ELSEVIER

Ecological Modelling 157 (2002) 23–41

ECOLOGICAL MODELLING

www.elsevier.com/locate/ecolmodel

A multi-agents architecture to enhance end-user individual-based modelling

Vincent Ginot [a,*], Christophe Le Page [b], Sami Souissi [c]

[a] *INRA, Unité de Biométrie, Domaine St. Paul, 84914 Avignon Cedex 9, France*
[b] *CIRAD-Land and Resources Programme TA 60/15, 73 av. J.F. Breton, 34398 Montpellier Cedex 5, France*
[c] *Ecosystem Complexity Research Group, Station Marine de Wimereux, Université des Sciences et Technologies de Lille 1, CNRS-UPRES A 8013 ELICO, BP 80, 62930 Wimereux, France*

Received 27 July 2001; received in revised form 19 April 2002; accepted 28 May 2002

Abstract

The increasing importance of individual-based modelling (IBM) in population dynamics has led to the greater availability of tools designed to facilitate their creation and use. Yet, these tools are either too general, requiring the extensive knowledge of a computer language, or conversely restricted to very specific applications. Hence, they are of little help to non-computer expert ecologists. In order to build IBM's without hard coding them nor restricting their scope too much, we suggest a component programming, assuming that each elementary task that forms the behaviour of an individual often follows the same path: an individual must locate and select information in order for it to be processed, then he must update his state, the state of other individuals, or the state of the rest of the 'world'. This sequence is well suited to translation into elementary computerised components, that we call primitives. Conversely, task building will involve stringing out well-chosen primitives and setting their parameter values or mathematical formulae. In order to restrict the number of primitives and to simplify their use, 'information' must be carried through well defined structures. We suggest the use of the multi-agents system paradigm (MAS) which originates from the distributed artificial intelligence and defines agents as autonomous objects that perceive and react to their environment. If one assumes that a model can be described entirely with the help of agents, then primitives only handle agents, agent state or history. This greatly simplifies their conception and enhances their flexibility. Indeed, only 25 primitives, split into six groups (locate, select, translate, compute, end, and workflow control) proved to be sufficient to build complex IBM's or cellular automata drawn from literature. Furthermore, such a primitive-based multi-agents architecture is very flexible and facilitates all the steps of the modelling process, in particular the simulation engine (agents call and synchronisation), the results analysis, and the simulation experiments. Component programming may also facilitate the design of a domain specific language in which these models could be written and exported to other simulation platforms.

Keywords: Individual-based models (IBM); Multi-agents systems (MAS); Population dynamics; Component programming; Modelling tools

* Corresponding author. Tel.: +33-4-32-722185; fax: +33-4-32-722182
E-mail addresses: ginot@avignon.inra.fr (V. Ginot), lepage@cirad.fr (C. Le Page), sami.souissi@univ-lille1.fr (S. Souissi).

0304-3800/02/$ - see front matter
PII: S0304-3800(02)00211-9

1. Introduction

In population biology, individual-based modelling (IBM) and artificial life concepts date quite far back (Conrad and Patten, 1970; Kaiser, 1979). However, the increase in computer power and object language has really made them popular only in the last 10 years. Particular examples are Taylor et al. (1989) using the RAM simulator, Hogeweg (1989) with Mirror, but most importantly DeAngelis and Gross (1992) who edited the proceedings of the Symposium on 'individual-based modelling' at Knoxville, Tennessee, in 1990. Since then, applications have increased in number and expanded into many fields. The readers may refer to Grimm (1999) who counts more than 500 applications, and analyses 50 of them with respect to animal population dynamics. If one concentrates on fish population dynamics, which represents 40% of the models studied by Grimm and from which this present project originally evolved from, then a number of models or simulators deserve mention, such as *SSEM* (Sekine et al., 1991), *SimDelta* (Bousquet et al., 1993), *Hobo* (Lhotka, 1994), *EcoWin* (Ferreira, 1995), *SeaLab* (Le Page and Cury, 1996), Van Winkle et al. (1998), or *Osmose* (Shin and Cury, 2001).

On the computer side, IBM relies increasingly upon object programming. For all that follows, an attempt shall be made to maintain a wide vision of the individual. It shall of course remain the individual in the classical sense of the term such as a fish or a cow. But it may also prove useful to break up an individual into the different stages of its life cycle (juvenile vs. adult for example), or into the different 'behavioural roles' that it may play (as breeder and non-breeder). Contrarily, it may also be useful to assimilate an 'individual' to a group of individuals, which may be considered as identical, such as a fish school. The point here, rather than focussing on the very definition of an individual, is that they are numerous, theoretically different, and generally in interaction. This is why it is worthwhile to go beyond the object notion, and to assimilate the individual into the computer terminology of 'agent'. A detailed description of agents and multi-agents systems (MAS) can be found in Ferber (1999) or Weiss (1999) for example, but for the purposes of this paper, one can just consider that an agent is essentially an autonomous object, i.e. an object that controls its own behaviour. When compared to classical modelling methods, which usually consist in translating the conceptual model of the population dynamics into a numerical form that can be solved by a computer (often with differential equations), the individual-based approach, especially when treated with a MAS, offers substantial advances for ecological modelling. These advances are well described by Judson (1994) or Lomnicki (1999) for example, but three main points are highlighted below that are deemed fundamental in the field of population dynamics.

1) Recognition of the individual behaviour, both for individuals and for space. The individual-based approach allows for the inclusion of the diversity in behaviour of the individuals of a same population. A simulation of an (hypothetical) mean behaviour of the population is no longer made, but the population dynamics rise up from the very trajectory of each individual. At the same time, with a MAS and agent-cells, it is easier to describe a spatially complex and fluctuating environment in which individuals are inserted.
2) Unity of thought between the biologist and the computer scientist. It is probably the only modelling approach where the computer program corresponds to the biological model. The biologist and the computer scientist find themselves centred on an artificial fish or bird for example. There are almost no more problems associated with the translation of a functional concept on the one side (the idea that the ecologist has of his animal) and the computed reality encoded by the computer programmer on the other side.
3) With a MAS, the computer script is no longer centralised, but distributed in a multitude of autonomous agents. One can add, eliminate or modify agents without affecting the rest of the model. Therefore, creating, testing and modifying a model becomes a much faster and easier process when compared to more conventional techniques.

But the creation and execution of an IBM is a delicate operation that requires substantial computational investment, as emphasised early by Hogeweg and Hesper (1990). And even more so for a MAS. It is therefore beneficial to use a modelling workbench. Although these workbenches remain rare, this area is presently blooming. Following a classification close to that of Lorek and Sonnenschein (1999), these tools may be split into three main categories, and a few significant examples are given below.

1.1. Generic frameworks

They rely on a programming language and help computer scientists to write MAS, whatever their field of application. The best known is probably Swarm (Minar et al. 1996), already commonly used by the research community in artificial life. It is based upon Objective-C and Java. The Geamas platform (Marcenac, 1997), also under Java, originates from Geophysics and offers the originality of a layered system of agents (notion of groups and hierarchical relations between agents) and the processes of aggregation and disaggregation between layers. The Mad Kit platform (Gutknecht and Ferber, 1997), also in Java, relies on the notion of role (abstract behaviour, perceivable by and on interaction with other roles) before that of the agent. It is well adapted for the writing of models distributed over many computers working across the web. Starlogo (Resnick, 1996) must also be mentioned, being more accessible to the non-computer expert, as it is based upon a macro-language. It is mostly a didactic tool, permitting the piloting of agents over spatial grids.

1.2. Ecosystem oriented frameworks

They propose utilities that are more specifically dedicated to the simulation or management of ecosystems. They are still highly generic, being also based on programming languages. Ecosim (Lorek and Sonnenschein, 1998), is based on a C^{++} objects library and a simulation engine based on discrete events. Cormas (Bousquet et al., 1998), using Smalltalk, is dedicated to the modelling of interactions between human society and renewable resources. Its most original feature is in its integration of evolved spatial support, including multiple scales and directly linked with geographical information systems (GIS).

1.3. Dedicated platforms or 'simulators'

These are much more numerous and have precise applications. Practically speaking, simulators are structurally-fixed models, but they are equipped with a user interface through which the user sets the values of the parameters and analyses simulations. For example Manta (Drogoul et al., 1992) is involved with the emergence of task specialisation in insect (ant) societies. SugarScape (Epstein and Axtell, 1996) looks at the way in which agents organise one another to exploit two resources, the one abundant and cheap, sugar, and the other rare and expensive, spice. Formosaic (Liu and Ashton, 1998) deals with fragmented forestry dynamics. Bacsim (Kreft et al., 1998) models microbiological dynamics. Osmose (Shin and Cury, 2001) generates virtual fish species interacting via predation.

