ACM Ada Letters (March/April 1995), 15(2): 46-53

# Implementing self-reproducing artificial organisms with Ada

Philippe Laval

Observatoire océanologique
CNRS URA 716 - Station zoologique
BP 28, 06230 Villefranche-sur-Mer, France
laval@ccrv.obs-vlfr.fr

Artificial Life techniques have a potential use in ecological models, but they have not yet been programmed in Ada. Ada tasks provide a powerful means to express the behavior of self-reproductive autonomous individuals. This article shows how artificial organisms with a complicated reproduction (alternation of sexual and asexual generations) have been implemented in an ecological simulation software written in Ada.

## Introduction

Artificial Life (ALife) techniques, the representation of biological cells or organisms by computer programs, are beginning to be used by biologists and ecologists as a modeling tool [8]. When the entities simulated in software are organisms, Alife models offer an alternative to the classical study of populations with mathematical equations.

Alife programming is done mainly with Lisp dialects — an understandable choice, because Lisp leads naturally to "evolutionary programming" (where the source code itself undergoes mutations and substitutions while running). Concurrency in Alife software is mostly considered in terms of highly parallel computers running small identical pieces of code in multiple nodes.

Another recent trend in ecology is the use of object-oriented programming [3], using Smalltalk or C++. In ecological simulations, concurrency is found in event-driven Smalltalk programs.

The exploratory nature of ecological modeling may explain why Ada is not usually chosen for an activity rather related to prototyping. Also the steep learning curve for mastering Ada is not appealing to biologists.

However, Ada provides many features favoring object-oriented (or more exactly for Ada83, object-based) programming, as well as concurrent object behavior. Murray [7] has advocated the use of Ada tasks to represent ecological processes. But tasks constitute only an aspect (the dynamic one) of an ecological model. Software engineering methods devised for dealing with complex software systems may be useful to understand ecosystems, which are another kind of complex systems.

An Ada object-based design of a small ecosystem simulation may be found in a companion paper [5]; this software, named CALIFE (for Computational Artificial LIFE) is designed along the HOOD (Hierarchical Object-Oriented Design) methodology [4]. The present article explains how the artificial organisms constituting one of the "objects" in CALIFE (the biological population under study) are implemented in Ada.

For readers not familiar with biology, however, a short description of the relevant features of these organisms is necessary.

## 1. The real organisms

The individuals represented in CALIFE are gelatinous planktonic animals a few cm long, members of the class of Tunicates (see for example [1]). They play an important ecological role because when conditions are favorable they reproduce very rapidly, and can "bloom" forming swarms covering thousands of km$^2$ [6]. This high reproductive rate is made possible through a budding phase in their life cycle, in which each individual may produce (asexually) more than 100 individuals. These asexually produced individuals remain attached to one another, forming a "chain". In biological terms, the oozooid (individual issued from the egg) gives birth to a chain of blastozooids (individuals arising from budding — also named aggregates). When the chain eventually breaks down, each blastozooid, when fecundated, produces an egg, which yields an oozooid. An oozooid may produce two or three generations of chains. The fecundation of an aggregate leads to only one oozooid.

This complicated mode of reproduction was one of the incentives of the CALIFE implementation in Ada: it is an example of the power and expressiveness of the language.

## 2. Ada implementation of a Tunicate

### *2.1 Biological structure*

In the CALIFE object-based design, the object Tunicates represents a population of organisms issued from one initial individual. The modeler (the user of the software) only sets the stage for the simulation and observes what is happening. There is no user interaction with the organisms after the initial parameters values are read. This point has profound implications for the design.

In an Alife representation, Tunicates should be modeled as self-reproducing entities with concurrent individual behaviors. Each individual should have local states, and should maintain a local clock. An artificial Tunicate should exist in two forms: oozooid and chain, with oozooids producing chains, and each individual in the chain producing an oozooid.

For simplicity a chain is represented in the software by one "super-individual". This is not an important bias, because each individual in a chain is a clone; all individuals in a chain behave identically — move together, breath the same, etc. Moreover, with this convention a chain occupies just one position in "space". For modeling purposes, the physiology of a chain is simply the sum of the individual physiologies of its constituents. In the following presentation the term *zooid* will refer to any "individual", whether an oozooid or a chain.

