Intelligent Design

Emil Vikström
880728-0493

Patrik Broman
810416-7138

Erik Samuelsson
890112-0413

Alexander Persson
910207-1033

Process Oriented Programming (1DT049),
project proposal for group 2,
version 1, 2012-04-25

1 Brainstorm

We brainstormed to get different ideas. The result of that session is presented here.

1.1 Preconceptions

Before starting we thought of brainstorming as a good way to get a lot of ideas (good and bad ones).

1.2 Summary

We introduce some ideas and evaluate the concurrent possibilities of the ideas.

1.2.1 Ideas

Conway's Game of Life A grid with cells, each cell live or dead according to a set of rules depending on neighbouring cells.

Chat Server IRC or similar.

Evolution simulation Similar to Game of Life, but each cell have a set of properties such as attack, defense, energy need, temperature range... Properties is inherited or possibly randomly mutated.

Web Crawler Crawl web pages, following links and extractig data (links to other sites or important words from the page). Possibly specialised for certain web sites like Wikipedia or similar. Can possibly be extended to build a search engine.

MMO Some kind of multiplayer game with lots of players.

Facemash Web page showing a pair images, letting the audience pick a "winner" amongst them and then showing a new pair. Rank images using some algorithm.

Compiler Compile and optimize some language.

Robo Rally Board game where each player "program" a robot with five instructions at a time, trying to reach valuable targets on the game field. Each robot, however, run only one instruction at a time before it's the next robot's turn. Robots may move each other, making it harder to reach the target.

1.2.2 Concurrency Evaluation

Conway's Game of Life Each cell is only dependent on its neighbours. Therefore it should be possible to calculate the new state of a cell (according to the rules) concurrently with knowledge about the neighbours. This may be done by checking neighbours' states.

Chat Server Users and "rooms" are logically independent units. Messages are sent between users and rooms. We can easily see this implemented using message passing.

Evolution Simulation Cells are dependent on local factors like temperature, neighbours and the cell's properaties. Calculations should be possible to do concurrently in each cell with knowledge about the surroundings.

Web Crawler Web pages may be independent units. Different pages can then be crawled, parsed and saved in a concurrent fashion. This problem is also heavilly I/O bound due to both network traffic and performance of the web servers. This means that a large amount of web pages should be possible to crawl at the same time.

MMO An MMO must be implemented in some concurrent fashion because a large amount of users (with possibly different state) must be handled at once. The exact circumstances are defined by the game rules.

Facemash Running "pair competitions" is a concurrent problem since possibly thousands of users may use the service at once. Designing a ranking algorithm which is concurrent may be part of the project problem.

Compiler Concurrency depends on the language compiled. Some possibilites are to compile functions or modules independently.

Robo Rally The only concurrency possibility is to have different playing rooms and handling user connections conccurently. The game itself is turn-based and inherently sequential.

1.3 Conclusion

We didn't come up with that many ideas. The reason may be that the few ideas we came up with was that good that we didn't find it worthwhile to continue.

The next time we should probably try to come up with some bad ideas as well. Really bad ideas may spark creativity.

2 Project Selection

We chose "Evolution simulation". The difficulty seems to be at a good level and the simulation also seems very good for concurrency. We also considered Conway's Game of Life and chat server, which both seemed to be too simple or at least not that interesting. Robo Rally is hard to make truly concurrent, which was a big showstopper.

Evolution seems interesting both in the sense of learning how to program concurrently, but also to get a better understanding for how evolution works. We will not fine tune the program to reach a certain result. We will rather put in what we believe is a good model and see the result from various parameters.

Apart from what has already been mentioned we hope to learn more about cooperation when programming. This is also what we believe will be the biggest challenge.

3 System Architecture

We here describe the main idea with the simulation, as well as a sketch of the system design.

3.1 Main Ideas

We focus on two main components:

- Environment
- Animals

The environment is where the animals live. Examples of properties is temperature, light, food, humidity, radiation etc.

It exists two different animals, herbivores and predators. Every individual should have genes that specifies different properties, such as prefered temperature, ability to withstand different environments etc. These properties will in some cases be ordered in pairs, like more muscles means a need for more food.

There will be random mutations which alter the properties. Animals will be able to mate, but there will be constraints so that an individual only can mate other individuals that are close enough (both locallly close and "genetically" close).

We plan to let every individual be a separate process in Erlang.

