Case study **24**

Agent-based modeling and simulation

Part 1

Objectives. Agent-based modeling aims to simulate the behavior of complex real-world systems made up of a population of interacting autonomous entities. These elementary (microscopic) interactions define the overall (macroscopic) behavior of the system. For example, the expansion and reduction of a pandemic depends on the behavior of each member of the affected population. In agent-based modeling, each entity of the system is represented by a software component, called an agent, which models its main properties and behavior rules. Starting from a set of initial parameters, each execution of this model simulates a history of the complex system. The analysis of the data generated by a series of simulations enables us to better understand and predict the evolution of the complex system.

This study experiments with the development of agent-based models using database concepts. It is shown that these concepts apply naturally to the specification of the complex data used and generated by simulators. It is also shown that the *trigger* mechanism perfectly translates the behavioral rules of agents and their interactions.

In this first part of the study, we build a series of models of increasing complexity. Each is the subject of a parametric implementation enabling the global behavior of the modeled system to be visualized and analyzed.

Keywords. agent-based modeling, large system simulation, moving agent, agent life cycle, time management, trigger, UDF, active database.

Table of content

24.1 Introduction

Let us start with a concise but instructive definition of Agent-based models:

> *Agent-based models (ABMs) are computational models that simulate behavior of individual agents in order to study emergent phenomena at the level of the community. Depending on the application, agents may represent humans, institutions, microorganisms, and so forth. The agents' actions are based on autonomous decision-making and other behavioral traits, implemented through formal rules. By simulating decentralized local interactions among agents, as well as interactions between agents and their environment, ABMs enable us to observe complex population-level phenomena in a controlled and gradual manner.*
>
> Quoted from Šešelja, Dunja, *Agent-Based Modeling in the Philosophy of Science* in *Stanford Encyclopedia of Philosophy*, Sept 2023 [https://plato.stanford.edu/entries/agent-modeling-philscience/]

A first illustration

We consider a road network comprising highways, ordinary roads and connections between them. Many vehicles circulate on this network. As the traffic becomes denser, we observe phenomena that are difficult to explain.

For example, why, in certain sections of this network, at certain times of the day, the flow of traffic is often slowed down, sometimes to a halt, although no particular upstream disturbance has been observed to explain this phenomenon?

The attitude generally adopted by the engineers responsible for the network is to build a computer model of the network and the vehicles, and run it according to different parameters. If the model is accurate, it will behave in a similar way to the phenomena observed in reality. By varying these parameters, the users of the model can establish the relationship between certain sets of the parameter values and traffic behavior. They can then decide to widen certain roads, divert part of the traffic or limit vehicle speed.

Then comes the critical question: what modelling technique should we adopt?*

One of the most popular techniques is to consider that the flow of vehicles on the network is analogous to the flow of a compressible fluid in a pipe network. Thus, by applying the equations of *fluid mechanics* to our problem, the engineers can calculate, at each point of the network, at each instant, quantities such as the speed and density of the flows of vehicles. By varying the values of the parameters of the equation system, they try to identify the critical conditions that produce the observed phenomena.*

The technique we are going to develop in this study takes the opposite approach. Instead of building a global model of the road traffic system, we will model each section of road and each individual vehicle in the form of software components called *agents*. A section of road is characterized by the condition of the surface and the number of lanes. Each vehicle is described by its dimensions, engine power and

driver's physical condition. We also describe the driver's reaction in certain circumstances, such as a reduction in the distance between vehicles, the perception of an obstacle or the dangerous behavior of other drivers.[1]

Once the agents have been coded, they are run and their behavior is examined in order to deduce the evolution laws of the traffic system.

In the first (top-down) approach, we infer the behavior of each network point from the general model. In the second (bottom-up) approach, we deduce the general rules from the observed behavior of each network point. This latter approach is known as *agent-based modeling and sim*ulation.

Agent-based modeling

An agent is an autonomous piece of software designed to simulate some of the properties and the behavior of a real world entity. So, an agent is a *model* of this entity. We consider real world entities that communicate between each other as well as with their environment. The entities we are interested in in this essay have a dynamic behavior: they move, they explore and possibly modify their environment, they exchange information, they acquire new knowledge (they learn), they look for resources, they meet, they are born, they grow up then they die. In short, they are, or behave like, living beings.

A collection of real world entities situated in a close environment constitutes a system that evolves according to a set of laws. In the computing world, the agents that simulate these entities constitute also a system driven by a set of rules that translate these laws. Since such real systems comprise a large number of agents, possibly of different species, one often speak of *multi-agent systems*.

The agent system is intended to be a model of the real system. What one expects from a good model is that the way the agents evolve be a faithful image of the evolution of the real system. In this way, the user of the agent system hopes to learn how and why the real system evolves as it does, and get enough knowledge to modify the evolution of the real system.

What is an agent made up of?

Studies available in the scientific literature and more generally on the web show that there is a huge variety of agents, from individual grains of sand in the study of beach stability during high tides to human beings in the study of pandemics.

An intuitive way of understanding the nature of the agents we'll be studying in this essay might be to adopt a *zoomorphic* analogy. For example, to study the evolution of a primate population, we model each animal as an agent to which we assign

1. Nguyen, J. et al., An overview of agent-based traffic simulators, in Transportation Research Interdisciplinary Perspectives, Vol 12, Dec. 2021, 100486 [https://www.sciencedirect.com/science/article/pii/S2590198221001913]

certain properties (age, gender, health status, social status, etc.) and specify how it reacts to certain conditions or events.

Compared with other modeling paradigms, such as object-oriented models, agent-based models exhibit some distinctive properties; among them:

- the agents of a model are *distinct* entities; to this aim they are given a unique identifier (called AgentID in the following),
- agents are defined by their *properties* and their *behavior*,
- the behavior of an agent is defined by a set of *rules* that specify how to act according to each situation,
- agents evolve in an *environment* which they can examine and modify,
- agents are *active*; they can perform tasks, such as moving, interacting with their environment and exchanging information with other agents,
- agents have a *state* which records the current (and possibly passed) level of their properties,
- agents can *learn* from experience and accumulate knowledge,
- agents are *autonomous*; they are able to decide which actions to carry out,
- agents contribute to an individual or common *objective*.

Not all agent models include all these properties. For example, cellular automata[2] are sometimes considered as agent models, despite their extreme simplicity and deterministic evolution rules.

Among the properties mentioned above, the first and most fundamental that we will study is that of the specific strategies according to which agents *move*.

Agent-based simulation

The aim of an agent system is to generate information that enables its users to understand how and why a real-world system behaves, and to predict its behavior under different conditions.

Each run of this system simulates a possible history of the real system. Starting from realistic initial conditions, the agents are prompted to evolve according to the rules of their behavior. At regular intervals during this execution, users (the observers) collect statistics from which they expect to gain a better understanding of the behavior of the real system.

The functions of a simple ABMS simulator, as conceived in the experiments below, are as follows:

- allow the user to specify the agent model,
- run a series of simulations of a model,

2. See for example Case Study 27: *Conway's Game of Life*, available on the SQLfast web site.

– visualize the evolution of the model (through still pictures or animation),
– generate statistics,
– validate the model (verify the code for programming errors, check the adequacy of the model as a faithful representation of the real-world system).

ABMS applications

TBW

ABMS technologies and tools

ABMS simulators can be coded in general languages such as Java, Python[3] and C#, sometimes with the help of specialized libraries. However, to implement large and complex agent systems, using specific ABMS platforms is strongly recommended, thanks to the richness of their functions, notably in result analysis and visualization.

According to a recent comparison of ABMS platforms and tools[4], Swarm[5], RePast[6] and NetLogo[7] are among the most frequently referenced simulation environments.

Why use database concepts to build agent-based models?

This is the central question of this study. How can a technology whose primary purpose is simply to store data be considered suitable for implementing complex ABMS systems?

Maybe an illustration of the popular adage: "*If your only tool is a hammer, every problem looks like a nail*"? To a monomaniacal database expert, couldn't every problem be solved by a database?

The question isn't entirely bizarre. Those familiar with database technology know that most DBMS are also, in a way, *application development environments*. First

3. See for example Bassel Karami, *Intro to Agent Based Modeling* [https://towardsdata-science.com/intro-to-agent-based-modeling-3eea6a070b72]
4. https://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software
5. https://en.wikipedia.org/wiki/Swarm_(simulation)
6. https://en.wikipedia.org/wiki/Repast_(modeling_toolkit); https://repast.github.io/index.html
7. Wilensky, U., Rand, W., *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*, The MIT Press, 2015

and foremost, SQL is a universal and powerful language, particularly suited to the creation and management of large, complex and rapidly evolving data sets. Given the volume, variety and complexity of the data used and generated by ABMS simulators, storing this data in a database is the first thought that comes to mind.

Secondly, and most decisively, SQL includes the *trigger* mechanism, a powerful ECA (*Event Condition Action*) function that converts a passive data repository into an *active database*[8].

Consider a table TA whose rows represent the state of objects of type A, for instance *agents*. An SQL trigger associated with this table is the immediate translation of the type of rules that make up the agents' behavior:

- **event**: whenever an event affects an object of type A
- **condition**: if some condition is met,
- **action**: then this object is asked to execute this action.

These components are directly translated into the trigger syntax:

```
create trigger Tr_TA
after/before <data modification event> on TA
when <condition to be satisfied>
begin
   <action to execute>
end;
```

24.2 Architecture of a simulation

The simulators we will develop comprise two main components: the controller and the world. The controller is common to all the agent systems while the world is the implementation of a specific agent system.

24.2.1 The controller

The controller's responsibility is to create the space in which the agents will live and evolve, to populate it with a collection of agents and with the resources they may need, to start the simulation, to control the evolution of the world and to extract the information we would like to know about this evolution, such as statistics and static or dynamic graphical representation.

A critical question is the way the simulation makes the world evolve, that is, *time management*. The theory of agent-based simulation suggests two main strategies: discrete event simulation and discrete time simulation.

In the first strategy, *discrete event simulation*, each agent writes the events that will affect it (for instance, the date it will get sick after being infected) on the time

8. See for example Case Study 8: *Active databases*, available on the SQLfast web site.

line. Then the controller advances from date to date to activate the concerned agents.

In the *discrete time (or clock-based) simulation*, the evolution of the world is decomposed into a sequence of steps, corresponding to (generally small) equal size time slices. During each time slice, all the agents have the opportunity (or are required) to execute a small step of their *life*, that is, the scenario which they are programmed for. At the completion of a step, the world is in a new state. For small time slices, it can approximate a continous behavior of the world.[9] The controller triggers each step and analyzes each generation, not only to extract useful information and to show graphically the state of the world, but also to check that the conditions of the simulation are still met (for instance, there should be enough rabbits to feed the foxes, otherwise, they will all die, which will terminate the simulation).

The strategy we will implement in our simulators is a variant of the latter technique in which the duration of a slice is the time needed for all agents activated in the step to complete the actions required by the rules that define their behavior.

To start each step, the controller sends a *message* to some agents (all of them or a selected subset) to make them act according to their behavior rules. In the simulation, each active agent analyzes its environment, moves to a neighbor cell according to the result of this analysis then applies its evolution rules. The step stops when all the agents that were activated have finished their actions.

At the completion of each step, the controller examines the state of the world and records the desired statistics. It also determines when the simulation must be terminated.

24.2.2 The world

One generally distinguishes the agents from their environment, that is, the virtual space in which the agents act.

The world is made up of a finite space[10] structured as a grid of cells identified by their (X,Y) coordinates. This space is populated with a collection of agents. The agents can be of a single kind (for example, cars in a road network) or, in more complex worlds, agents of different types can coexists and interact (such as rabbits and foxes). Each cell may contain one or more agents as well as, in some cases, other objects the agent can interact with. The agent is represented by its properties, among them the coordinates of the cell it is located in. These properties form the

9. In the last two experiments of this part (*Life cycle* and *Life cycle with birth control*), the agents are given two new properties, Ticks and Latency, implemented as autoincrementing counters that specify future time points on which the agent acquires new capabilities. They provide a limited but inexpensive substitute to the scheduling facility on which the *discrete event simulation* is based.

10. Some agent models do not impose the finiteness constraint of their space. Such is the case of the Conway's cellular automata, already cited, which include no boundary conditions. They are not considered here.*

state of the agent. An agent can examine the content of any cell, besides that in which it resides. It can receive a message, in which case it reacts in a way that depends on the nature of the message and on its environment. It can also send messages to other agents and examine their properties.