The preceding specifications call for a *Tunicates* package, with an abstract data type *Zooid*, and a visible procedure *Create* (taking a parameter of type *Zooid*). The private type *Zooid* is an access type to a variant record structure, *Zooid_Pattern*. The record discriminant governs the fields characteristic of either an oozooid or a chain. The fixed fields contain information such as the individual birth date, or current position, swimming direction, etc.

### *2.2 Behavior*

The concurrent behavior of an individual is naturally captured by a task. Each individual has a field named *Action* in the *Zooid_Pattern* record pointing to a task type *Vital_Process*. The task has an entry *Start* (with an *in* parameter of type *Zooid*). The main internal loop inside the task periodically executes a delay statement. The loop is entered at birth (just after the entry *Start* is accepted); age increments accumulate until a *Longevity* parameter is reached (unless some external event causes the death of the individual). The behavioral events (moving, feeding, reproducing, etc.) take place during each main loop iteration.

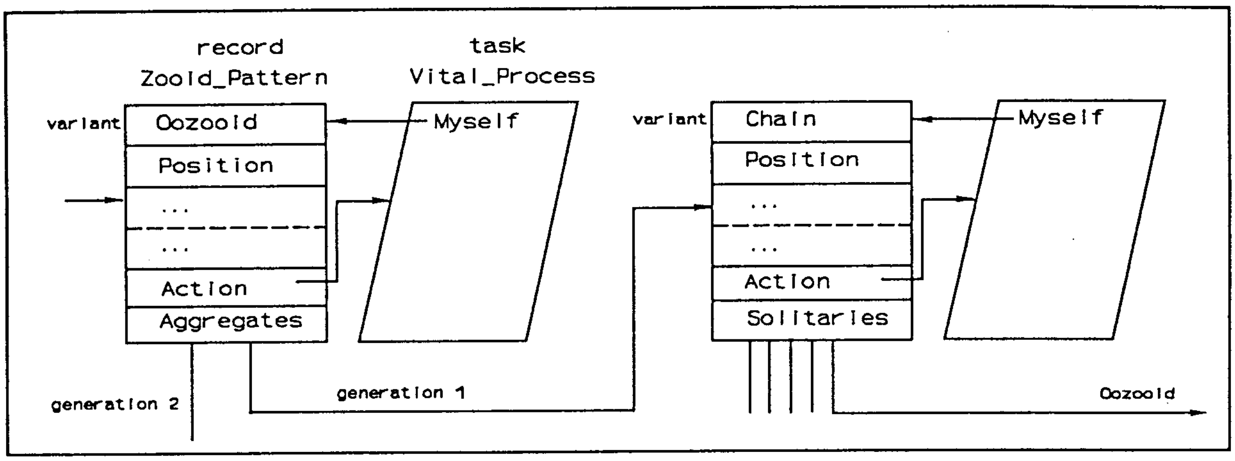

Figure 1: Representation of a Tunicate generation

All these events need to refer to the *Zooid_Pattern* record structure: moving requires updating the current position, feeding the current amount of reserves, and so on. This is done in the task with an access value to the record structure. This access value is stored in a local variable called *Myself* (Fig. 1). To make the first individual, the *Create* procedure allocates a *Zooid_Pattern* record; the access value returned by the allocator provides an access to the first individual. The record is initialized with a qualified aggregate, in which the *Action* field is initialized by allocating a task of a *Vital_Process* task type. The task begins immediately executing but waits at an initial rendezvous at its *Start* entry. A copy of the access value designating the first individual is then passed during the rendezvous to the *Myself* local variable. The task may thus refer to the *Zooid_Pattern* record.

### 2.2.1 Reproduction

Reproduction constitutes a slightly more difficult problem: when maturation time is elapsed, the first individual (arbitrarily chosen to be an oozooid) should spawn a new individual (this time a chain). When the chain is mature, it should spawn as many oozooids as there are individuals in the chain, and so on.

