Hydroinformatics' 98, Babovic & Larsen (eds) © 1998 Balkema, Rotterdam, ISBN 90 5410 983 1

The potential of proficient bioinformatics for aquatic ecology

P.Laval
Observatoire Océanologique, Station Zoologique, Villefranche-sur-Mer, France

ABSTRACT: Artificial life techniques are most of the time used to study evolutionary processes. In these programs, simple creatures reproduce with mutations and cross over, and undergo selective pressure. Because of the simplicity of the artificial entities, it is possible to put hundreds of them in interaction, and to run the program over thousands of time-steps. Many evolutionary hypotheses are thus accessible to experimentation, with a programming effort within the reach of a biologist. However, it is possible, with some software engineering (SE) knowledge, to go beyond simple programming and to address ecological problems with larger software constructions. SE methods help in building a sound software structure for articulating the major entities of the domain, with software agents operating in a spatially-explicit environment. SE methods are also useful for managing the concurrency inherent in complex systems and for keeping the 'solution space' close to the 'domain space', with computer artefacts reduced to the minimum. Ada is the language of choice to meet these requirements (provided of course it is wisely used). SE principles are at the heart of Ada. Moreover, the latest standardized form of Ada, Ada 95, provides safe object-orientation, and improved real-time concurrency. Real-time concurrency is important to simulate autonomous organism interactions. Taking account of the parallel interleaving of processes is difficult because it questions our cartesian view of causes and effects. Mathematical modeling postulates a closed universe where processes are chained in deterministic ways. The real world is open, and process combinations can be unnumerable. Informatics provides tools to faithfully represent this real world, at the expense of provability. Mathematics lead to provable deductions, but they only apply to particular situations. In building over several years what I called a virtual mesocosm for simulating fragile marine planktonic organisms, I had many opportunities to verify the importance of a good design based on sound SE. A satisfactory design, from both informatics and ecological points of view, is usually not attained from the start. It is imperative to stay close to the biology; without meaningful biological specification, it is all too easy to err toward computer artefacts, using nice programming subtleties but unrealistic modeling assumptions.

1 INTRODUCTION

Bioinformatics is a recent discipline at the intersection between informatics and biology. The biology here is rather molecular biology. But informatics as a discipline (not merely the use of computers) may transform other biological domains, such as ecology or evolution (Levin et al. 1997). In this marriage, informatics brings its load of technical culture, whereas biology is more or less passively handled.

As a biologist/ecologist heavily involved with informatics for over 25 years, I will try to explain how I experienced this interpenetration. Informatics is pervaded with technicality, part of it being more the result of marketing pressure than of thoughtful thinking. Another detrimental aspect is technique for the sake of technique. On the ecological modeling side, there is often a lack of appreciation of the nature of complex systems. In a legitimate search for elegant simplicity, some models sometimes resemble mathematical recreations.

I will consider, on the informatics side, programming practices, object orientation, and concurrency. On the modeling side, open systems, nondeterminism, simulations, artificial life and intelligent agents. In conclusion, I will stress the need to better integrate biological concepts in informatics terms.

2 INFORMATICS

2.1 *Programming practices*

Writing application programs is different from writing the components of an operating system. For applications, there is no necessity to stay close to the machine level. Instead, the programmer should build abstractions of the domain. Abstracting requires a large vocabulary, encompassing all aspects of the problem. Philosophical questions cannot be easily discussed by a child, and the same is true in programming: with a simple language it is not possible to express more than ground-level evidences.

However, many programmers do like simple languages: these languages allow them to do tricky things which make the fun of programming (they think it is fun because they did not have the opportunity to do program maintenance). Unfortunately, a biologist or ecologist has little time to learn a large language, and everybody around him or her says this is unnecessary. They think so because they only see programming as a way to code algorithms to make calculations.

Programming may be more than that. It should be rather viewed as a means of modeling. It is only by composing high abstractions that creative programming may emerge. But the way is not an easy one. An unnecessary technicality is most often confused with complexity. This is a natural characteristic of the human mind: except for geniuses, simplicity can only be attained by hard work.

