

USING JAVA TO IMPLEMENT AGENTS IN ARTIFICIAL ECOSYSTEMS¹

<http://www.cnptia.embrapa.br/~kleber/AE>

Kleber X. S. de Souza, Mario A. Nascimento

*Embrapa - Brazilian Agricultural Research Corporation, National Center for Technological Research on Agricultural Informatics, P.O. Box 6041, 13083-970 Campinas SP BRAZIL, {kleber, mario}@cnptia.embrapa.br
<http://www.cnptia.embrapa.br/~{mario, kleber}>*

Abstract: An Artificial Ecosystem is a self-contained environment populated by independent entities representing some level of organization. Agents implement levels of abstraction representing entities in a system. Agents can also explore parallelism and cooperative work. artificial ecosystems can thus be implemented using agents, where a global pattern is not coded in the agents specifications, but is a result of their interactions. The Java language provides parallel threads of control and runs in a number of very distinct hardware architectures. As such, in this paper we illustrate the suitability of Java to implement artificial ecosystems using the artificial ecosystems paradigm.

Keywords: Agents, Artificial Ecosystems, Java, Cooperative Work, Cellular Automata.

1 Introduction

Artificial Ecosystems have been defined in the literature (Olson & Sequeira, 1995) as a system in which a self contained environment is populated by independent entities (agents) representing some level of organization. They are a class of the Artificial Life Model, which is an abstraction of natural living systems, preserving some characteristics of those systems that are the subject of study. The agents in an ecosystem interact locally with each other and with their environments.

The modeling of such systems and its computational implementation has faced some difficulties. Mathematical models are limited to analytical cases for which there are tractable solutions. As a result, only system models of extreme simplicity as, for example, simple differential equations with very few variables, can be solved. For instance, even a simple system representing a predator-prey relationship may result in a system of nonlinear differential equations (Batchelet, 1984) for which there is no explicit analytical solution.

As the size of the problem increases, an alternative to analytical models is the use of simulation. However, it has been argued that traditional simulation models are inadequate to implement Artificial

¹ Java is a registered trademark of Sun Microsystems.

ecosystems. They have always some sort of embedded constraints due to the model itself, forcing the system to behave according to the strict rules the model is based upon. This characteristic has been called in the literature "structure dominance" (Olson & Sequeira, 1995).

An alternative is the use of *agents*. The rationale behind the modeling through agents (Olson & Sequeira, 1995) is that each of them implements the level of abstraction necessary to represent an entity while leaving aside aspects considered less important. Their behavior specifies to which stimuli they should react, interacting with the surrounding environment and/or with other agents. Since the level of granularity is the behavior of an agent, the global patterns at the macroscopic level are not explicitly coded in their specifications. They are rather the result of the interaction between agents and between them and their environment.

Several proposals have emerged to precisely define the macroscopic behavior. Among them, we can cite: cellular automata, artificial neural networks and genetic algorithms (Olson & Sequeira, 1995). They all keep the same basic idea of independent entities. What really changes is the information exchanged: information about the surrounding agents (cells in a cellular automata), the weight to be considered when interacting with certain agents (neurodes in artificial neural networks) and encoded string like data structures called chromosomes (genetic algorithms).

The modeling using agents created some new opportunities still unexplored, such as parallelism and cooperative work. Parallelism comes from the fact that agents are self contained entities whose execution is not dictated by a central control. Cooperative work is gained through the interaction among the agents themselves. In addition, their execution could be performed (in parallel) over a computer network. Cooperative work yields, that being separately modeled and implemented, the construction of a system could be splitted among domain experts, each of them specifying the entity he/she knows better. That poses a problem though, namely the heterogeneity of computer hardware and operating systems. The Java programming language addresses both problems. It provides native commands for exploiting parallelism and the Java runtime byte- code, which runs in any architecture for which a Java interpreter is available. Also, due to byte- code, the transmission of messages across the network is possible without any conversion, regardless of the various machines' storage structure.

The paper is organized as follows. In Section 2 we describe the idea of agents. Next, in Section 3, we present, briefly, some of Java's features, that are particularly interesting for our domain of application. To demonstrate Java's suitability, the paper is illustrated with an example in Section 4 and some of the Java source code implementing such example using cellular automata (part of the source code is presented in the Appendix). We conclude the paper in Section 5.

2 Agents

The research in Distributed Artificial Intelligence is divided into two basic classes (Kraus et al., 1995): Distributed Problem Solving and Multi-Agent Systems. The former considers how the work involved in solving a particular problem can be divided among a number of agents which are centrally coordinated and designed to achieve the global goal. Usually, negotiation is used as a mechanism for assigning tasks to agents, for resource allocation and even for deciding which tasks to undertake (Zlotkin & Rosenschein, 1991). In the latter there is no global control, no globally consistent knowledge, and no common goals or success criteria. The concept of agents we use in this paper is the one used in the second approach, i.e., an autonomous entity with its own utility functions. This function represents its reason to live for and it will try to maximize that function during its entire life span. As a consequence, the competition among agents becomes a reality. As we can see, this approach is closer to the behaviour of an ecosystem than the former because associations among life beings do exist, but only when there is a motivation linked to the possibility of a reward from that association.