But as recognised by Lorek and Sonnenschein (1999), simulators are too specific to answer new questions, and frameworks remain inaccessible to the non-computer programmer because of there coding language. Therefore, there is often a distance (if not a gap) between the modeller and the end-user, which may have heavy negative consequences: how many excellent models have been quickly forgotten due to the lack of evolution after being supplied to their end-users? And one of Grimm's (1999) principal criticisms concerning the use of IBM's is that the importance of the model structure on results is dramatically insufficiently explored, and that one rushes towards an ever growing complexity without questioning whether this complexity is really necessary. But how could the researcher address this issue if the model is a black box he has not build by himself and that he cannot modify?

Now, even if it means restricting the field of application a little bit, it seems possible to design a tool for IBM creation and use which is accessible

to biologists. In this view, the task of the computer scientist is no longer to offer a model or in the best case ready made agents, i.e. a simulator, but far more elementary components from which a user may build his agents and his own simulation environment. Lorek and Sonnenschein (1999) paved the path with Wesp-tool dedicated to metapopulation dynamics. Based upon the concept of roles and on the formalisation of the transitions between them, it allows the biologist to describe graphically the life cycles of the individuals without the need of a programming language. The approach presented here is rather different and allows for a much greater range of models to be written. The idea is that the biologist is more interested in what the agent does than in what it actually is. Hence, the encoding effort shall no longer be concentrated at the level of the agent or that of its role, but at the level of the different tasks that the agent has to execute, or at the even more elementary level, that we call primitives, at which these tasks may be broken down.

But for this braking down to take place efficiently without falling back into the complexity of a programming language, primitives must remain simple in use and small in number. It is thus the entire computational architecture that one must take into account with this concept. As it would be difficult to understand how primitives could work without having a vision of this architecture, we shall briefly present the principle of primitives (Section 2), and then focus on the computational choices: agents definition and role (Sections 3.1 and 3.2), agent architecture (Section 3.3), platform general architecture and results memorisation (Section 3.4), time management and agent synchronisation (Section 3.5). We shall then go deeper on primitives (Section 4), and test this concept on three classical models drawn from literature (Section 5). In conclusion (Section 6), we emphasise that, at the current stage, such a concept is mainly designed for 'rapid prototyping' but might be too limited to implement and analyse complex and multidisciplinary IBMs. But conversely, it may facilitate the design of a domain specific language in which IBMs could be written.

2. Breaking down a behaviour into tasks and tasks into primitives

We consider here a task as an elementary self-sufficient behaviour. An animal behaviour for example may be split into different independent tasks such as movement, predation, growth or reproduction. The important notion to retain is that a task is autonomous. It may be activated or deactivated independently of the other tasks. Certain tasks are very common and are found in most models, and asking a user to create them from scratch would be illogical. In population dynamics, these tasks particularly concern ageing, survival, reproduction, mortality or movement. Users who wish to give such behaviours to their agents shall only select and parameterise the corresponding predefined tasks.

But predefined tasks, even flexible ones, will always result in stereotyped actions. The challenge of this project is to assume that most of the non-stereotyped tasks an agent has to execute may be broken down into successive stages that are always more or less identical: one must locate, sort and treat the relevant 'information' in order to update the system state. Hence, a task may be viewed as a chain of elementary sub-actions, and conversely, stringing up well chosen sub-actions will lead to the creation of new tasks (Fig. 1). In practice these sub-actions will be supported by small computer modules, that we call primitives, which include their own user interface in order to parameterise

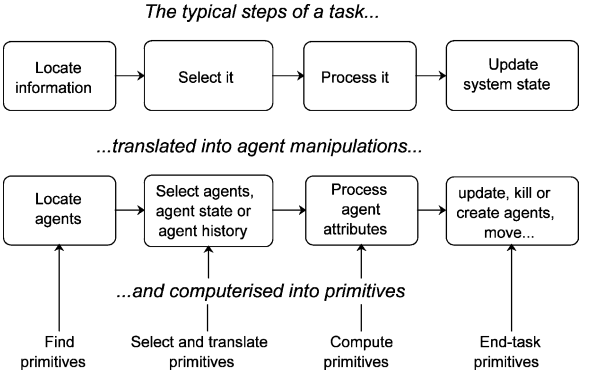

Fig. 1. Building tasks with primitives. A support for designing individual behaviours without the use of a programming language.

them, and which exchange information with other primitives through an argument. This needs to define what 'information' consists of. As shall be seen in Section 3, considerable simplifications are obtained if this information is supported by agents only.

3. An 'all agent' architecture designed for primitives

3.1. Simple reactive agents

In a MAS, reactive agents are often characterised by responding only to stimuli, whereas cognitive agents can respond to more complex information and usually take time to reflect before acting. Reactive agents have much simpler behaviours, which lend themselves to an easier breaking down into tasks and primitives. Hence only reactive agents shall be considered here, which represents a first restriction in the field of application of the primitive concept. In this architecture, agents may be considered as very simple objects including two dictionaries. A dictionary of attributes that shall fix the state of the agent (age, location, size etc.), and an ordered dictionary of tasks that shall fix its behaviour. The critical idea behind such simple agents is that end-users will freely build up agents by filling up these two dictionaries. In order not to multiply the data structures on which primitives have to work, only two types of attributes have been retained. The standard type provides a name, a value (quantitative or qualitative), and a unit. For example, an attribute 'size' in cm. The second type is similar but holds a list of values instead of a single one, so that an agent can store more information. As already said, the tasks are either predefined or user-defined with the help of primitives. They are executed in a sequential way, one after another, in the dictionary order.

3.2. 'All agent' models

In a MAS, all elements that are not agents go under the heading of environment. Agents play in this environment, searching for information and often being able to modify it. Environment is a source of richness, but also a source of complexity since the access it provides to the resources depends on the structure of the objects it contains. Access to a non-agent environment is thus difficult for primitives, since it multiplies the number of objects and data structures they have to handle. It is impossible to exclude the environment totally since output files or graphic interfaces displaying results are objects that also form part of the agent's environment. But two groups of objects may be distinguished: those that have an influence on the dynamics of the system and those that have not, hence those that belong to the model and those that do not. To simplify primitives' conception and use, the model should be described with agents exclusively, so that primitives treat and exchange only agents, or data structures that contain agent states. This 'all agent' choice forces us to better define the role of the agents since some of them will have to play the part usually played by the environment. We believe that three types of agents only may describe most IBM's.

3.2.1. The animats

This term was introduced by Wilson (1987) and was adopted by the International Society for Adaptive Behaviour (ISAB). Animats are all agents that are located in space and that may move or reproduce. Hence, they are the classical individuals of IBM's. Animats have two default attributes, location (as they are located in space) and number. The number attribute is set to one in order to represent an individual, to an integer for groups of identical individuals, or to a real number if the agent represents the concentration of an entire sub-population.

3.2.2. The cells

In an IBM, space is generally part of the environment and may be discrete or continuous. But since our environment must be agent, only discrete space, made up of agent cells, can be defined. Hence it is impossible to link a model to a continuous space directly, and therefore, this 'all agent' choice represents a second restriction to the application domain. But in return, space gain behaviour which opens the world of cellular automata.

3.2.3. The non-located agents

Another important role of the environment lies in its use as an information support for the simulation, namely for possible environmental scenarios or for the collection and display of important results. This is typically the role of a third type of agent, the non-located agent. These agents are not thrown into space as animats are, but always remain visible to any other agent as well as to the user. An agent (or the user) needs merely to know their name in order to access them. These agents may also be used for controlling or observing, statically or dynamically, groups of animats or cells. A default non-located agent is the `Simulator` agent, which holds the simulation time step and duration.