Informatics does not flow naturally into simplicity. Moreover, the technical character of the field leads easily to the perusal of jargon (see any computing magazines); in programs, cryptic names and undocumented tricks (often associated with illiteracy in their own language) give their authors a false sense of scientisticity.

Some ecologists with a weak background in informatics may be tempted to compensate this missing skill by collaborating with engineers. However, students freshly from a technology institute have their own technical culture. They were educated to get a task done, with no insistency to polish the means used: if the program works correctly, so far, so good. This is understandable in an industrial context, where there are time and money constraints. Davis (1992) has written a vivid analysis of the different approaches of academic researchers and engineers faced to software design problems.

However, a program may "work" and give correct results in many different ways. I was confronted with this evidence when rewriting pieces of a large software several times during the past years (because I wanted to explore a different architecture, or because I wanted to use the possibilities of a new hardware, or, recently, when a major language revision — the change from Ada 83 to Ada 95 — opened new avenues).

When, after several years of satisfactory operation, a software system is somewhat dismantled in order to be improved, one is forced to examine every component from a different point of view: is this software object really necessary?, what was the very reason for the presence of another one?, and so on. A feature which seemed wisely coded (a feeling reinforced by the fact that "it worked"), suddenly appears questionable. Of course even a slight change in a consistent large system has rippling effects. When, after hard work, these modifications are consolidated, the new solution is sometimes obviously simpler than the old one. This a good indication that is better. This new design would not have been disclosed if the system had not been touched. It is probable that many software designs carry unnecessary contortions, whose hidden costs are non obvious logic, difficult maintenance, and hindrance of further progresses.

It should be fair to stress that what I would call "software models" in ecology should not be compared with industrial systems. They are rather working prototypes to help in understanding a research problem. When they work, they illustrate a possible way the system may function; if they don't work well, they help to find where improvements are needed. I insist on the clarity and elegance of the software because in their absence the system would be much difficult to understand.

2.2 *Object-orientation*

The object paradigm has, rightly, replaced functional design. There is a definitive advantage to program with clean abstractions, each having a public well-delimited interface and a modifiable private implementation. Problems arise when the programmer needs, at a higher level, to consider objects which are not abstractions of real objects but conceptual objects. Here the position of the ecologist is less comfortable than that of the engineer. An engineer devising, for example, an airplane control system, may obtain any information he or she needs to model the system. The difficult work amounts to cleverly arrange this information.

In contrast, the ecologist has only at hand an incomplete representation of the system: some large-scale knowledge of the physical environment, some laboratory experiments or field observations. The rest is speculation, ranging from more or less established theories to tentative explanations. Further problems are the representativity of the sampling, the validity of the laboratory experiments, problems of scale (in space and time), etc.

Bioinformatics should implement this knowledge, taking care not to introduce too much additional biases. Object-oriented features offer a powerful way to model real-world relationships. Here informatics (more precisely software engineering) is useful because it provides methodologies for system analysis. Simulation techniques may also be used to observe the behavior of the model.

For hydrological simulations, an attempt has been made by David (1997) to integrate legacy FORTRAN code in an object-oriented framework. This pragmatic approach is still at a too low-level of abstraction, because it glues together modules which were not crafted from the start with an object-orientation in mind.

Like any powerful technique, informatics may be easily misused. Object-oriented languages introduced inheritance, which was first viewed as a handy means to reuse code: just write features distinguishing a new object from an almost similar one. This useful concept looked attractive, particularly to the novice programmers (those believing that is more "efficient" to write a=a++ instead of Undigested_Preys = Undigested_Preys + 1). In ecological modeling it is not uncommon to see such things as Zooplankton inheriting from Phytoplankton (this is inheritance at a too coarse level, an indication that the Phytoplankton object has very few properties). Moreover, inheritance can easily be confused, by non biologists, with evolutionary descent; this may only be justified in special cases. As Casais (1998) wrote (about re-engineering legacy code): "Software designers rely on powerful object-oriented mechanisms, especially inheritance, to bypass modeling weaknesses or avoid a rigorous process when extending object-oriented systems". Inheritance, wisely used, can nevertheless be a useful way of augmenting the diversity of a system without radically changing its structure.