The global behaviour of the entire system is therefore not hard coded in the specifications of the individual agents, but rather it is an outcome of the interactions among agents. Therefore, a global trajectory of the state of the system is hard to predict beforehand. Notice that in this way the system is non-deterministic, i.e., closer to our perception of an (artificial) ecosystem's behaviour. As agents must mimic actions performed by real entities, their behaviour can be expressed through the following directives:

- perceive what is happening in the environment (sense);
- take some local decisions based in that perception (think);
- change the environment by interacting with one of its components (act);
- create new agents (reproduce);

How those directives are mapped to the supporting language is discussed in the following section.

3 The Java Language

The Java programming language, developed at Sun Microsystems (Sun, 1997), was designed to be a machine independent programming language, safe to traverse networks and powerful enough to replace native executable code (Niemeyer & Peck, 1996). To make a language independent from underlying operating systems or machine architectures one has to make it interpretable by an interpreter implemented on that system. Unfortunately, that imposes the penalty of reducing the speed of execution. Java's designers, however, greatly minimized this problem by generating an intermediate code called Java compiled byte-code.

Java byte-code is read by the Java run-time interpreter, which acts as a normal processor. However, before beginning the execution of the byte-code, the interpreter verifies whether the read blocks of code are in accordance to the strict specification defined for a byte-code. That can be made because Java left nothing undefined nor implementation dependent regarding the language itself and the byte-code generated by the compiler. As a matter of fact, those specifications are openly available for whoever wants to construct an interpreter.

The strict specification of the language and byte-code allows for the fulfillment of other important point, viz. the security mechanisms. They are extremely desired for someone to accept a code which might have traversed the Internet before reaching his computer. Such a code, if not surveilled, could install a virus or access the system files and send them through the net even without the user awareness. Therefore, before being executed the byte-code first is analyzed by the byte-code verifier, and then by the class loader and the security manager. Their function is to make sure the code is in accordance to the language and is well behaved, i.e. does not try to forge pointers, violate permissions or replace built-in system classes.

3.1 Concurrency in Java

In this paper the term *concurrency* is used in the sense of potential parallelism (as in Ben-Ari,1982). This means that whenever there is more than one processor available two tasks can be carried out at the same time, resulting in true parallelism. By doing so, it is possible to explore parallelism without having to worry about the number of processors in a certain machine.

One of the ways operating systems and languages have used to allow users to explore concurrency is providing *user level threads* (Goscinski, 1992). Using the abstraction provided by threads a user may decide which of the tasks to be performed could run in parallel and which were to be serialized. The implementation of the threads in that particular machine would be responsible to map threads to processors in the architecture. Nevertheless, if the language does not support threads, as is the case for the C (Kernigham & Ritchie, 1978) and C++ (Stroustrup, 1994) programming languages, writing code

that deal with them is much more difficult. Moreover, in most cases there are no assurances from libraries vendors that their code is thread safe, which means that it is not implemented in a way such that it can guarantee it can be executed by multiple concurrent threads without resulting in inconsistencies. That is not the case in Java, whose run-time libraries are indeed thread safe.

Java provides full support for threads in its syntactic and semantic level. The Java library provides a Thread class whose methods can start, run stop and check the state of a thread. Also, there are a sophisticated set of primitives for synchronization based on monitors and condition variables. The word *synchronized* in the declaration of a method means it cannot run concurrently. So, the thread which is to execute that method will wait until the executing thread has left that method.

4 A predator-prey example

To illustrate the use of Java in the modeling of an artificial ecosystem, the predator-prey example found in (Eckart, 1997) was used. Such example was also built based upon the Starlogo program called Rabbits developed in Starlogo Project at MIT (Starlogo Project,1997). In that example, the world (a cell array) is populated by grass (prey) and rabbits (predators). The example performs as follows:

- **Initial state:** a certain amount of grass is distributed randomly in the world and a certain number of rabbits is randomly positioned. Rabbits are created with a pre-defined amount of energy;
- **Move:** rabbits' energy is increased whenever they eat grass. They move to another cell randomly, spending energy, where they can find grass or not. If they do, they eat it all;
- **Reproduce:** rabbits reproduce when they have accumulated a determined level of energy, generating other two rabbits. Grass has a growing rate which is kept constant during the whole process;
- **Die:** when a rabbit has not enough energy to move it dies.

Part of the Java code used to implement the example is shown in the Appendix. Further details and the complete source code can be obtained at the URL in the title page, or directly from the authors.

5 Conclusion

We have argued why traditional approaches such as mathematical modeling and simulation, while powerful, may not be the most suitable ones when dealing with artificial ecosystems. On the other hand, the paradigm of agents provides interesting capabilities for such task. Namely, cooperative work and functional parallelism. In addition, the global behaviour of the ecosystems is not explicitly coded, it naturally arises from the agents behaviour. We have also argued that the Java programming language is suitable for implementing agents. In fact, it runs in a plethora of hardware architectures, provides independent, parallel and potentially cooperating threads of execution.

