sätt att få energi är genom att lyckas döda ett annat djur, men sannolikheten att lyckas med det beror på den egna attackstyrkan och på offrets försvarsstyrka. Vid lyckad “jakt” dör offret och jägaren får all dess energi. Vid misslyckad jakt tappar jägaren energi bestämt av förhållandet mellan dess attackstyrka och offrets försvarsstyrka.

Programmet har ett enkelt CLI. I detta finns grundläggande funktionalitet för att skapa en ny population och stega framåt i tiden. Efter varje kommando får man statistik på hur världen ser ut (medelvärdet på olika paramterar tillsammans med standardavvikelsen).

För närvarande kan man bara ange hur stort antalet djur ska vara vid start. Andra parameterar måste man ändra direkt i programmets källkod och sedan kompilera. Om man skriver ”help” får man upp hjälpen som visar vad man kan göra med kommandotolken. Se även tabell 1 för de kommandon som finns. I figur 1 visas en typisk körning av programmet.

Tabell 1: Kommandon i programmet

[Enter]	Stega simulationen ett steg
<n>	Stega fram <n> steg
help	Visa hjälpen
quit	Avsluta



Figur 1: En körning av programmet. Notera hur egenskaperna ändras.

Programmeringsspråk

Vi valde språket Erlang eftersom det lämpar sig väldigt väl för system med många processer, då processerna i detta språk är väldigt resurssnåla, men ändå säkra och lätta att jobba med. Tidigt i projektet diskuterades möjligheten att en art skulle vara Erlang-processer och en art C-processer, då detta skulle vara riktigt komiskt ur ett evolutionsperspektiv. Dock insågs vansinnet i detta relativt snabbt.

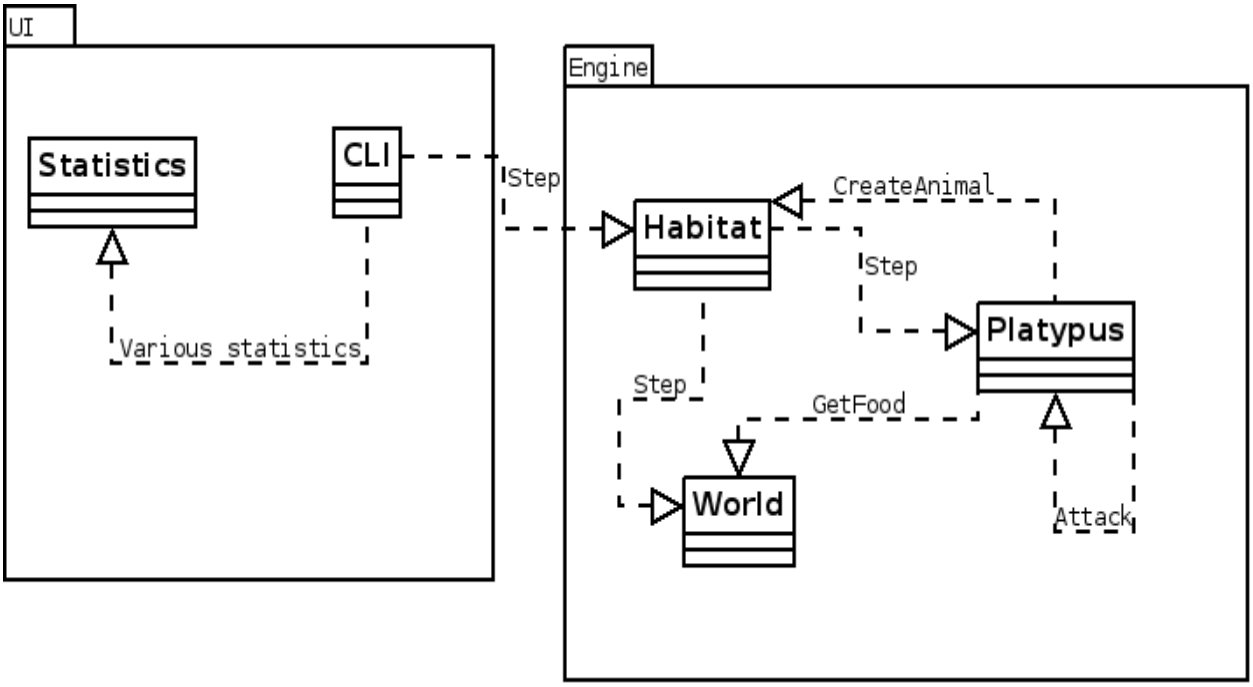
Erlang är ett funktionellt språk med få sidoeffekter. Detta gör det lätt att isolera kod så att den inte påverkar programmets globala tillstånd. Det behövs i princip inga lås eller semaforer eftersom data ändå inte delas mellan olika processer.