3.3. From the template to the real agent—communication between agents

In fact, the user does not directly fill the two dictionaries of the agents, but rather, fills the dictionaries of the templates of the agents, one template for each type of agent. The template is the model upon which all agents of this type shall be copied from during simulation. Hence, the template plays the role of the agent's class in more traditional simulators based upon object languages. But unlike these classes, the templates are not hard-coded once and for all by the computer scientist, but dynamically filled out by the end-user. The difficulty is now: how building these templates, and how, from these templates, generating real agents that share the same behaviour but have their own identity, i.e. behave like conventional objects? In classical object languages, classes are hard-coded by modellers, and the language engine automatically generates the agents by instanciation from these classes. With templates, we must mimic this process. We suggest a solution which proved to be very close to the Dynamic Objects Models design pattern, proposed by Riehle et al. (2000). Two conventional classes are used, the `AgentTemplate` and the `Agent` classes (Fig. 2). The first one provides methods to manage the filling of the tasks and attributes dictionaries by the user. The second provides general methods for the real agents, such as tasks calls and agent display. When the user wishes to define a new type of agent, a new template is instancied from the `AgentTemplate` class, and the user fills it. At runtime, real agents are classically instancied from the `Agent` class, but most of their state and behaviour is inherited from the template: they get an attributes dictionary copied out (duplication) from the template, and they are given direct access (a pointer) to the template's tasks dictionary. Hence, all agents originating from the same template have their own state (they have their own attributes dictionary), but the same behaviour (they point to the same template's tasks dictionary). But with such a simple pattern, it is impossible to use the attributes values (i.e. the agent state) in the tasks (i.e. in the agent behaviour), although in conventional object programming instance variables (the object state) can be used in procedure code (the object behaviour). Hence, our design pattern must be completed with links between behaviour and state (Fig. 2). With such links, any parameter in any task or primitive may either keep the template's value (therefore a fixed value common to all agents sharing this template), or become a dynamic link to the very value of an attribute of an agent. It should be emphasised that such a link may target not only the agent that owns the task (the running agent), but also any 'visible' agent in the context of the task. As a consequence, this design pattern allows the writing of (simple) interactions between agents, and we did not feel the need to implement the more complex computer notion of message. In this architecture an agent does not 'communicate' with other agents in the traditional sense of MAS's. It only launches the execution of its template's tasks, and these tasks can be given free access to its own attributes, as well as to the attributes of any other agent it is in contact with. We will further see on examples how with such links, an agent may harvest its cell, or a predator find and eat its prey.

3.4. Four structures and three worlds for the agents

The system handles templates and real agents. But for results saving it will have to handle agents' histories, i.e. objects that optionally save the

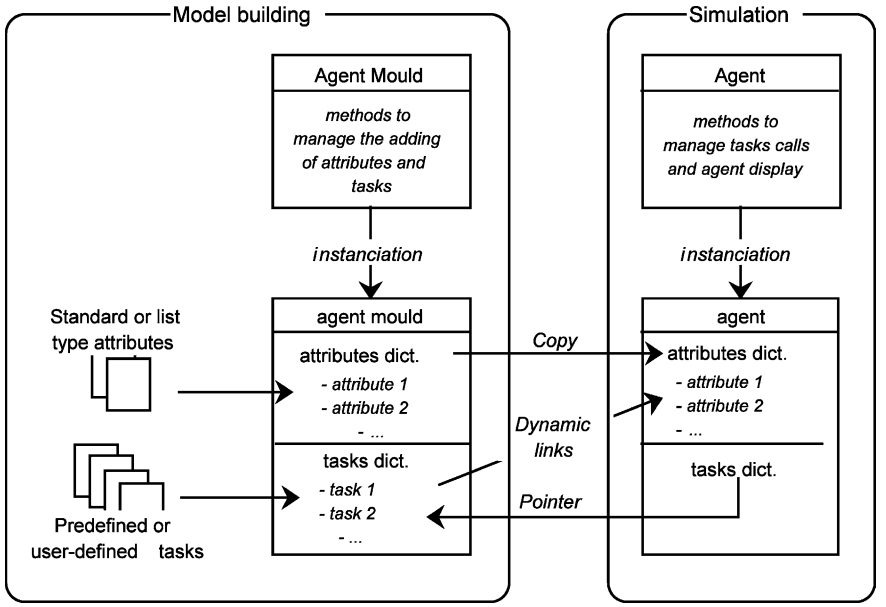

Fig. 2. Agent and Agent Template. To build agents, the end-user fills out the attributes and the tasks dictionaries of the templates, using predefined components and his own primitive-built tasks. At run time, real agents are created from the general `Agent` class, but attributes and behaviour come from the template.

successive states of agents in time. And in Section 3.5, we will see that for synchronisation purposes it will have also to handle agents' snapshots, i.e. the states of the agents at the end of the previous time step. Hence, the system handles templates, real agents, snapshots and histories. As these four entities are very similar, at least from there state (attributes) point of view, we suggest that they inherit from a common class (`ProtoAgent`) that manages access to attributes. Therefore, access to attributes is identical for a real agent, its template, its snapshot or its history, and a primitive, say `SelectOnAttribute` who select entities according to attributes' values, may work identically (or nearly identically) on any entity. To achieve this identical access to entities, they are plunged into four different 'worlds', the template, the real, the snapshot, and the history world. And these worlds (except for the template), share the same structure so that locating and sorting entities are carried out in the same way whatever the world. Hence very often, a primitive does not need to know in what kind of world or on what kind of agent structure it is working on. This much simplifies primitive conception and use, and provides a simple basis for data transmission between primitives and for agents synchronisation.

3.5. Time management and agent synchronisation

There are two ways to consider the temporal evolution of a system: continuous, or discrete. In the first case, focus is placed on the speed of evolution of the phenomenon, and the system is generally described with differential equations. In the second case, focus is placed on the events causing the system to evolve from one state to another between two time steps. Multi-agents simulation and IBM rely on discrete events, and our architecture uses the 'clock' mode (also known as discrete time simulation), used for its simplicity and common use, where time is divided into regular intervals. A much more powerful alternative is the 'event driven' mode (also known as discrete event simulation), in which agents generate 'events' on a common event queue, i.e. triplets that indicate to the simulation engine the agent and the task to be activated and the date of this activation. For discrete event simulation, see for

example the Ecosim framework (Lorek and Sonnenschein, 1998).

With a discrete time simulation, more than one event may occur over the same time step, causing synchronisation problems and possibly resource conflicts. Two great synchronisation modes are currently used. (1) Asynchronous (or sequential) mode in which each agent plays in turn during a time step, modifying the world in real time. In this mode, each agent sees the world as the previous one left it, and conflicts are avoided. This mode is the simplest, and used most often, being well adapted to numerous situations. However, the order of execution of the agents during a time step, generally random, may significantly influence results. (2) Synchronous (or parallel) mode in which all agents are supposed to play at the same time during a time step. This mode assumes that all agents 'see' the same world, in practice the state of the world at the end of the preceding time step. This state of the world must be saved, and this is the role of the agents and world snapshots (Fig. 3). Then the calling order of the agents has no influence but the modeller may face delicate conflicts. For example, a same prey should not be caught in parallel by several agents at the same time. Conflicts are difficult to handle as the solution depends on there biological meaning. At present, our architecture allows for a very simple solution only: the random choice of the one and only supplied agent. But this solution is rarely adequate and causes the calling order to get important again.

4. Building tasks with primitives

But back to tasks and primitives. The linear sequence of the different stages of a task as previously described (locate→sort→compute→update), leads us to expect that a simple primitive workflow should be sufficient for most cases: we suggest to use a linear sequence of primitives, which may be closed in a loop if needed. And for simplicity, primitives should exchange one argument only, the argument exiting one primitive forming the argument entering the next one. Always for simplicity, this argument should be a single object or a list of objects at the most. In our 'all agent' architecture, these objects are mainly agents. But primitives should also access agent histories, so that an agent may have memory. And

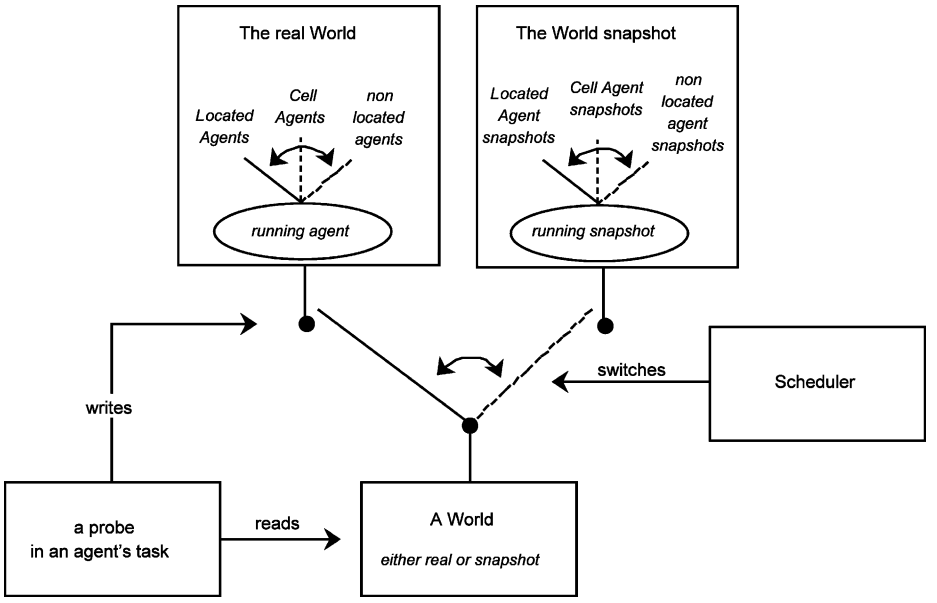

Fig. 3. Agent synchronisation.