In this essay, we consider that the agents are free to move across the cells of the space. Therefore, the way they move is an important aspect of the evolution of the world.

24.2.3 SQL implementation

The world is represented by two main tables, CELL and AGENT. Each row of CELL describes the properties of a cell: its absolute *coordinates* (columns X and Y) and its *content* (column Content), the precise meaning being left undefined for now. Each row of AGENT contains the properties of an agent, among them, its *Id* (column AgentID) unique among all the agents, the *coordinates* of the cell it is currently located in (columns X and Y), and the last *message* it received and to which it is currently reacting (column Message). To send a message to an agent the sender just updates the value of the Message column of the row of this agent, an event that will be caught by the appropriate trigger.

This is translated into the declarations of Script 24.1. Except for the primary key of AGENT, considering the strict way the data will be managed, we have defined no constraints. Other properties will be added according to the needs of the different agent systems we build.

```
create table AGENT(AgentID integer not null primary key,
                   X       integer,
                   Y       integer,
                   Message varchar(16) default '');
create table CELL(X       integer,
                  Y       integer,
                  Content integer default 0);
```

Script 24.1 - The main tables describing the world (first draft)

The space is defined by its width and height dimensions (in terms of cells), specified by variables **width** and **height**. A space of 60 rows of 80 cells is defined as follows:

```
set width = 80
set height = 60
```

If the rectangular shape of the space is considered inappropriate, more complex configurations can be built by placing special objects in certain cells to prohibit their access. They will be studied in Section 24.11 in the form of *repellers*.

24.3 The controller

24.3.1 Principles

The controller comprises four main operations:

– acquisition of simulation parameters, in particular the number of steps to be executed,

– creation of the initial state of the world, including the environment and the starting set of agents,

– execution of the specified number of simulation steps,

– processing the collected data, in particular graphical representation of the successive states of the world.

The last operation is important for the ABM simulations we plan to develop. We decide to display each state of the world graphically so that its evolution appears like an animated movie. After completion of each step, the description of the current state is stored in a graphic file using the SQLdraw graphics language.

The controller structure can then be refined in the following pseudocode:*

```
Create the tables
Get the simulation parameters
Create the world
Populate the world
Generate the initial state in the graphic file
for step = [1,maxStep]
    Send an activation message to the agents
    Generate the current state in the graphic file
endfor
Process the simulation history
```

We decide to include an intermediate concept, the clock, materialized by the CLOCK table. To start each step, the controller sends a message to the clock by inserting a row containing the sequence number of that step. Then, the clock sends an activation message to the selected agents[11]. Distributing the execution responsibility enables us to build a simpler, more abstract controller.

Now, we study the implementation of this architecture in SQL.

24.3.2 SQL code

The code of the controller is independent of the agent systems it activates and controls. Each agent system is materialized by a simulator defined by six procedures called by the controller:

11. Actually, all of them in the first part of this study

- AGENT-Create-Tables: creates the base tables of the simulator.
- AGENT-Get-parameters: asks the user for the parameters of the next simulation.
- AGENT-Create-World: specifies the behavior of the agents
- AGENT-Populate-World: if needed, places special objects in the space (such as obstacles or resources); creates the initial set of agents and places them in the space.
- AGENT-Generate-Current-State: analyzes the current state of the world and stores its description in historical data structures for further processing; this description is translated into graphical commands and, for more complex agent systems, into statistical measurements.
- AGENT-Process-History: the graphical and statistical descriptions of successive states of the world created during simulation are translated into usable information for the user.

The code of the SQLfast controller is shown in Script 24.2.

```
execSQL _AGENT-Create-Tables.sql;
execSQL _AGENT_Get-parameters.sql;
execSQL _AGENT-Create-World.sql;
execSQL _AGENT-Populate-World.sql;
execSQL _AGENT-Generate-Current-State.sql;
for step = [1,$maxSteps$];
  insert into CLOCK values($step$);
  execSQL _AGENT-Generate-Current-State.sql;
endfor;
execSQL _AGENT-Process-history;
```

Script 24.2 - General structure of the monitor

The structure of the CLOCK table is shown in Script 24.3. It associates a trigger to the table to catch the event created by the controller.

```
create table CLOCK(Tick integer);
create trigger T_CLOCK after insert on CLOCK
begin
  <step initialization>
  update AGENT set State = 'move';
  <step closure>
end;
```

Script 24.3 - The CLOCK table and its trigger

Its body itself creates an `update` event to send a message ("move", in this example) to the rows of the AGENT table. The value of this message as well as the (optional) initialization and closure sections of the body depend on the specific agent system.

Each simulator developed in this case study is available in two versions:

– the *Basic version* comprises the structures and algorithms discussed and developed in this chapter,

– the *Extended version* offers more parameters to provide detailed information on the simulation execution, such as traces of elementary actions, either to give the user a better understand of how the simulators work or for debugging purposes. The names of the procedures in these versions are suffixed with `'-extended'`. This code is of little interest for the purposes of this presentation and will not be described in detail.

24.3.3 Firering order

Because of the way the SQL engine works, the clock manager (Script 24.3) sends the same message to all agents *in a sequential order*. This order implementation dependent and is often the order of insertion or is defined by the values of the primary key: the first agent in this order will receive this message first, and the last agent will also receive it last. At each step of the simulation, it is highly likely that the same order will be followed, introducing an undesirable bias into the process.

There is a special form of the `update` query that allows a specific order to be imposed on its execution via an `order by` clause. In our case, this order is *random*, so that, for a large number of steps, all the agents tend to get an equal chance of being processed first (or last):

```
update AGENT set State = 'move' order by random();
```

However, this feature is not provided by all the SQL engines. It can be emulated by an SQLfast `for` loop:

```
for AiD = [select AgentID from AGENT order by random()]
   update AGENT set Message = 'move' where AgentID = :AiD;
```

or, in a more declarative way, by encapsulating the random order in a view[12]:

```
create view RANDOM_AGENT as
   select * from AGENT order by random();
create trigger R_AGENT
instead of update of Message on RANDOM_AGENT
```

12. A view being a (virtual) table, its rows are supposed to be unordered. While the concept of *ordered view* is a convenient extension, it violates the principles of the relational theory of databases.

```
begin
   update AGENT set Message = new.Message
   where  AgentID = old.AgentID;
end;
```

24.4 Agent behavior

An agent's behavior is defined by the actions it must perform in response to a certain type of message. Sending a message involves updating the Message column of the AGENT rows of the recipients concerned. In the first agent systems we are going to build, we will consider a single message, "move". In more complex agent systems, agents will have to react to several different messages.

24.4.1 SQL modeling

As explained in the *Introduction* section, the behavior of an agent is expressed in the form of ECA rules, the privileged implementation of which is via SQL triggers. Attached to the AGENT table, these triggers capture `after` events created by the updating the Message column of AGENT rows (`after update of Message on AGENT`). It fires under two conditions:

- the new value of Message is `'move'` (**`when`** `new.Message = 'move'`),
- (optionally) the current row satisfies a certain condition (`and <Condition>`),

The procedure of the trigger body is consists of a sequence of actions. Each action translates into an SQL query that can modify the state of the agent[13] or of the space of the world.

```
create trigger T_AGENT
after update of Message on AGENT
for  each row
when new.Message = 'move'
and  <Condition>
begin
    <Action 1>
    <Action 2>
    <...>
    <Action n>
end;
```

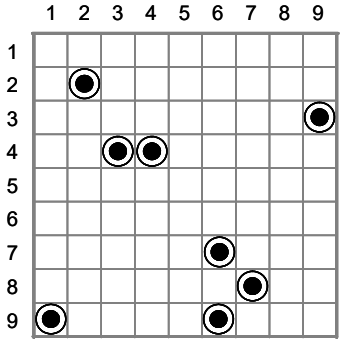
Script 24.4 - Implementing the behavior of agents via SQL triggers

13. or, in some agent systems, of the neighbors of the current agent

24.5 Graphical representation of the world

Graphical representation of the state of the world is an important function of agent-based simulators. In the family of models we are studying in this essay, we wish to identify each cell and each agent in this representation[14]. In the implementation of the different models, the space and its content will be displayed as a vector graphic image (Figure 24.1, left side) or as a simple text (right side). This figure shows a square space of 81 cells, 8 of which contain an agent.

Most simulators introduce additional properties (the gender of an agent, for example) and other objects (such as attractors and repellers). Their specific graphical representation will be described when needed.




```
  1 2 3 4 5 6 7 8 9
1 . . . . . . . . .
2 . 1 . . . . . . .
3 . . . . . . . . 1
4 . . 1 1 . . . . .
5 . . . . . . . . .
6 . . . . . . . . .
7 . . . . . 1 . . .
8 . . . . . . 1 . .
9 1 . . . . 1 . . .
```

Figure 24.1 - Two graphical representations of a 9x9 space

24.6 Five (not so) easy pieces[15]

We now have a generic *framework* from which to build a series of five agent models of increasing complexity.

- **The lawnmower** (Section 24.7): This model experiments with different trajectory styles. The world comprises a single agent whose sole task is to move according to one of these styles.
- **Aggregation** (Section 24.8): Now, the world includes several agents of the same type. These agents move according to one of the trajectory styles. When

14. This would make no sense in models based on a very large number of very small objects such as the one that represents each grain of sand by an agent to simulate the evolution of the sea shore as a function of tidal height. In such cases, the graphical representation comprises aggregates of the system, such as dm^3 or m^3 of sand.
15. Tribute to Bob Rafelson's eponymous movie (1970)

an agent visits a cell already hosting an agent, it forms an aggregate, which is a static object. If this cell contains an aggregate, the agent joins it.

– **Attractors and repellers** (Section 24.9): In addition to a collection of agents, the world includes static objects, *attractors* and *repellers*. Agents attempt to join one of the attractors, but try to avoid the repellers in their vicinity.

– **Agent lifecycle** (Section 24.12): In this model, agents behave like living entities: they have two genders, are born, grow old, mate, generate new agents and die.

– **Agent lifecycle with birth control** (Section 24.13): It appears that the life cycle model is intrinsically unstable: the population of agents tends to grow without limit and fill space, or, on the contrary, to decrease until extinction. In this model, we introduce a rule that links the fertility rate to the population size.

The Agent application (Agent-MAIN.sql) opens a selection box for the five simulators developed in this case study (Figure 24.2). Both *Basic* and *Extended* versions are available

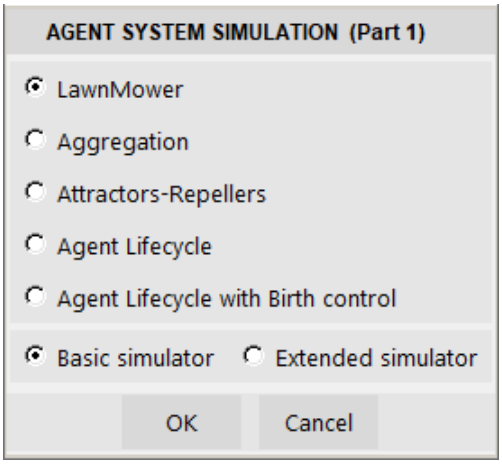


Figure 24.2 - The general selection box

24.7 Agents are moving: single-agent worlds

Although this first agent model may seem particularly simplistic, or even uninteresting, it will enable us to tackle and solve fundamental problems that will form the building blocks of all the models we develop in the rest of this study.

24.7.1 Trajectory styles

The common feature of the model we will build, agents *move around* in the space of the world. More precisely, at each step of the simulation, an agent can decide to occupy a cell in its direct environment (Figure 24.3).

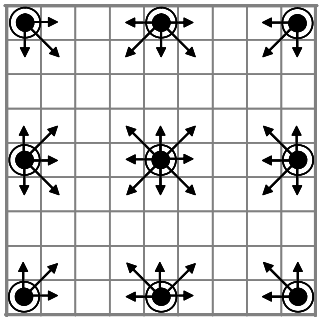


Figure 24.3 - The unconstrained valid moves of an agent

The choice the agent makes depends on its state, its goals and the characteristics of its current cell and of those of surrounding cells. It also depends on the style of trajectory the agent is trying to adopt, so that the question arises as to the shape of the trajectory.