Erlang kommer med ett ramverk som kallas OTP med stöd för många av de grundläggande saker man ofta gör. Vi använder det som kallas *gen_server* för våra processer. Gen_server håller koll på processens tillståndsvariabler samt abstraherar bort både synkrona och asynkrona anrop.

En nackdel med Erlang är att ingen i projektgruppen arbetat med språket tidigare. Med bättre förkunskaper hade vi kunnat utnyttja OTP bättre och till exempel försökt oss på att bygga en övervakare (supervisor) med koll på processer som dör och liknande. På grund av att det är ett funktionellt språk är det dessutom väldigt olikt den imperativa programmering vi sysslat med tidigare.

Systemarkitektur

Systemet består i grunden av två delar: Motor och gränssnitt. Gränssnittet skickar instruktioner till motorn, och motorn levererar utdata till gränssnittet vid begäran. Figur 2 visar systemets övegripande struktur samt vilken typ av meddelanden som skickas mellan de olika delarna.



Figur 2: Systemarkitektur

UI

Användaren har tillgång till ett enkelt CLI. Detta kommunicerar med programmotorn. Till sin hjälp har gränssnittet ett enkelt statistikbibliotek.

Motor

Själva programmotorn är uppdelad i tre delar. World håller koll på hur mycket mat det finns, hur fort den produceras och liknande. Platypus styr djurens beteende. Habitat är som en låda som samlar alla djuren tillsammans med världen.

Var och en av dessa är implementerade som *gen_server* i Erlang. Habitat hade möjligen kunnat skrivas som *supervisor* istället då det är den funktion den har, men så har vi inte gjort på grund av okunskap.

World

Här hanteras egenskaper hos miljön. Den styr hur mycket mat som finns, i vilken takt maten produceras och vilken temperatur som råder.

Platypus

Alla djur sköter sig själva, så en simulation har typiskt väldigt många platypus-processer. Var och en har ett ”genom” som styr dess beteende och olika egenskaper. De ber World om mat, kopierar sig, slåss och vilar. När de kopierar sig får avkomman automatiskt lite mutationer.

Habitat

Det är här som allt samverkar. Habitat håller koll på alla djur och miljön. Det är också med Habitat som gränssnittet kommunicerar.

Samtidighet

Den modell för samtidighet som används är actor-modellen, med meddelandepassning mellan processer.

Simulatorn utnyttjar samtidighet genom att varje individ, djur,i simulationen är en egen process. Individen håller själv koll på sina egenskaper, fattar själv beslut om vad den ska ta sig för och så vidare.

Detta skulle kunna leda till att schemaläggaren i systemet ger okontrollerbara fördelar och/eller nackdelar till vissa processer, till exempel att en individ äter upp alla andra individer innan de fått möjlighet att ens börja köra. Det skulle även göra det svårt att kontrollera hur World skapar ny mat; vi vill skapa maten innan djuren börjar slåss om att äta upp den.

För att undvika detta synkroniseras simulationen i *steg*. Varje steg innebär att individerna får utföra 10 olika aktiviteter (äta, jaga, föröka sig). I början av varje steg får processerna ett asynkront start-meddelande från habitatet och när de är klara skickar de ett klart-meddelande tillbaka. Habitatet inväntar alla klart-meddelanden innan den returnerar till gränssnittet så det går inte att dra igång flera steg parallellt. Synkronisering bär även med sig fördelen att det blir lättare att jämföra hur snabbt evolution sker i olika simulationer.

Samtidighetsproblem vi stött på

En samtidighetsflaskhals är att World, som delar ut mat, hanterar begäran om mat seriellt. En tänkbar lösning vore att dela upp World i flera olika processer som kan köras parallellt. Detta är inte implementerat. Samma sorts flaskhals fanns i att varje individ självt frågade World vad utomhustemperaturen är i varje steg. Detta optimerades bort genom att låta Habitat kolla upp det och sedan skicka med till individerna i deras steg-meddelande.

Ett tag körde vi stegningen asynkront i habitatet. Detta gjorde att om man körde flera steg tätt inpå varandra så kunde vissa individer påbörja nästa steg innan World var färdig. Vi stötte på en hel drös med problem här eftersom systemet inte är tänkt att köras på det viset. Detta löstes genom att byta till ett synkroniserat anrop (rent tekniskt bytte vi från gen_server:cast till gen_server:call för steg-meddelandet i Habitat, samt införde en räknare som gör att habitatet inte returnerar innan alla individer är klara med sitt steg).