For reasons which will be explained later, each "child" is connected to its "parent" with an access value. The *Zooid_Pattern* variant "*Chain*" has an *Aggregates* field, which contains an array of *Zooid* access values indexed by the number of individuals (aggregates) in a chain. An array of access values is not necessary for the variant "*Oozooid*" (which produces only one chain), but as each oozooid may reproduce several times, it is necessary to connect the different generations of chains to their parent oozooid. So in the variant "*Oozooid*" there is also an array of *Zooid* access values, this time indexed by the number of generations.

The reproduction of an artificial Tunicate involves 3 major steps.

(1) The parent task allocates from the memory a *Zooid_Pattern* record, of the opposite variant. The *Action* field of the child is initialized by assigning to it the access value resulting from an allocator for a *Vital_Process* task type; this has the effect of connecting the task of the child to the parent *Action* field;

(2) The parent connects the appropriate location of its array of *Zooid* access values to the child record;
(3) The parent calls the child task entry *Start*, passing as a parameter the address of the record it has allocated for the child.

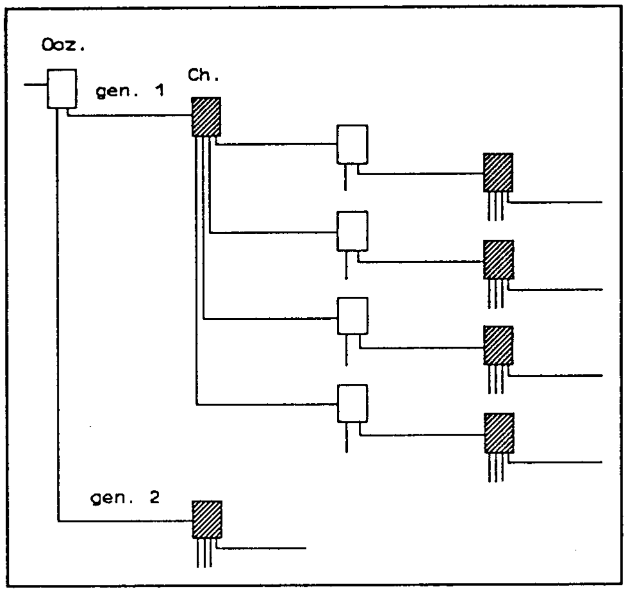

Figure 2: The linked list of zooids

The whole reproductive process yields a multibranched linked list (Fig. 2). After the 3 above steps, the newly born tunicate is autonomous. It "lives" concurrently with the other tunicates. The new individual is created at the same place, and with the same swimming direction than the parent; this position is of course already occupied, but it is replaced with the nearest free position found by an algorithm similar to the one used for moves.

### 2.2.2 Motion

In CALIFE, space is represented by a *Space* object, containing a two-dimensional grid of positions. The real space where the simulation takes place is mapped to this grid. A zooid at a position (X,Y) is displayed on the computer screen by sending a constant of type *Code* to the (X,Y) screen coordinates. In text mode, a zooid is displayed with a character; an attribute determines its foreground and background colors. In 80 x 25 text mode, *Codes* can be directly mapped to the PC video memory; *Codes* are then records featuring an ASCII character and a byte attribute. In graphic mode an individual could be represented with one or more pixels, depending on the resolution and the grid size. *Codes* are then implemented as integers corresponding to a pattern of bits corresponding to foreground and background values; they should be displayed using a procedure interfaced with a graphic library, or with an assembly routine addressing the video system. Each time an organism moves, its previous position is cleared and a *Code*, corresponding to the organism's identity and state (i.e. a *Live_Oozooid*, a *Chain_Cadaver*), is written to the new location.