We first consider two extreme styles: *random* and *linear*, called *Linear_0* and *Linear_5* respectively.

A. The random trajectory (*Linear 0*)

At each step of a *random trajectory*, the next position is selected at random from the surrounding cells. It is also possible that the agent decides to stay at its current position. The choice is therefore made from among nine cells. The screenshot on the left-hand of Figure 24.4 shows the random trajectory of an agent in an 8x24 space, where each visited cell is colored green.

B. The linear trajectory (*Linear 5*)

In a *linear trajectory*, on the other hand, each step extends the previous move (Figure 24.4, right-hand screenshot). On contact with a horizontal or vertical border of the space, the agent bounces off, theoretically at a symmetrical angle, as illustrated by its behavior on encountering the upper border. However, this rule risks leading to cyclical trajectories in which the agent occupies the same subset

of cells indefinitely. To counter this phenomenon, we introduce a random perturbation in the choice of the next cell during a bounce. This is illustrated by the agent's behavior when it encounters the left and bottom borders of the aforementioned figure.

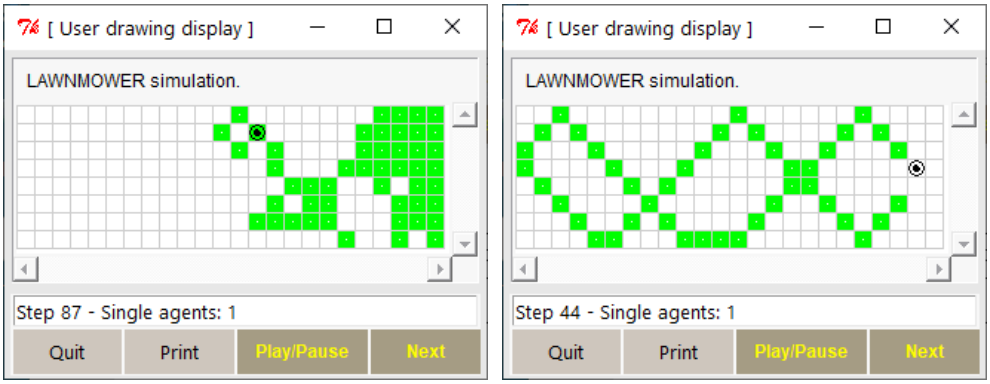


Figure 24.4 - Random (left) and linear (right) trajectories of an agent in a 8x24 world

C. Intermediate trajectories

Between these two trajectory styles, we define four variants of increasing linearity (and therefore decreasing *randomness*), denoted *Linear 1* to *Linear 4*. In each case, the next direction is chosen from a list in which the linearity (pure extension) is increasingly likely. In *Linear 1*, trajectories appear random, but with a slight tendency to extend the previous move. In *Linear 4*, trajectories are largely linear, but occasionally show a random change of direction.

These proposals raise the question of whether an agent (or, more generally, several agents) can cover the space according to each of the trajectory styles. If an agent is given enough time, will it eventually visit all the cells in the space, and if so, which trajectory style will be the fastest to achieve this goal? This is what a short series of experiments seem to show:

- whatever the dimensions of the space, all trajectory styles lead to total coverage of the space,
- not surprisingly, the last cells to be visited consume more and more steps; in particular, visiting the last cells may require as many steps as accessing all the previous ones,
- on the average (measured over some twenty trials for each), despite a very high variance, styles 3 and 4 cover the cells of the space the fastest.

This model could be used to simulate the behavior of a *lawnmower*[16]. However, the latter are generally supposed to operate more intelligently, which could translate into more elaborate trajectory styles!

24.7.2 SQL modeling

In this first study, we examine the behavior of a single agent whose sole aim is to move from one cell to one of its neighbors according to an imposed trajectory style. In brief:

- the space is finite and rectangular (bounded by four linear borders),
- only one agent is active, so no interaction with other agents, or with special objects in the space,
- the agent is located in a cell specified by its coordinates in the space,
- at each step of the simulation, the agent executes an elementary move to an adjacent cell,
- this move is determined by the selected trajectory style.

The travel of the agent is based on the following principle:

To execute the next move, the agent computes its new direction as a unitary vector[17] *(Δx, Δy) to add to its current position (X,Y). Then, the agent moves to the cell (X+Δx,Y+Δy).*

A. Data structures

In all the trajectory styles (but one, *Linear 0*), the next move depends on the current direction, that is, on the previous move. We store the previous (Δx,Δy) vector in new columns OldX,OldY (Script 24.5).

```
create table AGENT(AgentID integer not null primary key,
                   X       integer,
                   Y       integer,
                   OldX    integer default 0,
                   OldY    integer default 0,
                   Message varchar(6) default '');
```

Script 24.5 - Thanks to columns OldX and OldY the agent remembers the direction of its previous move

Besides its (X,Y) coordinates, each cell is described by two new properties: the number of agents it contains (Content column), and the trace of an agent's passage

16. Hence the name we give to this agent model
17. Here, by *unitary*, we mean that each of its coordinates is in the set (-1,0,1).

(Trace column) (Script 24.6). The latter column is set to 1 whenever an agent visits this cell.

```
create table CELL(X integer,
                  Y integer,
                  Content integer default 0,
                  Trace   integer default 0);
```

Script 24.6 - The cells keep the trace of the passage of the agents

B. Analysis of the agent behavior

The only task of the agent is to move across the space according to a definite style such as those illustrated in Figure 24.4.

The rules that define the trajectory styles rely on the concept of *vicinity.* We call *vicinity* - or *neighborhood* - of a cell (as well as of the agent it hosts) the set of its neighboring cells. In all the trajectory styles, the *next move* of an agent sends it in one of the cells of its vicinity. In some case, the next move may also be a *no-move*, that leaves the agent in its current cell.

At each step, each agent computes its next move as the vector (Δx, Δy) to add to its current position (X,Y). Then, the agent moves to cell (X+Δx,Y+Δy).

The computing rules of (Δx, Δy) depend on the trajectory style. However, whatever this style, the next move of the agent *when it has reached a border* sends it to one of the cells of its vicinity, randomly selected.

Now, we examine the rules that dictate the selection of the next move when the agent is *in the middle of the space*. We focus first on the two extreme styles, namely pure random (*Linear 0*) and pure linear (*Linear 5*)

- **The random trajectory (*Linear 0*)**

  The random trajectory is defined by a simple rule: the next move is randomly chosen from among the eight cells of its vicinity plus the current cell. It is therefore strictly independent of the previous move (the agent ignores its OldX and OldY properties). This rule is defined as follows for agents in a *non-border* cell[18]:

  ```
  (Δx,Δy) = random('-1,-1;-1,0;-1,1;0,-1;0,0;0,1;1,-1;1,0;1,1')
  ```

18. Used without argument, the `random()` function returns a random number (named `rand()` in some SQL versions). Used with an *string-list* argument, `random(S)` is a function that randomly selects an element of `S`. A *string-list* is a character string formed by a sequence of values separated by a specific symbol (*csv* is a popular *string-list* format). In this example, `S` serializes a list of couples '`x,y`' separated by semi-colons. This interpretation of function `random` is specific to SQLfast.

- **The linear trajectory (*Linear 5*)**

  In a linear trajectory (*Linear 5*), each move is a mere copy of the preceding one, defined by the properties OldX and OldY of the agent. So, for agents in a *non-border* cell:

  ```
  (Δx,Δy) = (OldX,OldY)
  ```

Before developing the intermediate styles, we solve the next step problem for the agent that has reached a border of the space.

- **Agent at the border: the bouncing rules**

  We distinguish eight current positions (see Figure 24.3), each requiring a specific reaction[19]:

  – top left corner: `X=1 and Y=1`
    `random('1,0;1,1;0,1')`
  – bottom left corner: `X=1 and Y=:height`
    `random('0,-1;1,-1;1,0')`
  – top right corner: `X=:width and Y=1`
    `random('-1,0;-1,1;0,1')`
  – bottom right corner: `X=:width and Y=:height`
    `random('-1,0;-1,-1;0,-1')`
  – left border, middle: `X=1 and Y not in (1,:height)`
    `random('0,-1;1,-1;1,0;1,1;0,1')`
  – right border, middle: `X=:width and Y not in (1,:height)`
    `random('0,1;-1,1;-1,0;-1,-1;0,-1')`
  – top border, middle: `X not in (1,:width) and Y =1`
    `random('1,0;1,1;0,1;-1,1;-1,0')`
  – bottom border, middle: `X not in (1,:width) and Y=:height`
    `random('-1,0;-1,-1;0,-1;1,-1;1,0')`

Let us now study the four intermediate trajectory styles, denoted *Linear 1* to *Linear 4*. We start with the *Linear 1* style, from which we will derive *Linear 2* to *Linear 4*.

- **The pseudo-random trajectory (*Linear 1*)**

  The agent selects its next step from three candidate directions:

  – extending the preceding step,
  – the next cell clockwise
  – the next cell counter-clockwise.

19. The suggested moves do not include the `(0,0)` *no-move*.

This rule is illustrated in Figure 24.5 for four representative cases (the other four are symmetrical). The large solid dot represents the current position of the agent, the incoming arrow its preceding step, and the outgoing arrows the three candidate next steps.

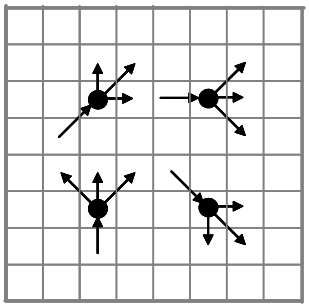


Figure 24.5 - Selection of the next step in the Linear 1 style

These patterns are translated into the rules of Figure 24.6. The remaining four patterns are specified similarly. We observe that the linear direction (in blue) is in the middle of the list.

(OldX,OldY)	(Δx,Δy)
(1,-1)	random('0,-1;**1,-1**;1,0')
(1,0)	random('1,-1;**1,0**;1,1')
(1,1)	random('1,0;**1,1**;0,1')
(0,-1)	random('-1,-1;**0,-1**;1,-1')

Figure 24.6 - Moving rules for the *Linear 1* trajectories

We still must address a special case, the *starting step*, for which there is no preceding step. By convention, the (OldX,OldY) properties are set to (0,0) and the next step is selected among the eight adjacent cells, as in the *Linear 0* style.

- **From pseudo-random to linear (*Linear 2* to *Linear-4*)**

  To augment the linearity trend of the pseudo-random style, we duplicate its linear component (which is a copy of the preceding step) in the candidate list (Figure 24.7). Now, the probability that the next step extends the preceding step is 0.5 (2 out of 4) instead of 0.33 (1 out of 3).

(OldX,OldY)	(Δx,Δy)
(1,-1)	random('0,-1;**1,-1;1,-1**;1,0')
(1,0)	random('1,-1;**1,0;1,0**;1,1')
(1,1)	random('1,0;**1,1;1,1**;0,1')
(0,-1)	random('-1,-1;**0,-1;0,-1**;1,-1')

Figure 24.7 - Moving rules for the *Linear 2* trajectories

By tripling, quadrupling, etc., this transformation, we build trajectories that look increasingly linear, such as *Linear 3* and *Linear 4*.

24.7.3 SQL code

The behavior of the agent is triggered by sending it a `'move'` message. This is performed by updating the Message column of the agent row:

```
update AGENT set Message = 'move';
```

Before examining the translation of this behavior into a SQL trigger, it's worth noting that, in the body of all the triggers we will develop in this study, the two aliases, *old* and *new*, denoting the states of the current row before (*old*) and after (*new*) the `update` operation are identical, with the exception of the value of the Message column. Consequently, for example, designating the current row identifier by the expressions "old.AgentID" or "new.AgentID" is strictly equivalent.

Architecture of the main trigger

The T_AGENT trigger is activated whenever the Message column of an AGENT row has been updated and when the new value is `'move'`. The reaction is decomposed into four successive tasks, each expressed as a single SQL query (Script 24.7).