I jaktsystemet, när ett djur vill döda ett annat djur, behöver de få kontakt med varandra. Den naiva lösningen var att fråga habitatet efter ett slumpmässigt djur. Detta ledde till ett dödslås eftersom habitatet är låst tills alla djur avslutat sina steg. Problemet löstes genom att man redan i stegnings-meddelandet får tilldelat en individ som man kan slås mot om man vill. en bättre lösning vore att bryta ut denna funktion i ytterligare en egen process men detta har vi inte gjort.

Ett annat potentiellt dödslås är ifall två individer samtidigt bestämmer sig för att slåss mot varandra. De kommer då dra iväg varsitt attack-meddelande till den andre och båda inväntar svar. Detta har vi löst genom att ha en kortare timeout på svaret, vilket även löser problemet att en individ kan dö mitt under simulationen och då inte svara på attacken.

Andra modeller

Det är svårt att se hur man skulle skriva systemet med en annan samtidighetsmodell. Det är klart att varje individ hade kunnat vara en egen tråd eller utnyttja en trådpool av något slag, men att då få till kommunikationen mellan slumpmässigt valda individer (i jaktsystemet) hade blivit pilligt. Någon form av brevlåda för meddelanden hade ändå behövt införas av den anledningen. Utan jaktsystemet hade man lätt kunnat använda flera trådar istället eftersom de då bara ska kommunicera med World-tråden under simulationen, något som kan lösas med enkla lås.

En semafor skulle kunna införas redan i det nuvarande systemet för att hålla koll på hur många individer som kör. Man skapar då en semafor med det antal individer som finns, låter varje individ skaffa ett lås från semaforen och slutligen försöker habitatet roffa åt sig samtliga låsen. När en individ är klar med sitt jobb så “lämnar den tillbaka” en enhet till semaforen som då habitet tar. Detta liknar rent logiskt vad vi redan gör när vi räknar hur många klart-meddelanden som kommit in.

Algoritmer och datastrukturer

Grunden i programmet är att man stegar sig fram genom tiden. Under varje tidssteg kommer följande att inträffa:

1. World producerar mer mat.
2. Varje djur utför ett antal handlingar. Dessa handlingar är att skaffa mat, jaga och att föröka sig. Fördelningen mellan dessa är individuell för varje djur, och kan ses som en del av deras genom.
3. Djuren avgör själva om de dör eller fortsätter leva.

Vi har inte använt några speciella datastrukturer. Det mesta lagras i länkade listor eller tupler/records. Vi beskriver egenskaperna hos systemets centrala bitar genom pseudo-C++-kod:

Platypus

```
class Platypus
{
	float energy; // När energy < 0 dör djuret
	float age; // När age > maxAge dör djuret
	float maxAge;
	float optTemperature; // Optimal temperatur för djuret,
				//“kroppstemperaturen”
```

```
	float attack;
	float defence;
	struct {
		float reproduce;
		float getFood;
		float fight;
		float rest;
	} actions; // Sannolikhet för att utföra olika handlingar.
	// Denna struct normaliseras så att summan blir 100.

	void step(); // Utför de handlingar som ska utföras under ett dt
	// samt ändra de attribut som ska ändras. Kolla även
	// om djuret ska överleva eller inte.
	void getStats(); // Självförklarande
	void mutate(); // Mutera djuret slumpmässigt. Används enbart vid
	// reproduktion.
	void reproduce(); // Skapa en muterade kopia. Skicka meddelande
	// till Habitat.
}
```

World

```
class World
{
	const float maxFood;	// I framtiden kanske const
	const float foodGrowth; // kan skrotas på dessa två
	float food;
	float temperature;

	void step() { food += foodGrowth; food=min(food, maxFood);
	temperature += random; }
	float getFood(); // Anropas av Platypus. World avgör om djuret
	// får mat eller inte.
}
```

Habitat

```
class Habitat
{
	World world;
	Platypus animals[]; //Egentligen en länkad lista
```

```
        void step(int n) { // Stega n steg.
                world.step();
                for(int i=0; i<sizeof(animals); i++)
                        animals[i].step();
        }

        void createAnimal();
        void deleteAnimal();
        T getXXX(); // Olika metoder för att titta på data.
}
```

Förslag på förbättringar

Som nämnts tidigare insåg vi att vi var tvungna att begränsa oss för att bli klara i tid, men det finns otaliga förbättringar som vi skulle vilja införa. I stort handlar det om att utöka djurens genom, vilket i kod betyder att vi vill införa fler egenskaper hos djuren som kan förändras. Vi skulle också vilja ha fler olika arter, där en del äter andra arter. Fler olika handlingar skulle också vara önskvärt. För närvarande kan de bara äta eller föröka sig. Vidare skulle det också vara intressant att låta World påverka mer. Istället för att bara förse djuren med mat skulle den kunna innehålla temperatur, luftfuktighet och andra miljöfaktorer.