To illustrate our claims we have implemented (and shown part of the Java source code) a simple artificial ecosystem using a cellular automata approach based on agents. We feel that Java showed itself indeed suitable for implementing agents and particularly the model of artificial ecosystems. The implemented model is extensible in the sense that the agents are independent by construction and therefore new agents can be added at any time, as long as their interaction among themselves and the other existing agents is properly specified.

Regarding directions for future research we envision work on designing a program design methodology which will ease re-use of previously constructed agents. In a sense, what we mean is to bring the concept of encapsulation from object-orientation in general to a specific domain of application, ecosystems in particular.

Another potential venue for further work is to allow a distributed execution of the artificial ecosystems. That is, consider a particular machine running an instance of the ecosystem model. By distributed execution we envision the possibility of another piece of code, modeling additional agents to be sent from a different machine over to the one executing the model. Naturally, the main machine (the one originally executing) must be able to interact in such a way.

6 Appendix - Java Code

GrassMgr.java

```
import java.util.Vector;
import java.util.Random;

public class GrassMgr implements Runnable {
    private Thread updateThread;
    private Cell cellArr[][];
    private int arrDim;
    Random rgen = new Random();

    public void run(){
        ...
        while(true){
            Thread.sleep(100);
            Vector v;
            for(int i=0; i<arrDim; i++)
                for(int j=0; j<arrDim; j++) {
                    v = cellArr[i][j].inspect();
                    if(!v.contains(this)){
                        if(Math.abs(rgen.nextFloat()/2)<=0.004)
                            cellArr[i][j].add(this);
                    }
                }
        }
    }

    GrassMgr(Cell[][] ca){...}
}
```

Rabbit.java

```
import java.util.Vector;
import java.util.Enumeration;
import java.util.Random;

public class Rabbit implements Runnable {
    int i,j;
    int energy=0;
    private Thread activity;
    static private Cell cellArray[][];
    static private int arrLength;

    public void run(){
        while(true){
            Thread.sleep(500);
            Vector v = cellArray[i][j].inspect();
            Enumeration e=v.elements();
            while(e.hasMoreElements()){
                Object o=e.nextElement();
                if(o instanceof GrassMgr){
                    if(cellArray[i][j].del(o)) energy+=5;
                }
            }
        }
    }
}
```

```

    }
    if(energy>200){
        Rabbit r1 = new Rabbit(cellArray, newPos(i),
            newPos(j), energy/3);
        Rabbit r2 = new Rabbit(cellArray, newPos(i),
            newPos(j), energy/3);
        energy/=3;
    }
    if(energy>=2){
        cellArray[i][j].del(this);
        cellArray[i=newPos(i)][j=newPos(j)].add(this);
        energy-=2;
    }else{
        cellArray[i][j].del(this);
        activity.stop();
    }
}
}
}
public int newPos(int pos){
    Random rgen = new Random();
    return((pos+rgen.nextInt()%2+arrLength)%arrLength);
}
}
Rabbit(Cell[][] ca,int fdim,int sdim, int engy){...}
}

```

7 References

- Batchelet, E. (1984). *Introdução à Matemática para Biocientistas*. Editora Interciência, Rio de Janeiro, 596 pp.
- Ben-Ari, M. (1982) *Principles of Concurrent Programming*. Prentice-Hall International.
- Eckart, D. J. (1997) <http://www.runet.edu/~dana/ca/examples>. Available as of February 22, 1997.
- Goscinski, A. (1992) *Distributed Operating Systems - The Logical Design*. Addison-Wesley, Sydney. 913 pp.
- Kernighan, B.W.& Ritchie, D.M. (1978). *The C Programming Language*. Prentice-Hall, Englewood Cliffs, USA, 228 pp.
- Kraus, S, J. Wilkenfeld & G. Zlotkin (1995). Multiagent Negotiation Under Time Constraints. *Artificial Intelligence*. **75**:297-345.
- Niemeyer P. & J. Peck (1996). *Exploring Java*. O'Reilly & Associates, Sebastopol, USA. 407 pp.
- Stroustrup, B (1994). *The C++ Programming Language*, Addison-Wesley, Reading, USA, 691 pp.
- Olson, R. L. & R. A. Sequeira (1995). Emergent Computation and the Modeling and Management of Ecological Systems. *Computers and Electronics in Agriculture*. **12**:183-209.
- Sun Microsystems (1997). <http://java.sun.com/>. Available as of February 22, 1997.
- Starlogo Project (1997). <http://lcs.www.media.mit.edu/groups/el/Projects/starlogo/projects/rabbits.html>. Available as of February 22, 1997.
- Zlotkin, G. & J. S. Rosenschein (1991). Cooperation and Conflict Resolution via Negotiation Among Autonomous Systems in Noncooperative Domains. *IEEE Transaction on Systems, Man and Cybernetics*. **21(6)**: 1317-1324.