Let us analyze each of these tasks.

```
create trigger T_AGENT
after update of Message on AGENT
for  each row
when new.Message = 'move'
begin
    <1. The agent quits its current position>
    <2. The agent computes its next direction>
    <3. The agent computes its next position>
    <4. The agent moves to its new position>
end;
```

Script 24.7 - Architecture of the main trigger

1. The agent quits its current position

The content of the current cell is decremented by one and this cell is marked to indicate that an agent once visited it.

```
update CELL
set    Content = Content - 1,
       Trace = 1
where (X,Y) = (old.X,old.Y);
```

Script 24.8 - The agent quits its current cell

2. The agent computes its next direction

This is the central activity of the agent. We suggest to design a generic query frame that provides a simple way to implement the six trajectory styles identified above (and any extensions of them).

Each style is defined by a series of lists of (Δx, Δy) directions. Each list specifies the recommended directions for one of the eight directions that has lead the agent in its current cell and that are stored in columns (OldX,OldY). For example, when the last directions was (1,1), the *Linear 2* strategy suggests to select the next direction in this list: `'1,0;1,1;1,1;0,1'`. If the agent selects the direction `'1,1'`, it will extend its preceding direction.

The specification of four of the six trajectory styles is shown in Script 24.9. A style is described by eight lists, one for each (OldX,OldY) couple.

These lists are stored in variables A to H. The special list in variable V (for *Vicinity*) is used to start the simulation, when columns (OldX,OldY) have not been set yet, and in the random trajectory style.

The current style is denoted by the `linear` variable. We note that the rules of *Linear 0* do not depend on the preceding direction and the rules of *Linear 5* strictly depend on the preceding direction only.

The generic query that computes the next position is structured as shown in Scripts 24.10 and 24.11. It returns in SQL variables DeltaX and DeltaY the direction the agent will apply according to the trajectory style and its current position in the space. It is instantiated by the values of these static variables:

– `width`, `height`: dimensions of the space

– V: vicinity of the agent (list of the eight directions of the vicinity of the agent, to which we add the *no-move* (0,0))

– A, B, C, D, E, F, G, H: list of candidate (Δx, Δy) directions according to the last step of the agent.

```
set V = '-1,-1;-1,0;-1,1;0,-1;0,0;0,1;1,-1;1,0;1,1';
if ($linear$ = 0);
   set A = $V$;
   set B = $V$;
   set C = $V$;
   set D = $V$;
   set E = $V$;
   set F = $V$;
   set G = $V$;
   set H = $V$;
endif;
if ($linear$ = 1);
   set A = '1,1;0,1;-1,1';
   set B = '-1,-1;0,-1;1,-1';
   set C = '1,-1;1,0;1,1';
   set D = '1,0;1,1;0,1';
   set E = '0,-1;1,-1;1,0';
   set F = '-1,1;-1,0;-1,-1';
   set G = '0,1;-1,1;-1,0';
   set H = '-1,0;-1,-1;0,-1';
endif;
if ($linear$ = 2);
   set A = '1,1;0,1;0,1;-1,1';
   set B = '-1,-1;0,-1;0,-1;1,-1';
   set C = '1,-1;1,0;1,0;1,1';
   set D = '1,0;1,1;1,1;0,1';
   set E = '0,-1;1,-1;1,-1;1,0';
   set F = '-1,1;-1,0;-1,0;-1,-1';
   set G = '0,1;-1,1;-1,1;-1,0';
   set H = '-1,0;-1,-1;-1,-1;0,-1';
endif;

...

if ($linear$ = 5);
   set A = '0,1';
   set B = '0,-1';
   set C = '1,0';
   set D = '1,1';
   set E = '1,-1';
   set F = '-1,0';
   set G = '-1,1';
   set H = '-1,-1';
endif;
```

Script 24.9 - Definition of trajectory styles *Linear 0*, *Linear 1*, *Linear 2* and *Linear 5*

```
select Next[1],Next[2] into DeltaX,DeltaY
from (select
        case
        -- bouncing moves
           when old.X in (1,$width$) or old.Y in (1,$width$)
           then <compute the bouncing step [Script 24.11]>
        -- starting move
           when (old.OldX,old.OldY) = (0,0)   then random($V$)
        -- internal moves
           when (old.OldX,old.OldY) = (0,1)   then random($A$)
           when (old.OldX,old.OldY) = (0,-1)  then random($B$)
           when (old.OldX,old.OldY) = (1,0)   then random($C$)
           when (old.OldX,old.OldY) = (1,1)   then random($D$)
           when (old.OldX,old.OldY) = (1,-1)  then random($E$)
           when (old.OldX,old.OldY) = (-1,0)  then random($F$)
           when (old.OldX,old.OldY) = (-1,1)  then random($G$)
           when (old.OldX,old.OldY) = (-1,-1) then random($H$)
        end as Next
     );
```

Script 24.10 - Computing the next direction

```
case
   when old.X=1 and old.Y=1 then random('1,0;1,1;0,1')
   when old.X=1 and old.Y=$height$ then random('0,-1;1,-1;1,0')
   when old.X=$width$ and old.Y=1 then random('-1,0;-1,1;0,1')
   when old.X=$width$ and old.Y=$height$
   then random('-1,0;-1,-1;0,-1')
   when old.X=1 and old.Y not in (1,$height$)
   then random('0,-1;1,-1;1,0;1,1;0,1')
   when old.X=$width$ and old.Y not in (1,$height$)
   then random('0,1;-1,1;-1,0;-1,-1;0,-1')
   when old.X not in (1,$width$) and old.Y =1
   then random('1,0;1,1;0,1;-1,1;-1,0')
   when old.X not in (1,$width$) and old.Y=$height$
   then random('-1,0;-1,-1;0,-1;1,-1;1,0')
end
```

Script 24.11 - Computing the bouncing steps

The query frame comprises three sections:

– The first section defines the behavior of the agent when it has reached a border of the space. It is developed in Script 24.11, which translates the bouncing rules

suggested above (title *Agent at the border: the bouncing rules*). This behavior is common to all the trajectory styles.

– The second section specifies the starting behavior of the agent. The first step is selected in the agent vicinity. It, too, is common to all the trajectory styles.

– The third section states the eight rules of which the Figures 24.6 and 24.7 are excerpts. These rules select at random one of the valid directions for each preceding move (old.OldX,old.OldY).

Remarks

The variables appearing in SQL queries and, more generally, in the algorithms we develop in this study are of two kinds: *static* and *dynamic*.

A *static variable* is a parameter of the world that is set before this world is built and remains constant during the simulation. The dimensions of space, width and height, are such variables. When used in the definition of the world, they appears as `$<variable>$` anywhere in a query or in an algorithm. For example, if variable `width` has been set to `24`, the expression "`where X = $width$`" must be read "`where X = 24`" throughout the simulation run.

A *dynamic variable* describes the instantaneous state of an object during the simulation. Its value can change during the execution of the current step and is usually computed by an SQL query. When used in a query, it appears in the standard SQL syntax `:<variable>`. Assuming that the future direction of an agent has been stored in variables `DeltaX` and `DeltaY`, the clause "`set X = X + :DeltaX`" of an `update` query computes the new `X` coordinate of this agent.

The queries of Scripts 24.10 and 24.11 are comprised in the body of the agent trigger and are executed during each simulation run, while the algorithm of Script 24.9 is executed at the initialization stage, when the world is built.

3. The agent computes its next position

The new direction is added to the agent current position and saved in the OldX and OldY columns. The Message column, no longer needed, is cleared (Script 24.12).

```
update AGENT
set    X = X + :DeltaX,
       Y = Y + :DeltaY,
       (OldX,OldY) = (:DeltaX,:DeltaY),
       Message = ''
were   AgentID = old.AgentID;
```

Script 24.12 - Computing the next position of the agent

4. The agent moves its new position

The content of the cell defined by the new position of the agent is incremented by one (Script 24.13). Note that (new.X,new.Y) do not denote the new position of the agent. Indeed, these values refer to the old and new states of the agent *when the Message column was updated*. To get the *current values* of X and Y, we must extract them from the database[20].

```
update CELL
set    Content = Content + 1
where (X,Y) = (select X,Y from AGENT
               where  AgentID = old.AgentID);
```

Script 24.13 - The new cell of the current agent is updated

Toward a smarter lawnmower

In an earlier section, we mentioned that the last cells to be visited consume more and more steps. As an example, the state of the world illustrated in Figure 24.8 has been reached in 620 steps. However, visiting the remaining six empty cells will require no less that 600 additional steps! Moreover, observing the movements of the agent is particularly frustrating. Quite often, its vicinity includes an empty cell that the agent blindly ignores in its next moves, missing the opportunity to reach its goal more quickly. Not to mention, in multi-mower configurations, the agents that roll over each other when they occupy the same cell!

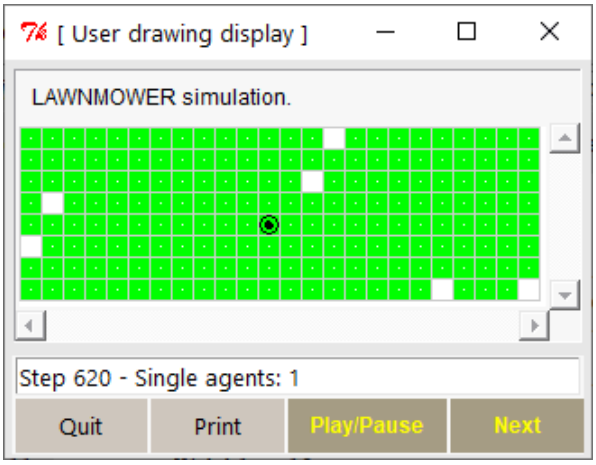


Figure 24.8 - State of the world after 620 steps.

20. or recalculate them as we did in Script 24.12, which would be less *SQL-minded*!

Could we improve the agent's behavior (its *intelligence*) by enabling it to consult its environment and, if necessary, to redirect its trajectory towards an empty cell? Certainly a good idea, as long as this process is only activated when the space contains only a small number of empty cells. Otherwise, the notion of trajectory style would become meaningless.

24.7.4 Experimentation: *The Lawnmower*

The control panel of the *Basic* module devoted to single moving agent (aka *Lawnmowers*) is shown in Figure 24.9. It comprises two elementary boxes.

The first box (1. STARTING PARAMETERS OF THE WORLD) defines the space dimensions, the maximum number of steps of the simulation, the delay between steps (in ms.) in the animated graphical representation and the trajectory style (from 0 to 5).

The second box (2. THE LAWNMOWERS) is a *bonus feature*; it allows the experimenter to create more than one agent. Note that, in this case, agent can run over another without damaging it!

A. The *Basic* simulator

The basic *LawnMower simulator* is implemented through the script **MAIN-LawnMower**. It calls 7 procedures that, together, define the agent system:

- _AGENT-Init-Parameters: assign initial values to agent system parameters.
- _AGENT-Create-Tables: creates of the tables.
- _AGENT-Get-Parameters: asks the user the parameters of the agent system.
- _AGENT-Create-World: creates the triggers that specify the behavior of the agents.
- _AGENT-Populate-World: initializes the world by placing an agent (or several agents) in random cell(s).
- _AGENT-Generate-Current-State: when a step is finished, writes in file AGENT.draw the commands that draw the content of the cells.
- _AGENT-Process-History: displays the evolution of the agent system as an animated movie.

Three ancillary procedures are independent of the agent system:

- _AGENT-Trajectories, called by _AGENT-Create-World, initializes the set of variables V, A, B, C, D, E, F, G, H containing the lists of candidate directions as defined in Script 24.9.

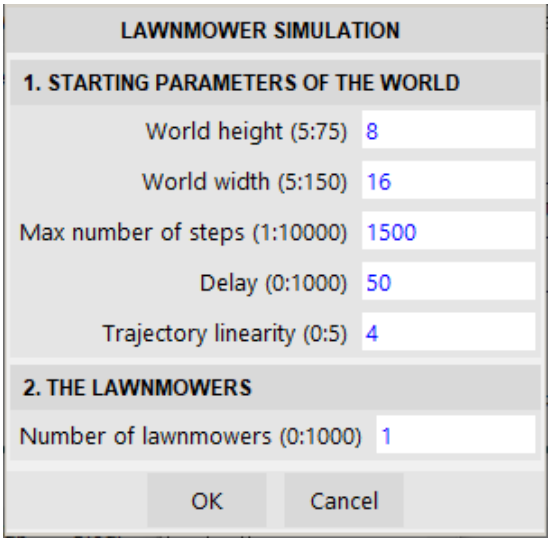


Figure 24.9 - The control panel of the basic *Lawnmower* simulator

– _AGENT-Generate-Grid, called by _AGENT-Populate-World, writes in file AGENT.draw the commands drawing the lines that delimit the cells.