The screen motions roughly mimic the displacements of real tunicates, which swim by pulsations of their entire cylindrical body. The artificial individuals move in one of 8 directions. At each loop cycle, each individual moves to the next available grid position. If this position is occupied, the swimming direction is reversed (a 180° change), as do real tunicates when they hit an obstacle. To avoid being stuck in corners, after 8 moves the direction is changed by 1/8. This has the additional advantage that, if there is no obstacle, the individual swims in large circles, like a real tunicate. A rather complicated algorithm is used to find the nearest available position when the next position is occupied by another individual, or on a screen edge. If the algorithm fails to find a position not occupied by a zooid, no move occurs; a free position usually becomes available at a later cycle (if after 10 cycles there is still no free position, the exception *Over_Population* is raised; this kills the task).

As individuals move concurrently and share a single display, there is of course a mutual exclusion problem. Accessing the screen through a controller task does not solve it completely, because when several hundred of individuals are moving, the controller becomes a bottleneck, and the program is eventually totally frozen. In a previous version of the software, the controller was suppressed. This had the slight inconvenience that, from time to time, two individuals occupy the same position; the second arrival does not search for a free position, but is displayed over the first. However, as each individual records in its *Position* field its true position, as soon as their paths diverge the positions of the two individuals are again distinct.

This problem is now solved by using an array of controller tasks, each in charge of a small portion of the grid. A zooid wanting to move determines its next potential location (X,Y) from its current position and direction. It then addresses a request to a *Space_CTRL* object. This object determines which controller in the array is in charge of the region containing (X,Y), and forwards the request to it. If the controller determines, in mutual exclusion, that the location (X,Y) is empty, it updates the grid in the *Space* object. If the (X,Y) location is occupied by another zooid, another controller request is made for a position at 180° from the current zooid direction. For births, especially when numerous oozooids arise from a chain, a more "compact" algorithm is used to find an empty position close to the parent: instead of exploring only the 8 directions around the requested position, the area is searched for consecutive positions in a spiral manner.

If every zooid moves or reproduces at fixed points in the internal loop of its task, this would result in quasi-synchronous moves of all the zooids, giving an unnatural jerky aspect to the simulation. Moreover, if all moves occur (quasi-)simultaneously, the controller tasks are likely to be overwhelmed when the zooids are numerous. To smooth the arrivals of controller requests, a very small random lag is introduced before each reproduction (sexual and asexual). This is done with a delay statement taking for expression a random function returning a duration between 0 and 200 milliseconds. This delay has the advantage of adding a supplementary synchronization point in each zooid task.

### 2.2.3 Feeding

Food is a spatialized resource, modeled in CALIFE by food objects. The body of a food object contains an (X,Y) array, whose locations correspond to those in the *Space* object array. Different kinds of food are represented by instantiations of a generic package, *Food*. This package makes visible a function, *Inquire_Food*, returning the amount of food present at a given location, and a procedure *Update_Food*, which, given a certain *Amount_Requested*, returns *Amount_Delivered*, updating the amount present at the given location. Feeding is done in mutual exclusion, when the requested destination in a move contains a non-null amount of food. Ingestion of food results in an increase of the reserves of a zooid (the amount of reserves is kept in a local variable). To help to visualize the positions containing food (which cannot be shown at the same time as the organisms), a green code is displayed when a zooid encounters food.

## 3. Error handling

When an artificial tunicate reproduces, it allocates for its child zooid the amount of memory needed for a record plus a task. Later, the memory allocated for the task is automatically freed by the runtime executive when the zooid dies (i.e. when the task main loop is exited). The memory reserved for the record could be recovered, using *Unchecked_Deallocation*, but the linked list of individual records would then be broken. An increasing amount of storage is thus allocated during the course of the program. An interesting situation occurs when almost all memory is allocated, and the next task which reproduces raises *Storage_Error*. This situation is not likely to happen, because growth is usually regulated by the amount of food. However, provision for it should be made.

The only sensible action in these conditions is to write a message and to output the results of the simulation. But the program may only proceed if all tasks are terminated, and there are usually several hundred tasks still active when one raises *Storage_Error*. These tasks are in various states, some being engaged in reproduction, others in rendezvous with their offspring. Moreover, *Tasking_Error* is raised at the point of call if the callee becomes abnormal.