in order to perform calculations on agent states (for example to compute the mean weight of a given group of agents), primitives should also exchange data on agent states or histories directly. Hence, we need to define a data structure that will hold the values of the attributes of given sets of agents or agent histories. This structure is multi-dimensional: you potentially have one data for each attribute of each agent and for each time-step. And a fourth dimension exists, that of the data itself as an attribute may hold a list of values in place of a single one. Now the first dimension (agent's attributes) does not present much interest: having to treat attributes of different natures globally is rare. Rather, it is a drawback since such a data structure could only hold agents having the same collection of attributes, when it is often more interesting to associate agents of different natures (for example juveniles and adults, or fishes from different species) that have at least one attribute in common. Hence, we suggest a three-dimensional table (the `3D_Array`), that carries data for one attribute only. The advantage is that a table for a single attribute is homogenous, with the same unit. The drawback is that imposing the same data structure to all primitives may lead to computer wastes: most of the time the table is only partially filled.

At present, 25 primitives have been defined, classified into six groups corresponding to six important functions (Table 1). (1) Research primitives are used as the departure point for the task. They do not have input argument and they return the agents or the data necessary for the continuation of the task. (2) Select primitives accept any list of agent on input and return a shorter list of agents that respond to certain criteria. (3) Transition primitives allow for the switch of one type of agent to another (e.g. from an animat to its cell or vice-versa), or from agents to data. (4) Compute primitives perform mathematical calculations. They differ depending on the input argument, agents list or 3D_Array data table. Amongst them, the primitive `ModifyAttributes` plays a pivotal role since it uses a mathematical formalism that involves agents attributes directly. (5) Close primitives are generally placed at the end of a task and are used for particular actions such as moving, killing agents, or saving results. (6) Control primitives allow exiting from the strict linear chain of primitives, in particular with the primitives '`While`' and '`Unroll`', which work in loop. Most of these primitives can be used by any of the three types of agent (animat, cell, or non-located agent). Tasks, whether predefined or created by the user, are considered as primitives without arguments. They may thus be integrated into more complex tasks. A simple example is to add a control primitive in order to make them conditional (the task is executed if ...), or to execute them in loops (the task is executed whilst ...).

Once the user has stringed out its primitives, a compiler checks for the consistency of the primitives sequence and dynamically creates the code of the new task. Hence, a user-defined task becomes a hard-coded class just as the predefined ones. And it has its own user interface, formed of the specific interfaces of each constitutive primitive. Therefore, primitive-based tasks are very flexible and may be saved by the user for use in other agents or models, besides the predefined ones. This way, the user progressively enriches his own task library and can reuse them for any agent or model.

5. Testing primitives on well-known models

The ability of the primitive concept to allow for the creation of various models has been tested on three well-known examples from literature: the model of DeAngelis et al. (1979), often considered as a pioneer of IBM's in ecology, the well-known game 'life' of John Conway (Gardner, 1970), and the SugarScape model of Epstein and Axtell (1996) which is very popular in economics. For these tests, we have used the platform we are developing on the basis of these concepts. Called MOBIDYC, an acronym for Modelling Based on Individuals for the Dynamics of Communities, this tool relies on this architecture and proposes a complete environment for the creation and use of models in population dynamics. In particular, it includes tools for the creation of cell agents (squared or hexagonal lattices, or issuing from ASCII contour files describing any shape of cells), tools to display

Table 1
Primitives description

Name	Function	Input	Output
Find			
Me	Returns the running agent		An Agent
MyCell	Returns the cell of the run. animat		A Cell
AllAnimats	Returns all living animats		A list of Animats
AllCells	Returns all Cells		A list of Cells
MyNeighbourhood	Returns adjacent cells of the running animat or cell according to a radius		A list of Cells
MyHistoric	Returns the historic of the run. agent		A HistoricAgent
MyHistoricAttribute	Returns an array with historic values of the specified attribute of run. agent		A 3DArray
Select			
SelectOnName	Selects animats that match a name	A list of Animats	A list of Animats
SelectOnValues	Selects agents that match conditions on their attribute values	A list of Agents	A list of Agents
SubSelect	Randomly selects a fraction of the input list (proportional or absolute)	A list of Objects	A list of Objects
FinalChoice	Chooses one agent of input list according to conditions on attributes (random, closest, biggest ...)	A list of Agents	An Agent
CellNeighbourhood	Returns the neighbouring cells of input cell according to a radius	A Cell	A list of Cell
Translate			
FromAnimatToCell	Returns the cells of input animats	A list of Animats	A list of Cells
FromCellToAnimat	Return all animats of input cells	A list of Cells	A list of Animats
FromAgentToHistoric	Return the historic of the input	A list of Agents	A list of HistoricAgents
FromAgentToValues	Returns a 3Darray filled with the values of the specified attribute; Dim1: attribute current values of input agents; Dim2: values in time (if input = HistoricAgents); Dim3 is used if attribute is of list type	A list of ProtoAgents (i.e. any kind of agent or Historic)	A 3DArray
Compute			
Count	Counts input items and put the result in an attribute of the running agent	A list of Objects	Output = input
ModifyAttributes	Computes mathematical calculations involving attributes of the running and of the input agent. If several input agents, repeats calculation for each	A list of Agents	Output = input
ArrayCalculations	Performs standard calculations (sum, mean, max, min) on any dimension (time, agents, or values of list type attribute) of a 3DArray. Output Array looses one dimension	A 3DArray	A 3DArray
End			
SaveValues	If dimension of 3DArray is 0 or 1, save input in the specified attribute of standard (dim 0) or list (dim 1) type	A 3DArray	Output = input
MoveWithTarget	Move directly to input cell, or move toward or away from input cell according to a specified speed	A Cell	Output = input
Kill	Kills all input animats	A list of animats	
Control			
Conditions	Continue task execution only if conditions are met on the running animat, its cell or any non-located agent		
While/End While	Loops primitives until conditions are met		
Unroll/End Unroll	Unrolls input list and gives single items to the following primitive. Stops at the end of the list or when conditions are met	A list of Objects	An Object

or save results such as a video recorder, a system to manage the units of the user, tools to generate initial agent populations automatically, and of course tools to generate new tasks from primitives. Besides the friendly primitive-based tool, we still provide a more conventional hard-coding tool to generate new tasks. A scheduler allows the user to schedule simulation: calling order of the agents, saving management, display procedures and choice of the synchronisation mode between agents. A last tool, too often absent from modelling workbenches, is a sophisticated batch procedure that allows the simulation experiments to be linked in order to test the sensitivity of the model to parameter values. For more information the reader may visit the site (in French, with soon an English summary) http://www.avignon.inra.fr/mobidyc.

5.1. Cannibalism in a fish school

This example, published in 1979 by DeAngelis et al., describes how sub-populations of young fishes (*Micropterus salmoides* or largemouth bass),

although sampled at the same time from the same tank and so coming from the same initial population, may evolve differently when split over different aquarium. The reason is that the sampled sub-populations do not have the same initial distribution: the presence or absence of a few larger individuals, able to eat their smallest brothers, is sufficient to switch the dynamics. Hence, such dynamics can no longer be described by the mean behaviour of the fishes, but the individual predatory behaviour must be taken into account. For each time step representing 1 day, each fish takes its turn and can reach all other fishes. It randomly chooses one of them and hunts it. It may capture it, according to a probability of capture, which depends on the size ratio between itself and its prey. If it succeeds, it eats the prey and consequently increases its weight. If it misses, then it searches for another prey. The cycle stops once all fishes have been hunted or when the running fish is no longer hungry. The next fish takes its turn, and so on for all fishes. Then, the next time step takes place. This rather complex behaviour is a good example for primitives. To give it to the fishes, the user proceeds with two steps. Firstly, he constructs the hunting task, stringing up the proper primitives (Fig. 4). Secondly, he fills the tasks dictionary of the agent template with this new task, setting up all the parameters and formulae of the constitutive primitives. One may note in particular the use of the primitive `UnrollInput` which enables the hunting cycle of the fish. The second key primitive is `ModifyAttribute`. As already said, this flexible primitive which is also a task if used without input argument, is designed to compute mathematical calculations involving numerical parameters or agent attributes, as soon as they are visible from (via there name), or in contact with (via the input argument), the owner of the task. For example, if the task is designed for an animat, expressions can involve attributes of the running animat (my_attributeName), of its cell (myCell_attributeName), or of any non-located agent (AgentName_attributeName). If used as a primitive, expressions can also involve attributes of the input agent (his_) or of his cell (itsCell_). To avoid syntax error the user only select items (i.e. formula and agent attributes) with the mouse (Fig. 5). This is a direct application of the dynamic links of Section 3.2. This example is not spatialised: all fishes live in the same aquarium represented by one cell. One may thus imagine using several cells to simulate a battery of aquariums, each containing a different initial distribution of the fishes, and using the batch procedure to replicate the experiment whilst modifying some parameters.