The real difficulty is, however, to compose a system with meaningful objects, having sound relationships. Parsons & Wand (1997) give a comprehensive view of analysis and design. They point out that objects can be both viewed as units which model concepts from the domain, and implementation units for the practical realization of the software. Focusing too early on the implementation aspect can be detrimental to the design of the system. This danger can only be avoided if it is the ecologist which drives the design. He or she should be able to strongly discuss with the computer expert to delineate objects and relationships which make sense for the problem.

It is my experience that the feedback from software engineering methods used in system design can be very rewarding. Design methods add guidance on what is possible or not. In turn, the ecologist should decide if a possible relationship or object composition permitted by the rules, or the compiler, is compatible with the knowledge at hand. Sometimes this interrogation leads to a new insight on the problem. This is the very essence of modeling.

A short example may be in order. Implementing host-parasite interactions in as software where hosts and parasites are autonomous concurrent artificial creatures (Laval 1997) leads to interesting bioinformatics questions. How will the parasite (a crustacean) detect its host (a gelatinous filter-feeder)? How will it feed on it? In an artificial world made of data structures, pointers, and protected objects (a concurrent high-level construct available in Ada 95), it is necessary that algorithms and data structures closely reflect the biological knowledge. A chain of pointers to records containing characteristic data may thus represent a chemical track left by a host in space. The way to access these data is solved in computer terms, but it should also closely mimic how the parasite will find its host.

2.3 *Concurrency*

Real-world objects behave concurrently rather than sequentially. However, computer languages allowing us to deal with concurrency developed slowly, and practitioners readily discovered that concurrent programming is much more difficult than sequential programming. Even if each component object executes a short and correct algorithm, they interact in combination. It is often not practicable to test every possible path. Moreover, the time now intervenes, so that the order in which the objects interact is relevant, often within milliseconds.

Do not confuse concurrency with parallelism. Parallelism is a way to improve computation time on a multiprocessor (see for example Foster 1995). Concurrency refers to the execution of logical concurrent threads of control. Even on a monoprocessor (for example on a personal computer), these flows of control may be imperceptibly interleaved -- a kind of time-sharing, giving the impression that each object executes on its own processor (in this case there is no speed increase, but the logic is much clearer).

The importance of concurrency in modeling has been underestimated. Mathematical modelers, for which programming essentially consists to translate equations in FORTRAN-like languages, are foreign to this concept. The mathematics of concurrency (mainly temporal logic, Petri nets) are precious for solving small examples, but becomes unmanageable for most real-world problems.

The engineering domain has always been closely concerned with safety and error recovery. The expertise and lessons engineers learned from real-time control of critical appliances (such as airplanes,

rockets, nuclear power plants) should not be considered as largely irrelevant by ecologists. Of course, in a model written for understanding an ecosystem there is no threat for human lives, no timing constraints within microseconds. Nevertheless, a computer language with built-in hard real-time concurrency, such as Ada 95, is a blessing for ecologists. It gives them a well-proven modeling vocabulary apt to deal with the dynamic scenarios of organism interactions.

Modeling a domain with concurrent objects conduces to comparable difficulties, but at a higher level, than when searching for good objects. Awad & Ziegler (1997b) point out that a system is made of processes, which are executed by objects. Focusing too early on the objects alone may produce a too low-level view of the system. Awad & Ziegler (1997a,b) show how processes and objects can be smoothly integrated during the design.