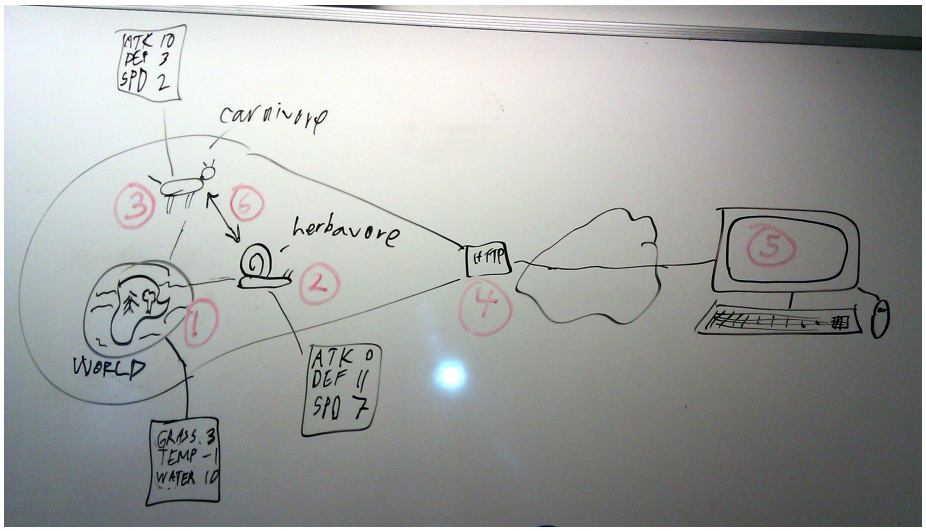

Figure 1: Simulation Design

Table 1: Example Animal

Attribute	Value
Attack	10
Defence	5
Energy consumption	2
Current energy	22
...	...

The simulation is going to be done step by step, and each animal (process) should handle its own actions and interactions with the environment and other animals. After each step, each process sends information to some GUI process.

At first the environment will be constant, but it is possible that we will try to use an environment that is changing over time.

3.2 Design

The system will consist of one "world" with attributes such as "grass" (that herbavores can eat), "temperature" and similar (see figure 1). In this world there will be "animals" which can communicate with each other. The communcation will consist of things such as "attack" (from a carnivore trying to eat someone), and the receiving animal will decide if the attack succeeds or not (replying this to the attacker).

The world will be it's own entity, which acts as a collection of animals but also faciilitates some environment variables such as "grass". Animals will also be their own entities, which can communicate with the world to get handles to

other animals and then directly communcate with those animals.

Animals have attributes (see table 1) which decides how easily they kill someone (attack), how much energy they consume (which determines how often and how much they must eat) and so on. Some attributes keep track of the current state of the animal, such as "current energy". When that energy is at zero, the animal dies. Some attributes may be dependant on each other, such as a high attack attribute may require higher energy consumption.

The GUI will be shown in the user's web browser. This means that the user will communicate with a HTTP process on the server side. This HTTP process is responsible for keeping a handle to the user's entire simulation. The HTTP process is responsible for translating Erlang messages to ECMAScript and/or HTML.

3.2.1 Animal Life

As mentioned before, each animal will be a separate process, with some properties. The main loop for each animal will be something like this (the exact details to be found out during implementation):

1. Try to get some food
2. Try to survive the environment
3. Try to mate
4. Report to GUI

For simplicity, the animals are pure herbavores or pure carnivores.

4 Concurrency Models and Programming Languages

The way this project is concurrent is that every animal is a separate process, making it possible for all animals to run its main loop concurrently. Some things has to be sequential, but for most things, like testing if an animal survives the environment with respect to temperature, radiation etc, each animal can check if they manage.

The system is also concurrent in it's simluation handling: multiple simulations can be carried out by different users at the same time. This will pose no big concurrency problems since simulations don't share any global state.

The language we are going to use is Erlang, since processes are really easy to handle in that language. GUI will also be made using ECMAScript (the only language supported by web browsers).

5 Development tools

The project requires multiple tools to build a stable product. We touch on the subject here by presenting our editor, source control, testing and building tools of choice.

5.1 Source Code Editor

Emacs is the editor of choice due to the coherent indentation style for Erlang. It would be nice to let each developer pick his own editor, but then it's hard to use the same indentation. Developers are free to choose another editor if they can manage to configure it to the style of erlang.el for Emacs.

5.2 Revision Control and Source Code Management

We are planning to use Git for source control. It will be hosted on the free service GitHub.

5.3 Build Tool

We will use a simple Makefile for the project. The Makefile will provide instructions for both building and testing the program. There is a build system for Erlang called Rebar which we will possibly utilise as well, in which case we will implement it into the same Makefile.

5.4 Unit Testing

Unit testing will be done with EUnit. Each module will have it's own unit tests. Each function will have at least one test, testing the general case. Optimally functions will also test border cases, which we will aim at as well. At a later stage we will also write some integration tests.

Some testing weill be done at the ECMAScript side of the GUI using the Jasmine framework. We will have more relaxed testing here, though, due to the complexity of testing a GUI. GUI testing will instead mainly be carried out by manually checking that everything works.

5.5 Documentation Generation

Every function should be documented with EDoc. A function is not complete before it is documented. Documentation can be quite informal, as long as it's clear what the purpose of the function is.

ECMAScript will be documented using ScriptDoc.