The solution to these problems was not easy to find; the debugger could not help in presence of the long linked chain of allocated tasks. The first task encountering *Storage_Error* should signal the other "normal" tasks to terminate. Local exception handlers (in blocks surrounding the calls to the *Sexual_Reproduction* and *Asexual_Multiplication* procedures) trap *Storage_Error* and *Tasking_Error*, and raise an exception *Over_Population*, which is handled in the task body, outside the main loop. Exceptions in tasks are not propagated, but the handler calls a procedure *Stop_Simulation* before the task is killed. *Stop_Simulation* belongs to the visible part of a *Simulation_Clock* package. This package provides a function *Simulation_Is_Over*, which is polled by the zooids before engaging in every event in their task main loop. In the normal course of the program, this function returns *True* when the simulation duration is elapsed. It also returns *True* if a zooid has called *Stop_Simulation*.

But this is only half the solution. Another tricky situation may arise if a zooid has spawned a child, sent a call to its *Start* entry, and died because of overcrowding before this call is accepted. If a task is "waiting at accept", the program never terminates. The solution is to make the "accept *Start*" a timed accept, and to raise *Over_Population* if the call is not accepted within a short interval. When there are hundreds of tasks active, it was found that this interval should be about twice the pace of the task main loop (that is, 2.0 seconds).

## 4. Output

The visual outcome of the software is suggestive: starting from a single individual, the tunicate population develops slowly on the screen. When most of the food is consumed, the population decreases: reproduction requires a certain amount of reserves, which cannot be reconstructed by feeding. Reproduction costs also to the reserves of the individuals, so that the exponential growth of the population cannot be sustained very long.

Besides the visual impact, it is necessary to get numerical and graphic data from the simulation.It would be impracticable to make the zooids write their state changes to a log file, with timestamps. Furthermore, we do not want to let the artifacts of the simulation interfere appreciably with the behavior of the zooids. This is why the linked chain of the zooid records was kept until the end of the program. When the simulation is finished, it is possible to start from the pointer to the first individual, and to visit every zooid *Zooid_Pattern* record. In this record, the fields *Position*, *Birth_Date*, *Disappearance_Date*, etc., were updated during the existence of the zooid. Simple counts give the number of zooids created during a given interval of time, the number of them still alive at the end, and all the statistics needed.

The chain of records starting from the first oozooid forms an unbounded arbitrary tree (in the terminology of Booch [2]). Visiting all nodes is done recursively using a preorder traversal. Each *Chain* node is the root of *Number_Of_Aggregates* oozooid subtrees. Each *Oozooid* node is in turn the root of *Number_Of_Generations* subtrees. During the traversal, counts are recorded in a file (in ASCII format). This is done only when the simulation is finished, so there is no time impediment which may slow down the simulation itself. To get to the first pointer, a *Track_Zooids* procedure is added to the visible part of the Tunicates package, besides the *Create* procedure.

## 5. Computer implementation and performances

The CALIFE program was developed with an Alsys 386-DOS Ada compiler and environment on a 80386 (20 MHz) micro-computer. The version 4.50 of the software represents about 4,500 lines of Ada (including 1,500 lines of comments), not counting the utility libraries (about 3,000 lines of code).

If not regulated, the Tunicate population derived from one initial oozooid leads in a few minutes to about 1,500 active tasks before *Storage_Error* is raised (with 6 Mbytes of RAM). When food availability regulates reproduction, the peak of the Tunicate bloom occurs when about 300 zooids are present. This number is smoothly handled on a PC, in text mode, with a 3 x 10 array of controller tasks in charge of the 25 x 80 space grid. However, the exponential growth of the zooids should be limited by allowing only a few individuals per chain, say 5, instead of their real number in the sea (which is around 100 for the most common species, *Salpa fusiformis*).

## 6. Discussion

With parameters set to reflect the relative durations of real Tunicates life cycle phases, a bloom simulation with CALIFE takes about 3 mn. In this lapse of time, several generations of zooids develop and die. A one-second pace is used for the task main loop, so that moves occur every second. In text mode, the 25 x 80 grid of the PC screen is too small to allow a seemingly unlimited travel of the zooids. However, a 200 x 200 grid or higher would be sufficient to let the population develop with no or few encounters with the screen edges. This could be accomplished in graphic mode, in which even the 480 x 640 size of the standard VGA video system provides enough space. The abstractions used in the software for specifying the *Space* object and to display a *Code* simplify the transition from text mode to graphic mode. This remains to be done in a forthcoming version.