5.2. Cellular automata in parallel mode: the John Conway's game 'life'

The game 'life' of John Conway, popularised by Gardner (1970), is without contest the best known example of cellular automata running in parallel. Space is divided up into square cells with eight neighbours, the four faces plus the four angles. Each cell has two states, living or dead. For each time step, the principle of an automaton is that the state of the surrounding cells conditions the next state of the running cell. A living cell dies if surrounded by more than three or less than two living cells. A dead cell becomes alive if surrounded by three living cells. This behaviour implies two steps: the counting of the living neighbours and the updating of the state as a function of this number. As the simple linear workflow that we propose cannot manage the conditional 'if then else', these two steps are to be decomposed into three distinct tasks. The first strings out three primitives to count the number of the neighbouring living cells and updates a cell attribute, termed here 'livingNeighbours'. The other two are built upon the same model and put together the primitive `Conditions` and the task `ModifyAttribute`. They update the state of the cell according to the value of livingNeighbours. A user who wants to build the 'life' game will first use the lattice tool to create the grid. Then he will fill the attributes dictionary of the cell template with two new attributes, say 'livingNeighbours', and 'status' (0 if dead, 1 if alive). These two attributes are initialised with zero values. The next step is to build the two sub-tasks (as sub-tasks 2 and 3 are identical) by stringing the proper primitives (Fig. 6). The user can now fill the task dictionary with the three sub-tasks, setting up at the same time the

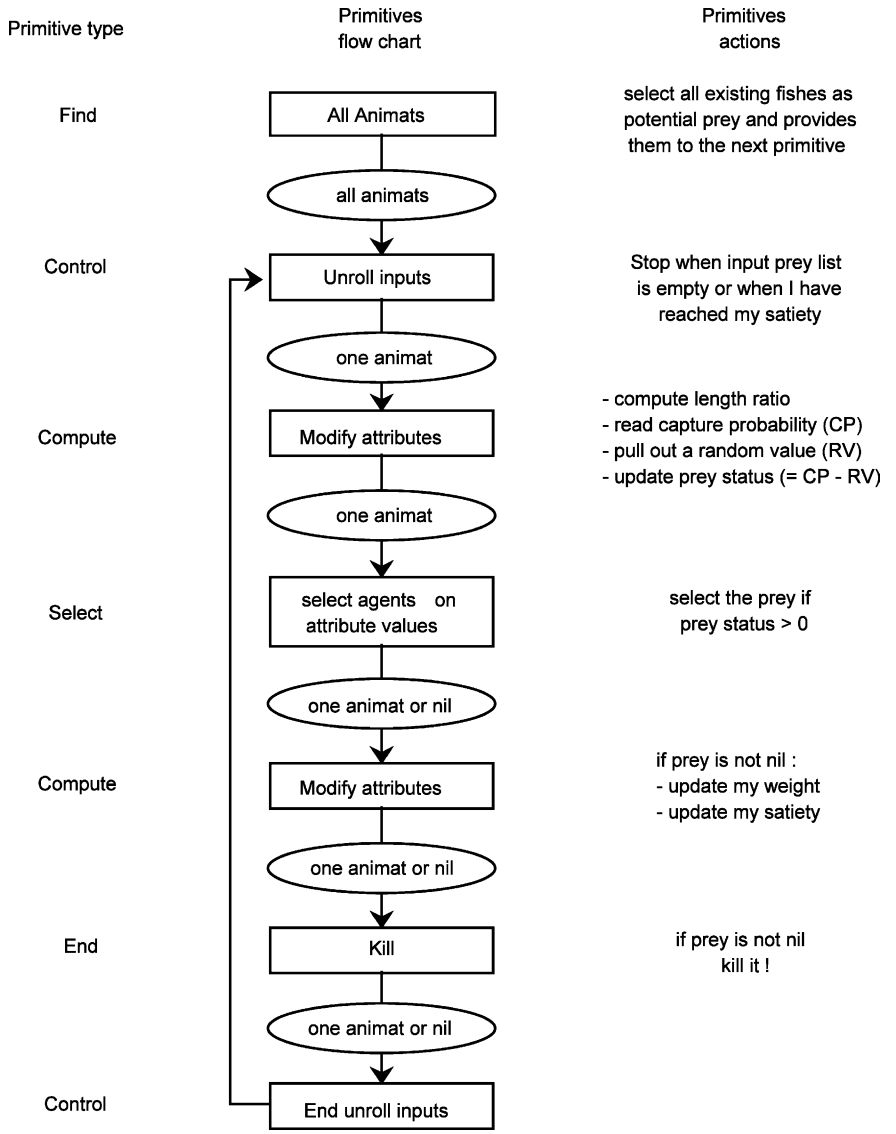

Fig. 4. Building tasks with primitives. The example of the hunting behaviour of the fishes in the DeAngelis et al. (1979) model. Seven primitives are stringed out to build this task. Boxes figure primitives, ellipses the primitive arguments.

behaviour (parameters or formulae) of each primitive. Before launching a run, the user must also define the display (which colours for living and dead cells), check the scheduler for a parallel call of the cell, and give an initial 'alive' status to some cells. All these operations do not need any computer knowledge and require not more than 10 min.

5.3. From agents to population: the SugarScape model

SugarScape (Epstein and Axtell, 1996) was developed to study agent organisation in space and time, when two resources are available: the one readily available and cheap, sugar, the other rare and expensive, spice. For our purpose, sugar

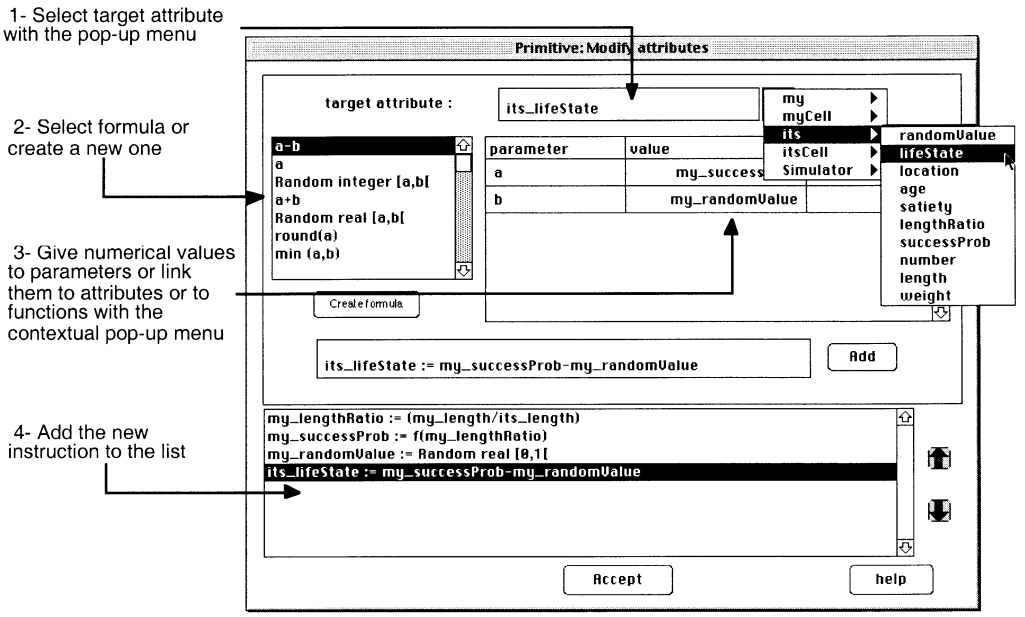

Fig. 5. The primitive `ModifyAttributes`, designed to compute mathematical calculations involving agent attributes. Presented here as used in the DeAngelis example to compute if a potential prey will eventually be eaten.

is sufficient and this example illustrates the possibility of writing complex explicit spatial population models. Hence we will linger a little on it.