3 ECOLOGY AND MATHEMATICS

3.1 *Open systems*

Mathematics is the royal avenue to deal with closed systems. The mathematical approach is very satisfactory because with its help, deductions completely consistent with a few initial propositions can be proved. Wegner (1997) has, in an illuminating article, clearly pointed out the problem with mathematics in modeling. Briefly resumed, its arguments are as follows. The mode of operation of mathematics is algorithmic: if (this proposition) is true, then (such consequence); else (another one). Any sequential algorithm can be simulated with a Turing machine, in which a tape coded with instructions is read stepwise; the next instruction executed results from the last one read.

In a Turing machine, a proposition cannot be both true and false, because the system is closed. In an open system, however, an external event can modify the state of the tape between the time is was read and the execution of what was read. Open systems are thus immensely richer in possible outcomes than closed ones. But this depends on predictability, because in most cases, the number of possible states cannot be enumerated (this is akin to the Gödel theorem of incompleteness of arithmetic over the integers). The limitations of a mathematical representation of the real world have been discussed by Casti (1996). Their repercussions on software engineering practices are commented by Zelkowitz (1995).

We are thus faced with the choice between using mathematics, and being able to solve only special cases, and using informatics, which can more closely simulate the concurrent interactions of the real world, but with some indeterminacy. Mathematical modelers may argue that these special cases which can be completely solved may give interesting clues on the functioning of the real system. This is certainly true in some cases. The question is how far this may apply, and if the richness of interactions in a large ecosystem may not produce emergent phenomenons not deducible from the premises.

3.2 *Nondeterminism and simulation*

Mathematicians confronted with the natural variability of natural systems have developed techniques to master this irritating lack of rigor. They postulated that variable phenomena obey mathematical equations containing an error term, the amplitude of which can be estimated if its statistical distribution is known or may be inferred to some degree.

It is very difficult to disprove this assertion. To establish with some confidence a slight departure from a statistical distribution a huge amount of data is usually required. Physicists and economists have recently shown that some complex phenomenons can be better described with a non-gaussian stable Lévy distribution. This distribution has the interesting property of having an infinite variance, so that statistical theorems cannot apply (Bouchaud & Walter 1996, Mandelbrot 1996).

The presence of self-organized criticality (Bak 1996) in many complex phenomena makes impossible to precisely predict when the "avalanches" which characterize them will be triggered. With a good software model (a software putting in action concurrent processes and objects), it should be possible to observe some likely outcomes, and to assign them probabilities. But it seems impossible to be exhaustive, so that it is not practical to completely rule out catastrophic situations where rare independent processes combine unexpectedly.

In simulation softwares, most efforts are devoted to provide statistically unbiased behaviors of agents. This is understandable and highly recommendable when there is an underlying statistical hypothesis that the simulation should test. If the statistical distribution is unknown or not testable, the aim of a simulation should better be turned toward understanding how a system is structured and how it works. A simulation software of this kind is more an animation of a system design than a statistical tool. It has nevertheless a great potential explaining power. It is actually a modeling tool.

The prevalent paradigm in simulation software is event-driven simulation. This is adequate when objects trigger events which are managed by some central controller. Concurrent messages are

dispatched by a scheduler which orders them and resolves ties. This pseudodeterministic outcome is convenient to apply statistical tests. But a more realistic modeling should not resort to a central controller: entities should be allowed to freely interact.

3.3 *Agents and artificial life*

Bioinformatics is a way to express biological concepts with computer techniques and theories. The effervescent field of artificial life is a good illustration of this notion. I would argue that even in what is called "weak artificial life", informatics may play an important role.

Programming numerous rudimentary creatures with a "chromosome" (a short string of symbols) is not an unsurmountable task for a student in biology with some computer skills. Moreover, it is very attractive and thought provoking. Very interesting and profound results are obtained when these programs are run over thousands or millions of generations.

Designing and programming a small ecosystem populated with elaborated artificial creatures requires both a good knowledge in biology/ecology and a greater proficiency in programming and software engineering. Current researches in agent and cognitive science are mainly made by computer scientists. Their informatics part is well developed. In contrast, the translation of biological concepts into informatics terms is not so well achieved.