– _AGENT-Clear-Tables: empties the base tables.

B. The *Extended* simulator

The *Extended* version of the simulator includes additional functions by which the experimenter can be provided with more detail on the simulations.

In particular, it records a synthesis of the state of the world after each step in the JOURNAL table, structured as follows:

```
create table JOURNAL(StartTime timestamp,
                     Step      integer,
                     Traces    integer);
```

Each record comprises the timestamp of the start of the simulation, the index of the step and the cumulative number of cells visited. Once completed, this table can be queried or represented graphically.

The control panel of the *Extended* version is shown in Figure 24.10. It includes 5 elementary boxes, two of which are the same as those of the *Basic* version: (STARTING PARAMETERS OF THE WORLD) and (THE LAWNMOWERS). The three new boxes offers the following features.

– 2. WORLD CREATION MODE: this box lets the user choose whether to create a new starting configuration (defining the number and initial position of the agents) or to reuse the previous one.
– 3. INFORMATION: this box specifies the tracing information to print in the SQLfast output window. It can be used to observe the detailed behavior of each agent at each step or to debug a model.
  • Show start state: print the state of the AGENT table before the simulation.
  • Show final state: print the state of the AGENT table when the simulation is finished.
  • Show grid: at the end of each step of the simulation, print the state of the cells; the format is that shown in Figure 24.1/right-hand side; this feature is useful for small spaces, up to 10x10.
  • Trace moves: print the detail of each move of each agent in each step.
  • Show AGENT table: at the end of each step of the simulation, print the state of the AGENT table.
  • Graphic animation: display the animated graphical representation of the simulation.
– 5. DISPLAY JOURNAL: the last box is used to write the contents of the JOURNAL table in various formats to the output window. At the end of each step, the controller inserts into this table either the cumulative number of cells visited, or the density of these cells (percentage of cells visited).
  • No, thank you: no output.
  • Full tabular: content of the JOURNAL table in a standard tabular layout.
  • Step + density CSV: at the end of each step, density (percentage) of cells visited; CSV format with *tab* delimiters.
  • Full CSV: content of the JOURNAL table in CSV format with *tab* delimiters.

The application code of the *Extended* simulator

The *Extended LawnMower simulator* is implemented by the script **MAIN-Lawn-Mower-extended**. It calls 7 procedures similar to those of the *Basic* version:

– _AGENT-Init-Parameters-extended
– _AGENT-Create-Tables-extended
– _AGENT-Get-Parameters-extended
– _AGENT-Create-World-extended
– _AGENT-Populate-World-extended
– _AGENT-Generate-Current-State-extended
– _AGENT-Process-History-extended

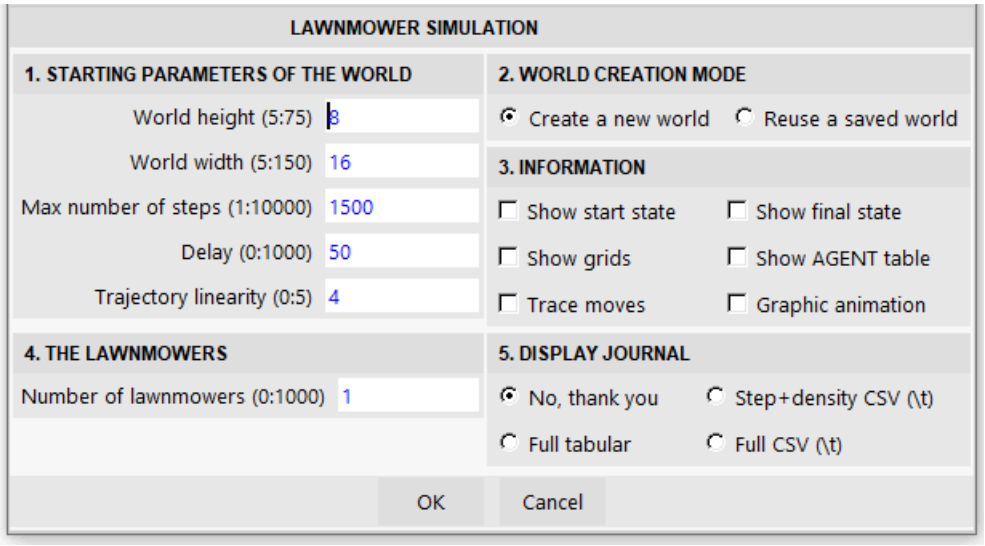


Figure 24.10 - The control panel of the extended *Lawnmower* simulator

Four ancillary procedures: _AGENT-Trajectories, _AGENT-Generate-Grid, _AGENT-Clear-Tables and Display-CELL-extended (called by INFORMATION function Show grid).

A first conclusion

We observe that the agent's behavior is entirely defined in the trigger body via just 4 SQL queries. This first experiment seems to validate the choice of database concepts and technology for implementing agent-based models and systems.

We have developed a first primitive but useful skill for agents: *moving in space*. We will build on this skill in the following experiments.

24.8 A first multi-agent world: *Aggregation*

In this second essay, the world hosts several agents of the same type and allows them to interact, albeit still in a very basic mode. Agents are randomly distributed in the space, then start to move according a predefined trajectory style. When two agents meet in the same cell, they assemble to form a new object, an *aggregate*. Similarly, when an agent moves to a cell containing an aggregate, it joins the latter, augmenting it by one unit. Aggregates are static and stay in their cell until the completion of the simulation. When an agent joins (or forms) an aggregate, it becomes inactive and stop moving.

The simulation ends when each agent has joined an aggregate, which can be considered the *objective* of the agent system.

24.8.1 SQL modeling

A. Space structure

The Content property of a cell indicates the number of agents it contains. Now, a cell is in one of three states depending on its content:

- Content = 0: empty
- Content = 1: contains one moving agent
- Content > 1: contains a static aggregate of size Content.

B. Analysis of the agent behavior

The way an agent reacts when activated is similar to that of the single-agent case studied in Section 24.7. Two differences however in the activation process: the definition of the set of agents to activate and the order in which the agents are activated.

The agents that form an aggregate must be discarded from the activation process. So, this process concerns only the agents that are alone in their cell.

During the execution of a step, all the agents are activated. To this aim, each of them receives a message ('move') instructing it to move. Though the sending of the message to all the agent translates into a single query and therefore appears as a global, set-oriented operation, we know that, internally, the agents will be processed sequentially, one at a time, according to an order which depends on the implementation technique of the table. The consequence is that, in the successive steps, the agents are likely to be activated in *the same order*, which creates a significant bias that can invalidate the evolution simulation. To overcome this problem, we suggest sending the activation messages to the agents in a random order.

24.8.2 SQL code

The extensions to multi-agent worlds of the code developed in Script 24.7 are shown in Script 24.14:

- the agents are activated in a random order (`update ... order by random()`),[21]
- the "`when`" clause of the trigger delimits the category of agents that will respond to the event: those located in a cell containing exactly one agent,
- it is no longer necessary to leave a trace of every cell visited.

```
-- The triggering event
update AGENT set Message = 'move' order by random();
...
-- The reaction
create trigger T_AGENT
after update of Message on AGENT
for  each row
when new.Message = 'move'
     and (select Content from CELL where (X,Y)=(new.X,new.Y)) = 1
begin
    <moving the agent [same as Script 24.7]>
end;
```

Script 24.14 - The trigger that controls the behavior of the agents: only moving agents are concerned

Installing the initial population

The creation of the initial set of agents is done in two steps (Script 24.15). First, the *content* property of a certain number of cells is set to 1, indicating that an agent is present, while the others are set to 0. This number is defined by the *density* of the agent population within the space. This density is specified through the density variable, set from 1 to 100 by the user. The decision to create an agent in a cell is implemented by the following expression, where `random_i(i1,i2)`[22] returns a random integer between `i1` and `i2`:

```
random_i(1,100) <= $density$
```

For instance, if density = 10, this expression is expected to be True for about 10% of the cells.

21. When this study was published, SQLite did not yet implement this form. A workaround would be to create a view cloning the AGENT table with an `order by random()` clause, then to send the message to the rows in that view.
22. The syntax depends on the DBMS. In SQLfast, `random_i` is defined as a UDF wrapping the `random()` SQLite core function.

Then, an AGENT row is inserted for each cell the Content of which is 1.

```
with
   CX(X,Y) as (select 1,1
                 union all
               select X+1,1 from CX where X < $width$),
   CY(X,Y) as (select X,Y from CX
                 union
               select X,Y+1 from CY where Y < $height$)
   insert into CELL(X,Y,Content)
        select X,Y,case when random_i(1,100) <= $density$
                        then 1 else 0
                   end
        from CY;
insert into AGENT(X,Y) select X,Y from CELL where Content = 1;
```

Script 24.15 - Creating the initial state of the agent population

Graphical rendering

The graphical representation of the result of each step must show both agents and aggregates with specific forms and colors, such as those suggested in Figure 24.11 that inform on the content of a cell, and, therefore, on the size of the aggregates (from 2 to 6+).

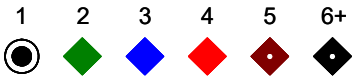


Figure 24.11 - Color codes for the different cell contents

The Figure 24.12 shows the evolution of a world in which 28 agents move according to a *Linear 1* trajectory and eventually generate 9 aggregates.

24.8.3 Experimentation: the *Aggregation* simulator

The simulator is available in two versions, *Basic* and *Extended*.

The control panel of Figure 24.13 can be used to experiment with the *Basic* version of the *Aggregation module* of the Agent application.

The parameters are similar to those of the *Lawnmower module* (Figure 24.9) except for the new parameter Agent density, through which one specifies the percentage of cells that contain an agent at start time.

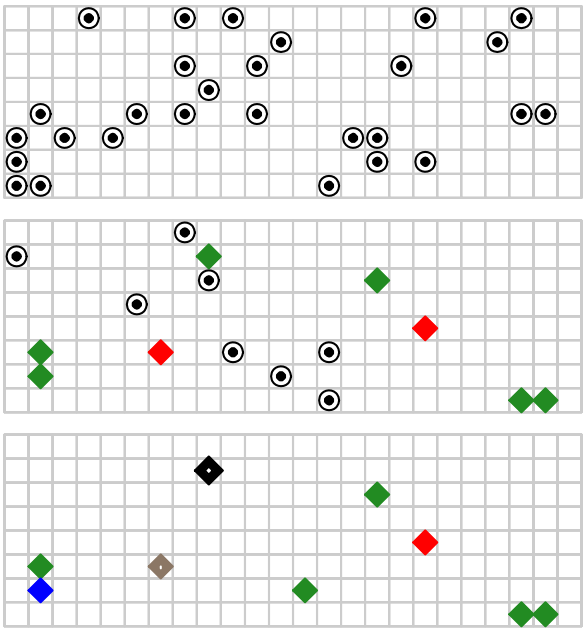


Figure 24.12 - Three representative states of the world: start (top), at step 10 (center) and final (bottom)

The application code of the simulator

The basic *Aggregation* simulator is implemented through the script **MAIN-Aggregation**. It calls the same 7 procedures as the *Lawnmower* simulator that define the agent system:

- _AGENT-Init-Parameters
- _AGENT-Create-Tables
- _AGENT-Get-Parameters
- _AGENT-Create-World
- _AGENT-Populate-World
- _AGENT-Generate-Current-State
- _AGENT-Process-History

The *Extended* simulator

In the same way as we did for the *Lawnmower* simulator, the *Extended* version of the *Aggregation* simulator includes specific functions by which the experimenter can be provided with more control and more detail on the simulations.