5.3.1. *Agents basic behaviour*

Mobile agents (animats) search and harvest sugar from the cells. At each time step, they individually search their neighbourhoods for sites rich in sugar, accordingly to their range of vision. Once located, they move to the cell and harvest sugar. After harvest, sugar grows back at a unit rate until it reaches a maximum which depends on the cell (there are rich and poor cells). If animats are unable to source enough, they die of starvation, as they burn sugar to respire. Hence, only the most adapted or lucky ones survive. Four predefined tasks are used to build the basic behaviour of animats and cells (Fig. 7). Amongst them, notice `Move`, a predefined task that allows animats to select and directly reach a cell, according to a radius of motion (in cell units), and to some predefined ways of motion: random or searching for gradients on cell attributes. As motion is of importance in most IBMs, a user will often have to build his own displacements, being too limited by the predefined one. With some well-chosen primitives of Table 1 combining research, select and translate primitives, a target cell can be selected on any other criteria than the simple cells state: for example searching for the nearest cell containing animats in a given state. Then, the primitive `MoveWithTarget` allows to reach (or contrarily to avoid) it, immediately or with a given speed. But back to SugarScape. To complete the agent, the user must also fill the attribute dictionaries: age, range, sugar and breathRate for animats, sugar and sugarMax for cells. The fate of the animats, on this example randomly thrust on a 30×30 lattice with five levels of sugar, from 0 (white) to 4 (dark grey), is determined by their individual range of vision (randomly chosen between 0 and 3 cells), breatheRate (between 1 and 4), and initial sugar stock (between 0 and 6). After stabilisation, only a hundred of them survive, those having a low breathing rate (Fig. 8).

5.3.2. *From individuals to population: managing reproduction*

Switching from individual to population implies reproduction and possibly metamorphosis if ani-

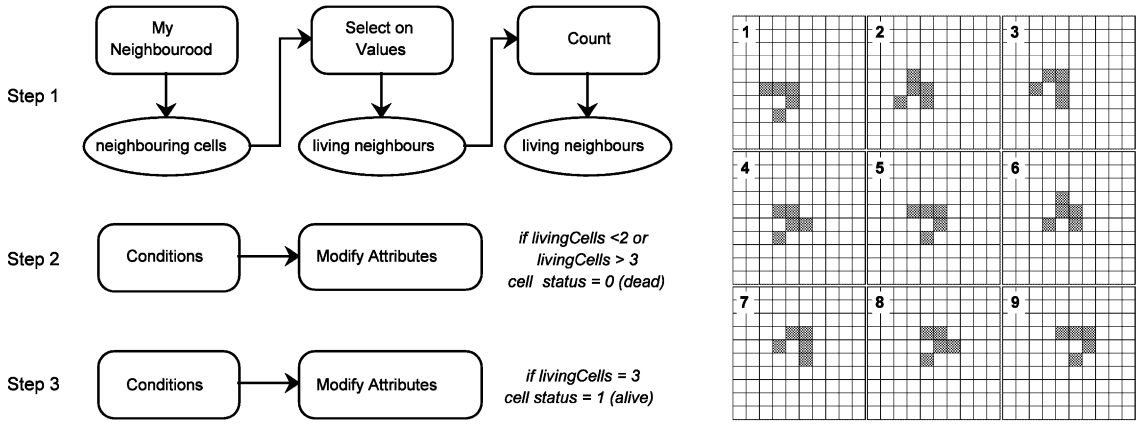

Fig. 6. Building tasks with primitives. The John Conway's cellular automata 'Life'. Left: primitives stringing and setting. Right: nine steps of the 'glider' configuration on a 10×10 lattice.

mats represent particular life stages of species. In particular, the initial values of the attributes of the new created animats must be set properly. We may imagine at last four ways to set those values: (1) take a default value (the value of the template); (2) inherit the parental value; (3) read this value from another agent the running animat is in contact with; (4) randomly select a value from a predefined list. One may combine these, and modulate (multiply or add) a given value with the aid of a value drawn from the list. For the purpose of our example, the attribute 'location' is inherited: the son is created in the parental cell. The range of vision is also inherited, but a series is used with the added option. As the series holds the values 0, 1 and -1, the son will have three equal possibilities: inherit the parental range, increase the range of one unit (one cell), or decrease it from one unit.

Agent task	Type	Setting
Animat		
Grow older	predefined	my_age := my_age + simulator_timeStep
Move	predefined	Move to the richest cell according to my_range
Eat	predefined (ModifyAttribute)	Two instructions: my_sugar := my_sugar + myCell_sugar myCell_sugar := 0
Breathe	predefined (ModifyAttribute)	One instruction : my_sugar := my_sugar - my_breathRate
Die	predefined	if my_sugar < 0 I'm dead !
Cell		
grow	predefined (ModifyAttribute)	my_sugar := my_sugar + 1 my_sugar := min (my_sugar, my_sugarMax)

Fig. 7. SugarScape. Basic behaviour of animats and cells. As in most models, `ModifyAttributes` is again a central task and is used three times. First to give the eating behaviour of the animat, second their breathing, and third the growing behaviour of the cells.

The breathing rate is randomly selected from a series containing the values 1, 2, 3 and 4, without involving default or parental values. The other attributes keep the default value of the template.

Fig. 9 shows an example of the development of a population with this inheritance management and with initial agents having the same initial range of one cell, and the same high breathing rate of three sugar units per time step (a day). An agent reproduces every 4 days and the simulation lasts for 3 months. After a short phase of 'adaptation' over which the population decreases, the population increases and stabilises at more than 1000 individuals (Fig. 9a). Logically, individuals with low respiration rates survive better (Fig. 9b). For the range, starting from a single value of 1, the 'natural selection' gives a nice gaussian-like curve (Fig. 9c). Zero or even negative values also occur. They are non-mobile animats that are able to survive when their respiration is at minimum (1 sugar per day), as the growing rate of the cells is also 1. But these 'hibernating' animats remain of no importance since they have no priority on their cell because of the random pull out of the animats at each time step: at any time a mobile animat (i.e. with a non-zero range) may come and harvest the cell before him. Hence, non-mobile animats are not assured to eat at each time step, and they always eventually die from starvation. For age structure, a highly asymmetric distribution is obtained with numerous young individuals (Fig. 9d).

5.3.3. *Giving memory to agents*

When cells become tired of producing sugar under the animats pressure and need to have a rest,

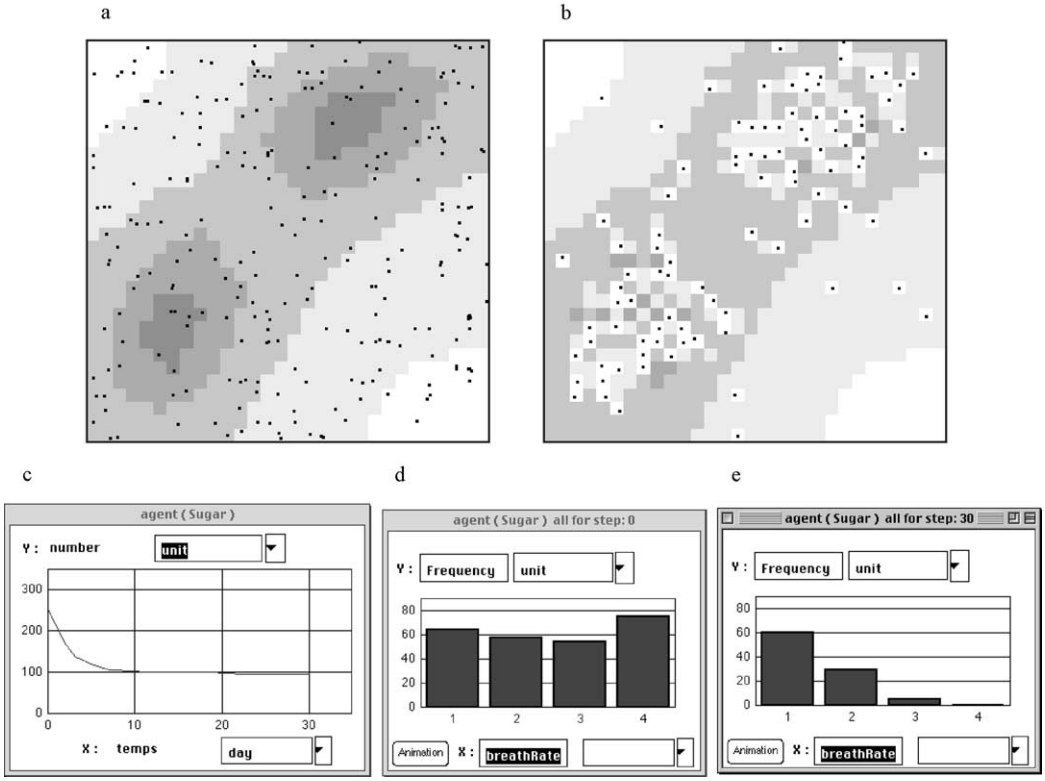

Fig. 8. SugarScape. A basic run. Initial population, 250 animats randomly thrust on a 30×30 lattice (a). After 30 days, about 100 animats survive (c), and concentrate on the richest cells (b). Initial (d) and final (e) distribution of the breathing rate in the population.