A (so-called intelligent) agent is an autonomous unit with an internal state, sensors, and a representation of its external environment. An agent may use a strategy to attain a goal. It is possible to simulate agents in economy because as human beings we know the rules (or at least some rules) of the game. But is it possible to affirm that we can craft an intelligent artificial fish or planktonic crustacean? We know very little of their sensory capacities (we cannot know their environment with our sensorial equipment), that is, how they see or feel the world. In our present state of knowledge, bioinformatics may rather help in disclosing the consequences of hypotheses: for example what would happen if a predator could detect its prey at such distance?

4 CONCLUDING REMARKS

In informatics a compiler may provide either a "thin" or a "thick" binding of a language to an external library, according to the degree of integration of the library. Similarly, an ecologist may have a thin or thick involvement with informatics. The former is the usual case, because ecologists have not enough time to assimilate concepts remote from their research expertise. I propose here that ecologists with a good background in informatics try another point of view, where informatics plays the role exerted until now by mathematics in ecological modeling.

In this perspective, bioinformaticians may work toward the elaboration of a library of reusable and customizable modules coding elementary processes from ecology and biology: reproduction (with many modalities: sexual, asexual, parthenogenetic, etc.), feeding (autotrophy, heterotrophy), growth, predation, parasitism, and so on. Object-oriented programming will give the means to refine and adapt these basic bricks. The difficult part lies not in informatics but in the specification of a consistent hierarchical arrangement of the different concepts. Then the ecologist needing to simulate the growth of a population will be in the same position than the mathematician wanting to extract the latent roots of a matrix.

Informatics is still in a state of effervescent development. There will be much work before such libraries attain a general agreement. Trying to translate concepts from ecology into informatics constructs may at the end be beneficial to ecology itself.

REFERENCES

Awad, M. & J. Ziegler 1997a. A practical approach to object-oriented state modeling. *Software Practice Exp.* 27(3):311-328.

Awad, M. & J. Ziegler 1997b. A practical approach to the design of concurrency in object-oriented systems. *Software Practice Exp.* 27(9): 1013-1034.

Bak, P. 1996. How nature works. New York: Springer-Verlag (Copernicus)

Bouchaud, J.P. & C. Walter 1996. Les marchés aléatoires. *Pour la Science (french ed. of Scient. Amer.), Spec. issue "Le Hasard"* Apr. 1996:92-95.

Casais, E. 1998. Re-engineering object-oriented legacy systems. *J. Object-Oriented Progr.* 10(8):45-52.

Casti, J.L. 1996. Confronting science's logical limits. *Scientific American*, 275(4):78-81.

David, O. 1997. A kernel approach for interactive-oriented model construction in Java. *Concurrency: Practice and Experience* 9(11):1319-1326.

Davis, A.M. 1992. Why industry often says 'no thanks' to research. IEEE Software, 9(7):97-99.

Foster, I.T. 1995. *Designing and building parallel programs.* Reading: Addison-Wesley.

Laval, P. 1997. A virtual mesocosm with artificial salps for exploring the conditions of swarm development in the pelagic tunicate *Salpa fusiformis. Mar. Ecol. Progr. Ser.* 154(1):1-16.

Levin, S.A., B. Grenfell, A. Hasting & A.S. Perelson 1997. Mathematical and computational challenges in population biology and ecosystem science. *Science* 275(5298):334-343.

Mandelbrot, B. 1996. Du hasard bénin au hasard sauvage. *Pour la Science (french ed. of Scient. Amer.), Spec. issue "Le Hasard"* Apr. 1996:12-17.

Parsons, J. & Y. Wand 1997.Using objects for systems analysis. *Comm. ACM* 40(12):104-110.

Wegner, P. 1997. Why interaction is more powerful than algorithms. *Comm. ACM* 40(5):80-91.

Zelkowitz, M.V. 1995. Algebra and models (and reality). *ACM SIGSOFT Softw. Engin. Notes* 20(2): 55-57.