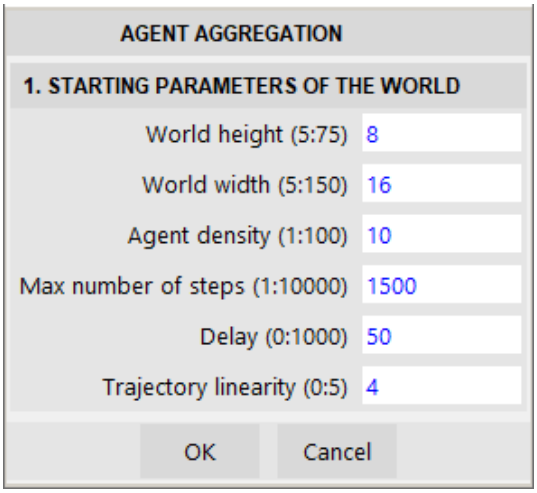


Figure 24.13 - The control panel of the basic *Aggregation* simulator

The synthesis of the state of the world after each step is recorded in the JOURNAL table, structured as follows:

```
create table JOURNAL(StartTime timestamp,
                     Step      integer,
                     Nagent    integer default 0,
                     Agreg2    integer default 0,
                     Agreg3    integer default 0,
                     Agreg4    integer default 0,
                     Agreg5    integer default 0,
                     Agreg6    integer default 0,
                     AgregMax  integer default 0);
```

Each record comprises the timestamp of the start of the simulation, the index of the current step, the number of remaining independent agents and the number of aggregates of 2, 3, 4, 5, 6 and more agents.

The control panel of the *Extended* version comprises three additional elementary boxes similar to those of the *LawnMower* simulator.

Observation

We observe that, once the problem of controlling agent moves has been mastered, extending the *Lawmower* model to agent aggregation just requires adapting the **when** clause of the **T_AGENT** trigger.

24.9 Attractors and repellers

In the first two experiments, the agents have not proved to be particularly intelligent. They just blindly follow the rules of their trajectory style, but are not even aware that they are forming aggregates. They never consult their environment to decide on their subsequent actions (with the sole exception of the bouncing rules).

In this section, we consider objects that agents can encounter and interact with, either positively or negatively. Interactions consist, for these new objects, in attracting or repelling agents moving in their vicinity. Agents will therefore try to join nearby *attractors* and to avoid *repellers*. For example, an attractor can be any resource, such as food or medicine, and a repeller a pit or a trap.

For now, we decide that attractors and repellers are not agents but fixed, static objects. In more complex agent models, we could think to assign the roles of attractor and repeller to active agents, such as potential partners or predators.

24.10 Agents and attractors

For the sake of simplicity, we choose to represent attractors by the aggregates we described in section 24.8. As soon as two agents meet in the same cell, they create an aggregate which, from that moment on, will attract and absorb any other agent moving nearby. We will consider two forces of attraction, *close* and *distant*. Near attraction applies between the attractor and an agent in its vicinity, i.e. when their distance is one cell. Distant attraction applies when the agent is close to the attractor's neighborhood, i.e. when their distance is two cells.

24.10.1 SQL modeling of attractors

A. Data structure

No new features needed.

B. Analysis of the agent behavior

The attraction is active in the eight neighboring cells of the vicinity of an attractor. If an agent enters the vicinity of such an object, it is forced, or invited, to join the latter. Considering the way attractors are created, any cell the content of which is greater than 1 comprises an attractor.

If an agent is in the vicinity of one or several attractors, then, conversely, these attractors are comprised in the vicinity of this agent. Therefore, an agent is under the influence of attractor(s) whenever its vicinity includes at least one cell with a content greater than 1. The Figure 24.14 shows a fragment of the world in which an agent just moved from cell (6,6) to cell (7,5), therefore entering in the vicinity of

two attractors, located in (6,4) and (8,5). Once these attractors have been identified, the agent joins one of them, randomly selected.

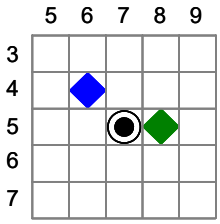


Figure 24.14 - The vicinity of current agent (7,5) includes two close attractors

Distant attractors are illustrated in Figure 24.15. Though the direct vicinity of the current agent (its 8 surrounding cells) does not include any attractor, there are cells in its vicinity that would place the agent next to an attractor and that are worth to move to. Let us imagine that the attractor provides some food, that this food smells particularly good (perceptible at a 2-cell distance) and that the agent is desperately hungry. It's not unreasonable to imagine that this agent tries to get closer to this *distant* attractor in order to be able to profit from its resource in the next step. So, when this agent is asked to move (`update AGENT set Message = 'move'`), its next move will be among the cells marked with a question mark in Figure 24.15 and not to those with an X mark.

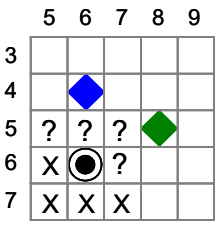


Figure 24.15 - Recommended (**?**) and invalid (**X**) moves of the current agent trying to *get closer* to a distant attractor

24.10.2 SQL code for attractors

A. SQL code for close attractors

When the current agent is activated, it is located in cell (`old.X`,`old.Y`). Its vicinity is computed as follows:

```
select X,Y
from   CELL
where  X between old.X - 1 and old.X + 1
```

```
and    Y between old.Y - 1 and old.Y + 1
and    (X,Y) <> (old.X,old.Y)
```

From the 8 cells of this area, we derive the coordinates of the attractors of the agent:

```
select X,Y
from   CELL
where  X between old.X - 1 and old.X + 1
and    Y between old.Y - 1 and old.Y + 1
and    Content > 1;
```

Two remarks

We have not kept the specific, but redundant, condition that discards the current cell (`(X,Y) <> (old.X,old.Y)`) since its content is not greater than 1.

We don't need to worry about neighboring cells beyond the space boundary, as they are not represented in the AGENT table.

This query produces a *set* of cell coordinates, for instance {(6,4),(8,5)} in the example of Figure 24.14, from which we must choose the attractor that will absorb the current agent. We could choose the biggest attractor, that with the largest content, but, for now, let us leave this selection to chance!

We convert, through the group_concat function, the set of attractor coordinates into a *string-list* (i.e., `'6,4;8,5'`), on which we can further apply the random function. The query of Script 24.16 translates this conversion, in which the restriction on the content of the cells has been converted into a more elegant *filter* condition[23].

```
select group_concat(X||','||Y ,';')
       filter (where Content > 1) as Att
from   CELL
where  X between old.X - 1 and old.X + 1
and    Y between old.Y - 1 and old.Y + 1;
```

Script 24.16 - Looking for the close attractors of the agent in cell (old.X,old.Y)

Actually, we are interested in the relative coordinates of the valid cells, that is, their directions. In addition, considering that the group_concat function returns a *null* value when applied to an empty set, we convert this value into an empty list with the coalesce function (Script 24.17).

23. The `filter` clause is associated with an aggregate or window function. It limits the set of rows on which the aggregate function is computed. While the same condition in the `where` clause of the query applies to all the aggregate functions, the filter clause applies to one function only.

```
select coalesce(group_concat((X-old.X)||','||(Y-old.Y) ,';')
                filter (where Content > 1), '') as Att
from   CELL
where  X between old.X - 1 and old.X + 1
and    Y between old.Y - 1 and old.Y + 1);
```

Script 24.17 - Computing the valid moves to the close attractors of the agent in cell (old.X,old.Y) - Coping with empty sets

Once an attractor has been selected, the agent moves to its cell to be absorbed.

B. SQL code for distant attractors

Let's go back to Figure 24.15. We postulate that the (close) vicinity of the current agent does not comprise an attractor, otherwise, the search for distant attractors would be useless.

We observe that the area of potential distant attractors is a square ring at a 2-cell distance from (`old.X`,`old.Y`). This square has four sides, N, S, E, W, with the following coordinates:

- **N side**: `old.X` - 2 ≤ X ≤ `old.X` + 2 and Y = `old.Y` - 2
- **S side**: `old.X` - 2 ≤ X ≤ `old.X` + 2 and Y = `old.Y` + 2
- **W side**: X = `old.X` - 2 and `old.Y` - 2 ≤ Y ≤ `old.Y` + 2
- **E side**: X = `old.X` + 2 and `old.Y` - 2 ≤ Y ≤ `old.Y` + 2

More synthetically,

- **horizontal sides**: abs(X - `old.X`) ≤ 2 and abs(Y - `old.Y`) = 2
- **vertical sides**: abs(Y - `old.Y`) ≤ 2 and abs(X - `old.X`) = 2

This translates easily in the following query which, in addition finds the attractors within this area (`Content > 1`),

```
select X,Y
from   CELL
where  (   (abs(X - old.X) <= 2 and abs(Y - old.Y) = 2)
        or (abs(Y - old.Y) <= 2 and abs(X - old.X) = 2))
and    Content > 1;
```

Now, we describe the *free cells* of the vicinity of these attractors. The query of Script 24.18 selects the empty cells in the vicinity of all the distant attractors of the current agent. The **Att** alias denotes the set of distinct attractors. The **Vatt** alias, joined to **Att**, describes the vicinity of each of these attractors.

```
select distinct Vatt.X,Vatt.Y
from  (select X,Y
       from   CELL
       where  (   (abs(X - old.X) <= 2 and abs(Y - old.Y) = 2)
              or (abs(Y - old.Y) <= 2 and abs(X - old.X) = 2)
              )
       and    Content > 1) Att,
       CELL Vatt
where  Vatt.X between Att.X - 1 and Att.X + 1
and    Vatt.Y between Att.Y - 1 and Att.Y + 1
and    Vatt.Content = 0;
```

Script 24.18 - Selecting the moves to the free cells in the vicinity of the distant attractors of the agent in cell (old.X,old.Y)

Finally, we compute the valid moves of the current agent to the vicinity of its distant attractors. The destination cells are defined as the set of common empty cells in the vicinity of the attractors and the vicinity of the agent. Expressing the coordinates of these cells (actually their relative coordinates ($\Delta x, \Delta y$)) in a single SQL query deserves a bit of explanation (Script 24.19). The main query computes the intersection of two sets of free cells expressed as two string-lists, **AttVic**, the merged vicinity of all the distant attractors, and **CurVic**, the empty vicinity of the current agent:

```
intersect(AttVic,CurVic)
```

The set **AttVic** has been computed in Script 24.18 while the set **CurVic** derives from the query of Script 24.17.[24]

C. Extension of the T_AGENT trigger

The overall structure of the trigger that specifies the behavior of the agents, and, in particular the `when` filter, are similar to those of the aggregation scenario (Script 24.20).

The content of the body is extended to apply the new rules governing the attraction phenomenon. The reasoning leads to a hierarchy of rules: first, the agent looks for close attractors; if none, it tries to find distant attractors. Finally, if no attractor has been found, it applies the rules of its trajectory style.

The updated version of the query that computes the next direction ($\Delta x,\Delta y$) is shown in Script 24.22.

24. One of the two conditions "`Vatt.Content = 0`" and "`Content = 0`" can be dropped. However we have kept both to make the query more readable.*

```
select intersect(AttVic,CurVic) as ValMov
from (select
            coalesce(group_concat((Vatt.X-old.X)||','
                                  ||(Vatt.Y-old.Y),';'),
                     '') as AttVic
      from  (select X,Y
             from   CELL
             where  (   (abs(X-old.X) <= 2 and abs(Y-old.Y) = 2)
                     or (abs(Y-old.Y) <= 2 and abs(X-old.X) = 2)
                    )
             and    Content > 1) Att,
             CELL                Vatt
      where Vatt.X between Att.X - 1 and Att.X + 1
      and   Vatt.Y between Att.Y - 1 and Att.Y + 1
      and   Vatt.Content = 0
     ),
     (select
            coalesce(group_concat((X-old.X)||','
                                  ||(Y-old.Y) ,';'),
                     '') as CurVic
      from  CELL
      where X between old.X - 1 and old.X + 1
      and   Y between old.Y - 1 and old.Y + 1
      and   Content = 0
     );
```

Script 24.19 - Computing the valid moves to the vicinity of distant attractors

```
create trigger T_AGENT
after update of Message on AGENT
for  each row
when new.Message = 'move'
     and (select Content from CELL where (X,Y)=(old.X,old.Y)) = 1
begin
    <moving the agent in presence of attractors>
end;
```

Script 24.20 - The trigger that controls the behavior of the agents when the space comprises attractors

```
-- 1. The agent looks for close attractors            --
    set V2 = <Script 24.17>
-- 2. The agent looks for for distant attractors      --
    set V3 = <Script 24.19>
-- 3. The agent chooses its closest attractor         --
    set V1 = case when :V3 = '' then :V2 else :V3 end;
-- 4. The agent quits its current position            --
update CELL set Content = Content - 1
where  (X,Y) = (old.X,old.Y);
-- 5. The agent computes its next direction           --
select item(Next,1,','),item(Next,2,',') into DeltaX,DeltaY
from   (<compute the (Δx,Δy) direction [Script 24.22]>);
-- 7. The agent computes its next position            --
update AGENT
set    X = X + :DeltaX, Y = Y + :DeltaY,
       OldX = :DeltaX, OldY = :DeltaY,
where  AgentID = old.AgentID;
-- 8. The agent moves to its next position           --
update CELL set Content = Content + 1
where  (X,Y) = (select X,Y from AGENT
                where  AgentID = old.AgentID);
```

Script 24.21 - Algorithm driving the behavior of the agents in presence of close and distant attractors

Just by curiosity

We have compared the speed (number of steps) of the three aggregation techniques to aggregate all the 128 agents of a representative world comprising 20x60 cells:

– pure aggregation, no attractors: 125 steps

– close attractors only: 50 steps

– close and distant attractors: 30 steps.