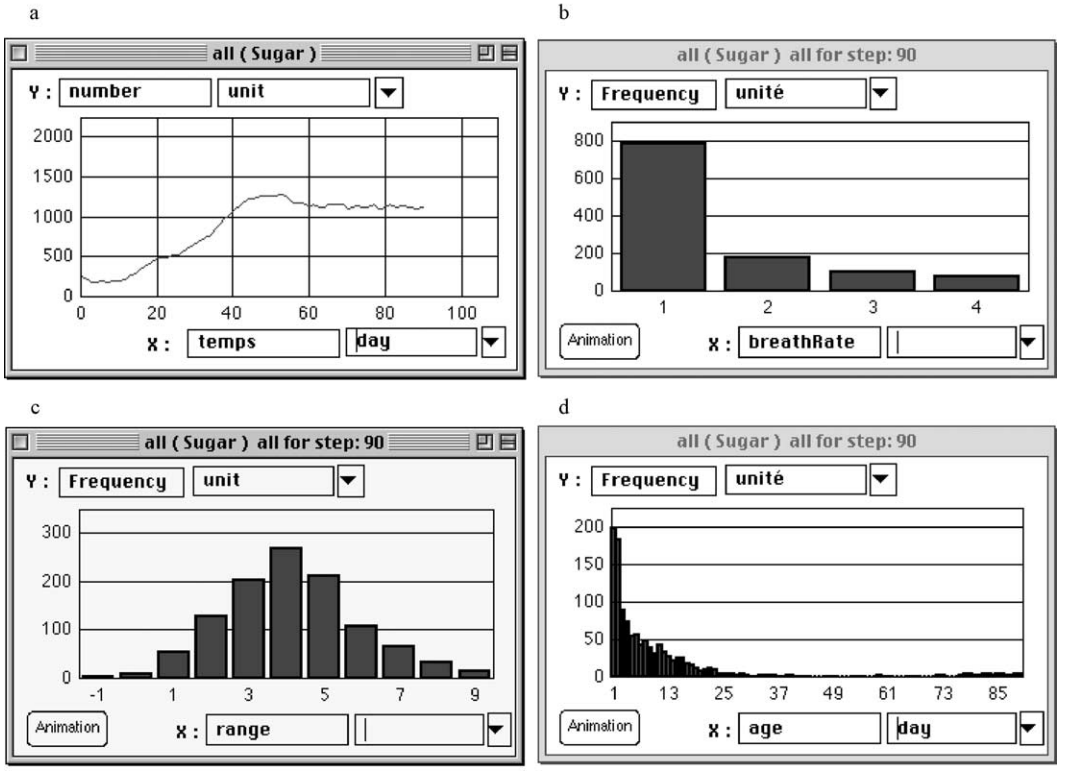

Fig. 9. SugarScape. A 3 months run with reproduction. (a) Population size; (b) final distribution of breathing rate; (c) final distribution of range; (d) final distribution of age.

there is overexploitation and habitat degradation. Overexploitation may be modelled in numerous ways. For example, cells may memorise the animat pressure over the past. If this pressure exceeds a certain threshold, the cells stop growing. If this pressure drops under another threshold, the cell starts to grow again. An example is shown on Fig. 10, where cells compute the mean number of animats they have supported over the last 30 time steps (1 month). The population does not install itself anymore over the whole area, but migrates regularly searching for sites where the growth of sugar has recovered. In order to define this behaviour for the cells, we have used the fact that primitives may access the agents history structures. An alternative is to give the cells an attribute of 'list of values' type in which they may save the information necessary for this calculation.

6. Conclusion

These examples drawn from literature show that numerous IBM's or cellular automata can be written with ease and speed by an ecologist with a few well chosen components. It must be emphasised that for all these examples (accessible on our web site with the Mobidyc package), no line of user-defined code was necessary, and that only a few but flexible and easy to understand primitives proved to be sufficient. Hence, it is no longer the computer encoding phase that takes time, as is too often the case with more conventional frameworks, but instead, the model design in interaction with the simulation. Our project intends to promote this way of component modelling. It proposes an 'all agent' architecture dedicated to population dynamics, and designed to make the end-users true masters of their models without

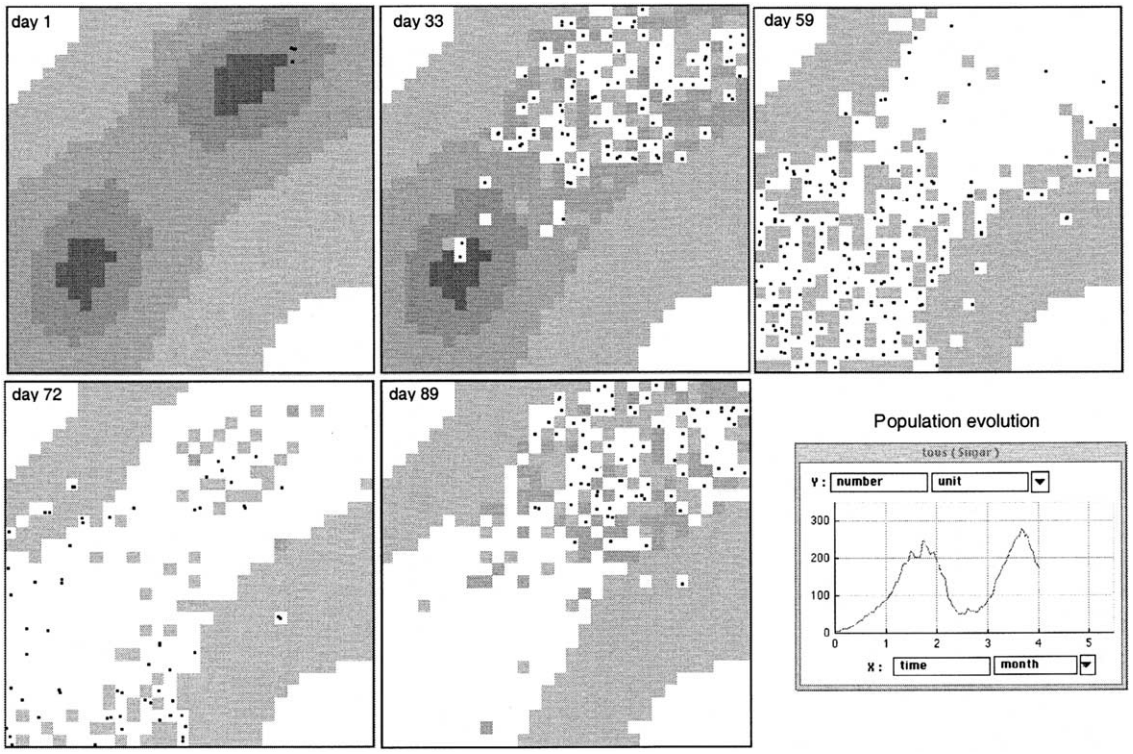

Fig. 10. SugarScape. Giving memory to agents. As primitives can access agent memory, computing cell overexploitation is straightforward. Here, cells stop growing when they are too often harvested, and restore their growth when agent pressure decreases. Starting from four initial agents, population fluctuates and agents migrate from 'striking' cells (white cells) to growing ones (grey ones).

assistance of computer experts. The proposed architecture is based on executable user-defined tasks made up of elementary primitives and manages agent synchronisation, results saving and the carrying out of simulation experiments. This architecture facilitates model conception, restricts programming errors and increases readability. One must however remain realistic. At present, such a tool is designed for rapid prototyping, intended for researchers who wish to test hypotheses rapidly on rather simple conceptual models. One should be conscious of the technical limits that the choices in computer programming impose to the platform in terms of calculation speed (often several minutes per run) and number of agents (from 1000 to 50 000 agents maximum, depending on the complexity of interactions between them). Complex and interdisciplinary models will probably always require a specific programming effort in order to optimise processing speed and memory use, and hence still require the aid of skilled computer scientists or modellers.