Could the Tunicates package be made generic to fit different kinds of organisms ? At first the answer seems positive: the record+task structure may apply to many species. However, designing a generic organism with a parameter indicating whether it has an asexual reproduction phase or not would be at least awkward. Moreover, the complicated reproduction of Tunicates is not a unique event in the biological world. Many reproductive modalities exist: parthenogenesis (cyclic or not), hermaphroditism, scissiparity, etc. A generic *Organism* package would have to be thoughtfully crafted. Furthermore, Ada 9X may provide still better solutions: a tagged private type *Organism* could have a few "basic" fields, with the specific ones programmed by extension.

The scheduling of Ada tasks is inherently nondeterministic. This is well known by Ada programmers, but difficult to admit by ecological modelers only trained to FORTRAN or Pascal. When confronted with the CALIFE simulations, which never give exactly the same outcome, they usually attribute this to the random delay introduced before reproductions (see section 2.2.2). A version with delay 0.0 (retaining the synchronization point) helps to illustrate this issue.

For biologists interested in Alife techniques, some analogies may be drawn. The organism's living matter is built from the heap memory space. Type definitions contain the information necessary to determine its structure; they play the role of the DNA sequences of nucleotides. The Ada *new* allocator corresponds to embryogenesis. The *Start* rendezvous with the offspring is the birth process. The main internal loop in the *Vital_Process* task is the equivalent of the "biological clock". These analogies are helpful to map the "solution space" to the "problem space". The resulting Ada code more naturally fits the biological or ecological problem.

## 7. Conclusion

Ada tasks are a natural vehicle for representing autonomous agents in software simulations. Alife techniques could benefit from the potential of the thoroughly studied model of Ada tasking. A package spawning a population of self-reproducing organisms may be a building block of an object-oriented model of ecosystem. Work is under way to add parasites of Tunicates to the CALIFE model. This should introduce interesting interactions, where parasites rendezvous with their hosts.

## Acknowledgements

This research was carried out in the URA 716 "Écologie du plancton marin" (Centre National de la Recherche Scientifique: Institut National des Sciences de l'Univers). The record+task structure representation originated from a program written by M. Koenig to model vehicles circulating on a bridge. The author thanks J. Dolan for helpful comments.

## References

[1] Alldredge, A. L. and Madin, L. P., 1982. Pelagic tunicates: unique herbivores in the marine plankton. *Bioscience*, 32 (8): 655-663.

[2] Booch, G., 1987. *Software components with Ada.* Benjamin/Cummings Publ. Co., Inc., Menlo Park, CA, 635 pp.

[3] Breckling, B. and Müller, F., 1994. Current trends in ecological modelling and the 8th ISEM Conference on the state-of-the-art. *Ecol. Modelling*, 75/76: 667-675.

[4] Delatte, B., Heitz, M. and Muller, J.F., 1993. *HOOD Reference Manual 3.1.* Masson, Paris and Prentice Hall, London, 175 pp.

[5] Laval, Ph., 1995. Hierarchical object-oriented design of a concurrent, individual-based, model of a pelagic Tunicate bloom. *Ecol. Modelling*, in press.

[6] Madhupratap, M., Devassy, V. P., Sreekumaran Nair, S. R. and Rao, T. S. S., 1980. Swarming of pelagic tunicates associated with phytoplankton bloom in the Bay of Bengal. *Indian J. mar. Sci.*, 9 (1): 69-71.

[7] Murray, A.G., 1990. Ada tasking as a tool for ecological modelling. *ACM Ada Letters*, 10 (7): 85-90.

[8] Taylor, C. and Jefferson, D., 1994. Artificial Life as a tool for biological inquiry. *Artificial Life*, 1 (1/2): 1-13.