Each result is the average of the 17 central values provided by 21 simulation runs (the two lower and higher values being discarded).

24.10.3 Experimentation: the *Attractors-Repellers* simulator

The control panel of Figure 24.18 can be used to experiment with the *Attractors module* of the Agent application.

The parameters are similar to those of the *Aggregation module* (Figure 24.13). The simulator offers three modes: pure *aggregation*, *close attractors* and *distant attractors*.

```
select
   case
   -- an attractor has been found
      when :V1 <> '' then random(:V1)
   -- bouncing moves
      when X in (1,$width$) or Y in (1,$width$)
      then <compute the bouncing step [Script 24.11]>
   -- starting move
      when (OldX,OldY) = (0,0)   then random($V$)
   -- internal moves
      when (OldX,OldY) = (0,1)   then random($A$)
      when (OldX,OldY) = (0,-1)  then random($B$)
      when (OldX,OldY) = (1,0)   then random($C$)
      when (OldX,OldY) = (1,1)   then random($D$)
      when (OldX,OldY) = (1,-1)  then random($E$)
      when (OldX,OldY) = (-1,0)  then random($F$)
      when (OldX,OldY) = (-1,1)  then random($G$)
      when (OldX,OldY) = (-1,-1) then random($H$)
   end as Next
from  AGENT
where AgentID = old.AgentID;
```

Script 24.22 - Extended version of Script 24.10 to cope with attractors

24.11 Agents and repellers

Similarly to attractors, we consider two forces of repulsion, *close* and *distant*. Near repulsion applies between a repeller and an agent in its vicinity, i.e. when their distance is one cell. Distant repulsion applies when the agent is close to the attractor's neighborhood, i.e. when they are at a 2-cell distance.

24.11.1 SQL modeling of repellers

A. Data structure

Unlike attractors, repellers cannot be derived from the meeting of agents. They must be installed in the initial state of the space as specific objects. In the graphical representation, they will appear as black diamond.

B. Analysis of the agent behavior

Avoiding *close repellers* consists, for the current agent, in ignoring any move to the cells of its repellers. This means that the valid move of the agent are to any cell of its vicinity, except to the cells of its repellers (Figure 24.16).

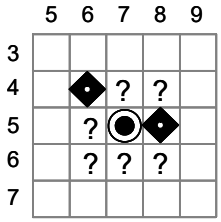


Figure 24.16 - The recommended moves (noted **?**) of current agent (7,5) trying to avoid its close repellers

The goal of trying to escape a *distant repeller* can also mean avoiding any future move that will place the agent in the vicinity of this repeller.

The world extract of Figure 24.17 shows the cells to avoid (marked with an X) and those that the agent must move to (marked with a ?).

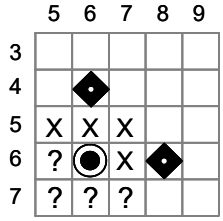


Figure 24.17 - Valid moves of the agent in cell (6,6) to get away from the *proximity* of its distant repellers

Interestingly, this figure is the exact opposite of Figure 24.15, in which the X and ? cells have been swapped. The valid destination cells are those of the vicinity of the current agent *except* those of the vicinity of the distant repellers.

24.11.2 SQL code for repellers

In this implementation, two agents are allowed to share the same cell while keeping their autonomy, without merging into an aggregate.

A. SQL code for close repellers

We decide to designate the cells of the repellers with the value Content = 9. Identifying the moves to safe cells in the vicinity of an agent is straightforward (Script

24.23). From the safe cells of Figure 24.16 (`'7,4;8,4;6,5;7,5;6,6;7,6;8,6'`), this query returns the six safe moves:

```
'0,-1;1,-1;-1,0;0,0;-1,1;0,1;1,1'
```

```
select coalesce(group_concat((X-old.X)||','||(Y-old.Y),';')
                filter (where Content <> 9), ''))
from   CELL
where  X between old.X - 1 and old.X + 1
and    Y between old.Y - 1 and old.Y + 1);
```

Script 24.23 - The valid moves of the agent in cell (`old.X`,`old.Y`) that *avoid* its close repellers

B. SQL code for distant repellers

The query derives from the algorithm for the distant attractors shown in Script 24.19, in which the set operator `intersect` is replaced by `except` (Script 24.24).

```
select except(CurVic,RepVic)
from (select coalesce(group_concat((Vrep.X-old.X)||','
                                   ||(Vrep.Y-old.),';'),
                      '') as RepVic
      from (select X,Y
            from   CELL
            where  (   (abs(X-old.X) <= 2 and abs(Y-old.Y) = 2)
                    or (abs(Y-old.Y) <= 2 and abs(X-old.X) = 2)
                    )
            and    Content = 9) Rep,
            CELL                Vrep
      where Vrep.X between Rep.X - 1 and Rep.X + 1
      and   Vrep.Y between Rep.Y - 1 and Rep.Y + 1
     ),
     (select coalesce(group_concat((X-old.X)||','
                                   ||(Y-old.Y) ,';'),
                      '') as CurVic
      from  CELL
      where X between old.X - 1 and old.X + 1
      and   Y between old.Y - 1 and old.Y + 1
     );
```

Script 24.24 - Computing the valid moves to avoid the vicinity of distant repellers

Applied to the world fragment of Figure 24.17, the query computes the set of valid cells:

```
'5,6;5,7;6,6;6,7;7,7'
```

converted into the relative coordinates of the recommended moves:

```
'-1,0;-1,1;0,0;0,1;1,1'
```

24.11.3 Experimentation: the *Attractors-Repellers* simulator

The *Attractors* simulator has been extended to include close and distant repellers. The new control panel is shown in Figure 24.18.

The Figure 24.19 shows the state of a 8x24 world comprising 6 distant repellers and 12 agents. To make the agent/repeller interactions more visible, the agents leave a trace in the cells they have visited, as in the lawmower simulation. The result clearly shows that no agent attempted to cross the distant frontier of any repeller.[25]*

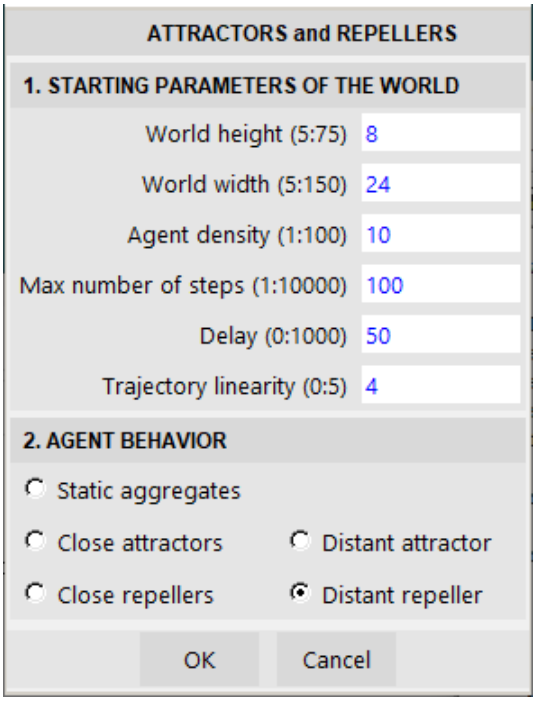


Figure 24.18 - Control panel of the Basic Attractor-Repellers simulator

25. The so-called *final state* shows only 11 agents. This means that two agents are currently in the same cell.

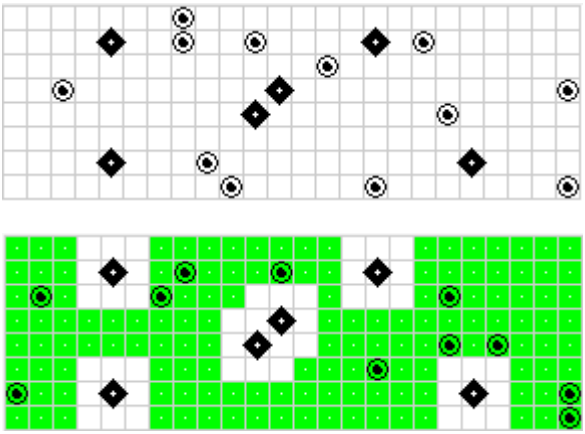


Figure 24.19 - Starting and final states of a 8x24 world comprising 6 distant repellers

The application code

The *Basic Attractors-Repellers* simulator is implemented through the script MAIN-Attractors-Repellers.sql. To make the code more readable, the *world creation* and *world population* procedures are split in two parts, one devoted to attractors and the other to repellers. The main script calls 9 procedures:

- _AGENT-Init-Parameters
- _AGENT-Create-Tables
- _AGENT-Get-Parameters
- _AGENT-Create-World-attractors
- _AGENT-Create-World-repellers
- _AGENT-Populate-World-attractors
- _AGENT-Populate-World-repellers
- _AGENT-Generate-Current-State
- _AGENT-Process-History

24.12 Agent life cycle

The attraction and repulsion reactions developed in the previous experiment, despite their primitive nature, nonetheless bring our agents closer to the behavior of living beings, be they animals or even humans. Let's now try to enrich our agent model by giving them a simple but reasonably realistic life cycle.

24.12.1 Towards autonomous agents

To mimic living beings, we allow agents to be born, to grow old, to generate new agents and to die, generally in this order[26]. Let us briefly analyze these events.

- *An agent is born*. Unlike the previous models, the population is not fully defined at the start of the simulation. Indeed, while the first members are created to start the evolution process[27], new agents will appear thanks to the generation procedure described below.
- *An agent ages*. Time will bring about a change in the agent's capabilities: depending on the time spent in its society, an agent acquires certain capabilities (such as food autonomy or the generation of new agents) and loses others, including life. Age measurement is based on the concept of an *age period* (days, months or years, depending on the species to be modeled), a unit defined by a number of clock ticks[28].
- *Agents generate new agents*. Given the infinite variety of reproduction modes in the animal world, we chose one of the most classical: two agents, of different gender, meet in the same place, and, if certain conditions are met, generate a new agent. This procedure is open to many possible refinements. Can one gender be attracted by the other (close or distant attraction), or do they meet by chance? Can one gender sometimes repel the other's attempts? Does attraction depend on the age of the agents? Can *parents* (so to speak) generate more than one new agent at a time? Does each parent have to respect a latency period between two successive generations? Is there a delay between the meeting and the birth of the new agent (a sort of pregnancy period)? Does each meeting automatically lead to the birth of a new agent? Can certain environmental conditions influence the generation process, such as overpopulation or food abundance?
- Finally *agents die*. After a certain number of age periods, agents are expected to die. Let us submit the description of this event to two questions: (1) does it depend on environmental conditions (food scarcity, disease, attacks from predators), (2) does this age limit represent an exact value, or an average and, in this interpretation, what is the distribution function?

26. In short, *Hatching, matching and dispatching*
27. Starting with the first amoeba would really take up too much time
28. We remember that each clock tick triggers a simulation *step*.

From this general analysis, we draw the subset of characteristics we intend to implement.

- At its birth, an agent has an age of 0. Each additional age period is acquired when the agent has received a definite number of clock ticks.
- To be able to reproduce, an agent must have acquired a definite level of maturity, defined as a minimum age. There is no maximum age limit. In addition, between two successive generations, an agent must wait a latency period defined as a definite number of clock ticks. An agent that satisfies the conditions of maturity and latency is said *fertile*. Otherwise it is qualified *sterile*.
- When a fertile agent finds a fertile partner of a different gender, it asks this partner to generate a new agent. This generation occurs with a certain probability.
- An agent dies when it has just completed its last age period. This limit is common to all the agents.