The great advantage of making IBM's with MASs is that they allow for a gradual programming of the models. Each agent, or better each task, may be tested in an isolated manner before integration into the whole model. This is fundamental for creating simple models and only gradually increase the level of complexity, or conversely for simplifying a model by deactivating agents or tasks. It also helps to understand better the behaviour of the agents and to unravel the inevitable errors of parameterisation, which is always a delicate step with more classical modelling approaches.

But one difficulty is that there is still no formalism, such as differential equations, where one can express, conserve and compare one model to others, or to transport it to another modelling tool. At present, an IBM or a MAS cannot be

dissociated from the simulator carrying out its execution, and there is no formal link between the language of the researcher describing his model and the very programming language in which it is actually written. Hence, an observer will rightly show some hesitation in the trust of these models, which still often look very much like black boxes. This problem of readability is crucial and still probably insufficiently discussed in our modelling community. Briefly evoked by Judson (1994), this point has been dealt with by Lorek and Sonnenschein (1999) only. We are in total agreement with their opinion that a user should be able to re-run a model from literature regardless of the platform. This imposes a better separation of the components of the model from those of the platform. And the result shall be more reliable as the platform shall have control on more components involved in the execution of the model. This is also the approach we support. Furthermore, we expect that a primitive-based component programming should also facilitate the development of a platform-independent language in which the models could be written, as such a language probably represents the only true solution to this problem of formalism.

To conclude, let we emphasise that IBM does not represent the only nor necessarily ideal approach for the modelling of populations, and the reader may refer to Judson (1994), Uchmanski and Grimm (1996) or Lomnicki (1999) for a more in depth discussion on this topic. In all situations where individual aspects are not important, or when a heterogeneous space does not have to be taken into account, it shall always be of preference to use a more classical approach, based on differential equations or matrix calculations for example. These approaches offer safe numerical results, as well as a mathematical formalism that allows for a useful theoretical analysis of the behaviour of the model to be carried out. For compartment (box) models for example, and many biological systems may be described this way, the already ancient precursor Stella™ represents for us the archetypal modelling tool for an end-user. We hope that the primitive-based component programming we have exposed here will represent a step in this direction, for systems where IBM represents a good modelling choice.

Acknowledgements

This work was supported by the French National Centre for Scientific Research (CNRS), section 'Biodiversitas' of the 'Environment Life and Societies' program, through the thematic action of the Public Group of Interests HydrOsystèmes entitled: 'the fish in its environment'. It involves teams from the French National Institute for Agricultural Research (INRA), University Paris-VI (LIP6), the French Institute of Research for Development (IRD), and the French Centre for International Cooperation in Agronomy Research for Development (CIRAD-Tera/Ere). The MOBIDYC platform (Beta version) and its code are freely available in French and English and a tutorial (in French) is provided. It uses the Smalltalk Visual Works 5i environment developed by the Cincom company. This environment is free for non-commercial use and runs on almost all platforms. http://www.avignon.inra.fr/mobidyc.

References

Bousquet, F., Morand, P., Quensière, J., Mullon, C., Pavé, A., 1993. Simulating the interaction between a society and a renewable resource. J. Biol. Syst. 1 (1), 199–214.

Bousquet, F., Bakam, I., Proton, H., Le Page, C., 1998. Cormas: Common-Pool Resources and Multi-agent Systems. Congrès IEA-98-AIE, Lecture Notes in Artificial Intelligence no. 1416. Springer, pp. 805–814.

Conrad, M., Patten, B.C., 1970. Evolution experiments in an artificial ecosystem. J. Theor. Biol. 28, 393–409.

DeAngelis, D.L., Cox, D.K., Coutant, C.C., 1979. Cannibalism and size dispersal in yong-of-the-year largemouth bass: experiment and model. Ecol. Model. 8, 133–148.

DeAngelis, D.L., Gross, L.J., 1992. Individual-based Models and Approaches in ecology. Populations, Communities and Ecosystems. Chapman & Hall, New York.

Drogoul, A., Corbara, B., Fresneau, D., 1992. Applying EthoModeling to Social Organization in Ants. In Biology and Evolution of Social Insects. Leuven University Press, Leuven, pp. 375–383.

Epstein, J.M., Axtell, R.L., 1996. Growing Artificial Societies. Social Science from the Bottom Up. MIT Press, Cambridge, p. 208.

Ferber, J., 1999. Multi-Agents Systems. An Introduction to Distributed Artificial Intelligence. Addison-Wesley, p. 509.

Ferreira, J.G., 1995. ECOWIN. An object-oriented ecological model for aquatic ecosystems. Ecol. Model. 79, 21–34.

Gardner, M., 1970. The fantastic combinations of John Conway's new Solitaire Game 'Life'. Sci. Am. 23 (4), 120–123.

Grimm, V., 1999. Ten years of individual-based modelling in ecology: what have we learned and what could be learn in future? Ecol. Model. 115, 129–148.

Gutknecht, O., Ferber, J., 1997. MADKIT: organizing heterogeneity with groups in a platform for multiple multi-agent systems. Rapport Interne LIRMM, Université de Montpellier, 1997. www.MadKit.org.

Hogeweg, P., 1989. MIRROR beyond MIRROR, puddles of life. In: Langton, C.G. (Ed.), Artificial Life. Addison-Wesley, pp. 297–315.

Hogeweg, P., Hesper, B., 1990. Individual-oriented modelling in ecology. Math. Comput. Model. 13 (6), 83–90.

Judson, O.P., 1994. The rise of the individual-based model in ecology. Trends Evol. Ecol. 9 (1), 9–14.

Kaiser, H., 1979. The dynamics of population as a result of the properties of individual animals. Fortschr. Zool. 25 (2/3) 109–136; G. Fischer, Stuttgart, New York.

Kreft, J.-U., Booth, G., Wimpenny, J.W.T., 1998. BacSim, a simulator for individual-based modelling of bacterial colony growth. Microbiology 144, 3275–3287.

Le Page, C., Cury, P., 1996. How spatial heterogeneity influences population dynamics: simulations in SeaLab. Adaptive Behav. 4 (3/4), 255–281.

Lhotka, L., 1994. Implementation of individual-oriented models in aquatic ecology. Ecol. Model. 74, 47–62.

Liu, J., Ashton, P.S., 1998. FORMOSAIC: an individual-based spatially explicit model for simulating forest dynamics in landscape mosaics. Ecol. Model. 106, 177–200.

Lomnicki, A., 1999. Individual-based models and the individual-based approach to population ecology. Ecol. Model. 115, 191–198.

Lorek, H., Sonnenschein, M., 1998. Object-oriented support for modelling and simulation of individual-oriented ecological models. Ecol. Model. 108, 77–96.

Lorek, H., Sonnenschein, M., 1999. Modelling and simulation software to support individual-based ecological modelling. Ecol. Model. 115, 199–216.

Marcenac, P., 1997. Modélisation de systèmes complexes par agents. Technique et Science Informatiques 16 (8), 1013–1038.

Minar, H., Burkhart, R., Langton, C., Askenazi, M., 1996. The swarm simulation system: a toolkit for building multiagent simulations. Santa Fe Institute Working Paper 96-06-042.

Resnick, M., 1996. Beyond the centralized mindset. J. Learn. Sci. 5 (1), 1–22.

Riehle, D., Tilman, M., Johnson R., 2000. Dynamic object model. In: Proceedings of the 2000 Conference on Pattern Languages of Programs (PLoP 2000), Washington University Technical Report number: wucs-00-29.

Sekine, M., Nakanishi, H., Ukita, M., Murakami, S., 1991. A shallow-sea ecological model using an object-oriented programming language. Ecol. Model. 57, 221–236.

Shin, Y.-J., Cury, P., 2001. Exploring fish community dynamics through size-dependent trophic interactions using a spatialized individual-based model. Aqu. Liv. Res. 14, 65–80.

Taylor, C.E., Jefferson, D.R., Turner, S.R., Goldman, S.R., 1989. RAM: artificial life for the exploration of complex biological systems. In: Langton, C.G. (Ed.), Artificial Life. Addison-Wesley, pp. 275–295.

Uchmanski, J., Grimm, V., 1996. Individual-based modelling in ecology: what makes the difference? Trends Evol. Ecol. 11 (10), 437–441.

Van Winkle, W., Jager, H.I., Railsback, S.F., Holcomb, B.D., Studley, T.K., Baldrige, J.E., 1998. Individual-based model of sympatric populations of brown and rainbow trout for instream flow assessment: model description and calibration. Ecol. Model. 110, 175–207.

Weiss, G. (Ed.), Multiagents Systems: a Modern Approach to Distributed Artificial Intelligence. MIT Press 1999, p. 619.

Wilson, S.W., 1987. Classifier systems and the animat problem. Machine Learn. 2, 199–228.