En tänkbar prestandaförbättring vore om World kunde prata med flera djur samtidigt. Detta skulle kunna lösas genom att World delar upp sig i flera olika delar och avrje djur har en av dessa delar tilldelad sig at kommunicera med.

Gränssnittet skulle kunna förbättras avsevärt. Dels skulle det vara önskvärt med fler funktioner för att analysera datan, men också funktioner för att kunna ange fler olika startvärlden samt ändra parametrar i realtid. Det vore också väldigt bra om lite grafer kunde ritas ut automatiskt. Eventuellt skulle detta kunna lösas genom samverkan med Octave. Ett webbgränssnitt, som vi ursprungligen tänkt, hade också varit stiligt.

Reflektion

Något som vi märkte var väldigt bra var att vi separerade UI från motor redan från början. När vi märkte att vi tagit oss vatten över huvudet med det webbaserade gränssnittet var det bara att skrota den idén utan att det påverkade det övriga arbetet.

Ett misstag som var väldigt lätt att göra, och som vi gjorde flera gånger, var att försöka få det resultat vi vill ha. Som exempel kan nämnas när könsmogen ålder implementerades. Det var ett smått desperat försök att få maxåldern att stabilisera sig. Det felaktiga var att vi införde en begränsning som naturen aldrig skulle införa själv. I naturen är könsmogen ålder en följd av att barnen inte kan ta hand om sig själva i ung ålder och liknande. Den är inte ett självändamål. I vår simulation klarar sig ungarna precis lika bra som de vuxna, och föräldrarna struntar

fullständigt i sin avkomma när den väl har fötts. Djuren påminner mer om bakterier än om riktiga djur. Kort sagt så var det lätt att försöka implementera följderna av en korrekt avbildning av verkligheten snarare än att försöka se till att följderna kommer naturligt av en korrekt avbildningen av världen.

Projektmässigt har vår grupp fungerat sådär. Vi har haft svårt att träffas och inte kodat speciellt mycket när vi varit ifrån varann. Vi hade planer på att höras vid varje dag men det höll inte heller. Jag tror att det vore bättre med en speciellt utsedd projektledare med tillgång till allas telefonnummer, och kanske en noggrant planerad tidsplan.

Ett återkommande stressmoment har varit att ingen riktigt känt till hur projektet ska gå till, både vad gäller dödslinjer och vad vi förväntas leverera. Flera gånger har vi plötsligt hört från andra grupper att man ska boka mötestillfällen och liknande. Även här hade en schemalagd tidsplan med datum varit uppskattat. Ingen i gruppen använder heller det gamla intranätet PingPong dagligen, vilket gjort att information smitit förbi obemärkt.

Installation och fortsatt utveckling

Du är välkommen att ladda ner och leka med systemet.

Systemkrav

- Erlang (vi har använt R15, version 5.9.1, men andra varianter bör fungera)
- EUnit (paketen erlang-dev i Debian, erlang-eunit i Ubuntu)
- make (för kompilering)

Hämta koden

Senaste versionen finns på www.github.com/emilv/id
Hämta med ***git clone git://github.com/emilv/id.git***

Vi har en mycket platt katalogstruktur. Koden ligger i src/ med det kompilerade programmet i ebin/ och genererad dokumentation i doc/
Varje modul är en egen fil, så som sig bör i Erlang. Testfallen ligger i samma fil som respektive modul.

Installation

make all
Programmet hamnar i mappen ebin/

Körning

- Öppna en kommandoprompt
- Gå till ebin/

- Starta Erlang-tolken med kommandot ***erl***
- Kör ***ui:start().*** (inklusive punkten)
- Ange antal djur i starten (avsluta med en punkt även här)
- Stega framåt med *[Enter]*. Skriv ***help*** för för mer hjälp

Testfall

Några få testfall finns skrivna i EUnit. De som finns körs med ***make test***

Dokumentation

Användardokumentation är inbyggt i gränssnittet. Systemdokumentation genereras utifrån EDoc-kommentarer i koden genom att köra ***make doc***

Licens

Kopimi: Kopiera gärna programmet. Sprid det vidare, ändra på det hur du vill, använd det fritt.