24.12.2 SQL modeling

A. World structure

To control the evolution of living agents, we need some reference parameters valid for all the simulations.

- ticksPerAge: the number of clock ticks of an age period
- maxAge: the common age of death of the agents
- maturity: the minimum age for an agent to be allowed to generate a new agent
- matchingProba: probability that two partners that meet actually generate a new agent
- latency: minimum period, measured in number of clock ticks, between two generation events from the same agent[29].

These global parameters are implemented as static variables.

B. Agent structure

So far, the state of the agents were limited to their position in the world space (X,Y) and their trajectory (OldX,OldY). Now we must give them new properties that represent their evolution. Hence the four new column of table AGENT:

- Age: its age, the number of completed age periods since its birth

29. This period is designed to model various real or technical phenomena, such as the feeding period of a newborn, pregnancy, preventing partners to continue generating immediately after a generation, therefore avoiding uncontrolled demographic expansion.

- Ticks: the current number of clock ticks since the last age increase
- Gender: its gender (a constant); conventionally denoted by meaningless characters 'F' or 'M'.
- Latency: the current number of clock ticks since the last generation.

Their values must satisfy these constraints:

- $0 \leq \text{Age} < \text{maxAge}$
- $0 \leq \text{Ticks} < \text{ticksPerAge}$

An agent is fertile iif:

- $\text{Age} \geq \text{maturity}$ and $\text{Latency} \geq \text{latency}$

A couple of agents (a1,a2) are allowed to generate a new agent iif:

- a2.X between a1.X - 1 and a1.X + 1
- a2.Y between a1.Y - 1 and a1.Y + 1
- $\text{a1.Age} \geq \text{maturity}$ and $\text{a2.Age} \geq \text{maturity}$
- $\text{a1.Latency} \geq \text{latency}$ and $\text{a2.Latency} \geq \text{latency}$
- $\text{a1.Gender} \neq \text{a2.Gender}$

C. Analysis of the agent behavior

The agents are free to move according to the chosen trajectory style. A cell can host several agents; they do not aggregate and keep their independence.

At each step, before moving, a fertile agent searches its close vicinity for potential fertile partners of different gender. It asks one of these partners, randomly selected, to generate a new agent by sending it the message 'generate'. The partner will generate, in its own cell, one new agent with probability matchingProba.

The new agent is assigned the following properties:

- (X,Y) = position of the generating partner
- (OldX,OldY) = (0,0)
- Ticks = 0
- Age = 0
- Gender = random('M,F')
- Latency = 0

Column Latency of the current agent and of the partner is reset to 0, so that they both become sterile. After the generation event, the current agent moves one position, as usual.

24.12.3 SQL code

The code of Script 24.25 defines the new structure of the AGENT table and sets some realistic values of the global parameters.

```
create table AGENT(AgentID integer primary key,
                   X       integer,
                   Y       integer,
                   OldX    integer default 0,
                   OldY    integer default 0,
                   Message varchar(6) default '',
                   Ticks   integer default 0,
                   Age     integer default 0,
                   Gender  char(1),
                   Latency integer default 0);

set maxAge = 30;
set ticksPerAge = 3
set matchingProba = 100;
set maturity = 6;
set latency = 3;
```

Script 24.25 - Structure of *living* agents

The initial population is created by Script 24.26 (the content of the CELL rows has been initialized according to the desired density). These initial values are chosen to ensure a certain diversity among the agent population.

```
insert into AGENT(X,Y,Ticks,Age,Gender,Latency)
     select X,Y,
            random_i(0,$ticksPerAge$ - 1),
            random_i(int(0.30*$maxAge$),$maxAge$-2),
            iif(random_i(1,100) <= 50,'M','F'),
            random_i(1,$latency$ - 2)
     from CELL where Content = 1;
```

Script 24.26 - Initial state of the agents

Architecture of the main trigger

The T_AGENT_MOVE[30] trigger must be activated for each AGENT row, so, the `when` clause is made up of the sole condition on the nature of the message. The body comprises six tasks, the order of which is important. The first task, *Managing agent age*, consists in updating the time-dependent properties of the agent stored in columns Age, Ticks and Latency. This task can change the status of a agent, which,

30. The new name of the T_AGENT trigger.

from sterile, becomes fertile. In this case, this agent will be allowed to look for a partner (*The agent looks for a partner*) and, in case of success, to ask this partner to generate a new agent (*The agent generates a new agent*). The next tasks move the agent as in the preceding experiments.

The development of each of the first three tasks is examined below.

```
create trigger T_AGENT_MOVE
after update of Message on AGENT
for  each row
when new.Message = 'move'
begin
    <1. Managing agent age>
    <2. The agent looks for a partner>
    <3. The agent generates a new agent>
    <4. The agent computes its next direction>
    <5. The agent computes its next position>
    <6. The agent moves to its new position>
end;
```

Script 24.27 - Architecture of the main trigger

1. Managing agent age

Managing the passing of time of an agent is straightforward (Script 24.28).

```
update AGENT set   Ticks = Ticks + 1,                          [1]
                   Latency = Latency + 1
           where AgentID = old.AgentID;
update AGENT set   Age = Age + 1,                              [2]
                   Ticks = 0
           where AgentID = old.AgentID
           and   Ticks = $ticksPerAge$;
delete from AGENT                                              [3]
           where AgentID = old.AgentID
           and   Age = $maxAge$;
```

Script 24.28 - Managing the age of agents

The values old.AgentID, old.X, old.Y contain the properties of the current agent at the time of the initial event (`update Message`), even when this agent just was deleted.

– The number of ticks received from the clock is incremented (query [1]).

– Once the maximum of ticks is received, the age is incremented and the number of ticks is reset to 0 (query [2]).

– At the death of an agent, the current row is deleted from the table AGENT (query [3])

– The CELL row of a dead agent will be updated later, in task 6.

2. The agent looks for a partner

The query of Script 24.29 stores in variable partner the value of AgentID of a valid partner (randomly selected in PartList) for the current agent [1].

```
set partner = '';                                                    [0]
select random(PartList) into :partner                                [1]
from  (select coalesce(group_concat(AgentID,';')                     [2]
                       filter (where Gender  <> new.Gender           [3]
                                 and   Age >= $maturity$             [4]
                                 and   Latency >= $latency$)         [5]
                       '') as PartList                               [6]
       from   AGENT
       where  X between old.X - 1 and old.X + 1                      [7]
       and    Y between old.Y - 1 and old.Y + 1                      [8]
       )
where  new.Age >= $maturity$                                         [9]
and    new.Latency >= $latency$                                     [10]
and    exists (select * from AGENT
               where  AgentID = old.AgentID);                        [11]
```

Script 24.29 - Searching the vicinity of the current agent for a partner

The quest for a partner is allowed if the current agent has not been deleted [11] and is fertile [9,10].

The subquery extracts a list of valid candidate partners that it returns in PartList [2-8]. It first explores the vicinity of the current agent [7,8]. It concatenates the Id's of those agents [2] that are of the opposite gender [3] and that are fertile [4,5]. If no valid partner has been found, the *null* value is converted into an empty string [2,6].

The existence of a valid partner is indicated by a non-empty value of variable partner.

If the current agent is not allowed to search for a partner, the query fails and the value of variable partner is an empty string [0].

3. The agent generates a new agent

The basic laws of agent modeling implicitly include the principle of *autonomy* according to which an agent is not allowed to change the state of another agent without its consent. Therefore, either the current agent decides to create a new agent,

or it asks its partner to create one of its own. We have decided to adopt the second behavior[31] and to code it as a new trigger.

To ask a definite service to an agent, we send it a message explaining the nature of this service. This is what the current agent does through the query of Script 24.30. The structure of the generation trigger will be described later in this section.

We note that this query fails (that is, it performs no action) if the current agent has been deleted.

```
update AGENT
set    Message = 'generate'
where  AgentID = :partner;
```

Script 24.30 - Asking the partner to generate a new agent

Then, the current agent changes its status from fertile to sterile by resetting its latency period (Script 24.31)[32]. Here too, this query is a *no-op* if the current agent has been deleted.

```
update AGENT
set    Latency = 0
where  AgentID = old.AgentID
and    :partner <> '';
```

Script 24.31 - The current agent become sterile.

4. The agent computes its next direction

This query fails if the current agent has been deleted.

5. The agent computes its next position

This query fails if the current agent has been deleted.

6. The agent moves to its new position

1. The agent quits its current position (also valid for the deleted agent)
2. Then moves to its next position (if the agent still exists)

31. We could chose a more realistic idea, to let the agent of a certain gender generate the new agent. This is left as an exercise.
32. This behavior is not completely correct is the value of the matchingProba variable is not 100. If the partner doesn't satisfy the demand of the current agent, there is no need to *sterilize* it. This also is left as an exercise.

Architecture of the generation trigger

Now, let us concentrate on the new trigger that captures the generation event (Script 24.32). Its when clause controls the message received. It also expresses that the demand of generation can be accepted or rejected according to the probability specified in variable matchingProba. Each of the three tasks of the body can be expressed as a simple query (Script 24.33).

```
create trigger T_AGENT_GENERATE
after update of Message on AGENT
for  each row
when new.Message = 'generate'
     and random_i(1,100) <= $matchingProba$
begin
    <1. The agent creates a new agent>
    <2. The agent adds a unit in its own cell>
    <3. The agent becomes sterile>
end;
```

Script 24.32 - Architecture of the generation trigger

1

```
insert into AGENT(X,Y,Gender) values(old.X,old.Y,random('M;F'));
update CELL set Content = Content + 1
where (X,Y) = (old.X,old.Y);
update AGENT set Latency = 0 where AgentID = old.AgentID;
```

Script 24.33 - Creating a new agent and updating the parent agent

24.12.4 Experimentation: the *LifeCycle* simulator

The first section of the control panel of the *Basic Agent Life cycle* simulator is similar to that of the previous experiments. It also includes a second section specifying the life cycle parameters of the agents (Figure 24.20).

The *Extended* simulator comprises the same additional sections as the preceding simulators. It also includes a JOURNAL table that records at each step, the number of agents, the (cumulative) number of births and the (cumulative) number of deaths.

The graphical representation of the agents in their space needs some additional properties to better show the status of each agent. The new code is presented in Figure 24.21.

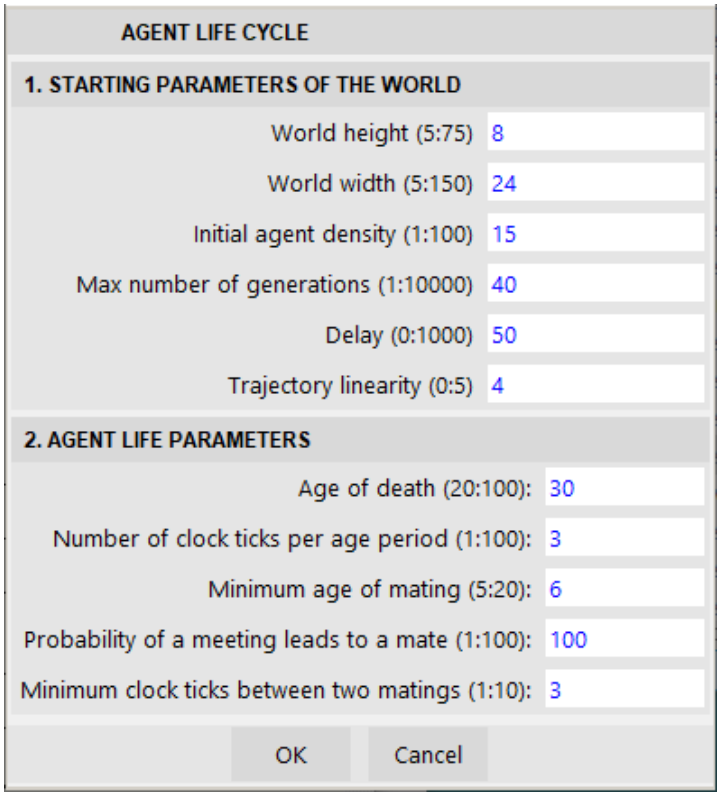


Figure 24.20 - Control panel of the Basic Agent Life Cycle simulator

A simulation has been run with the sample parameters shown in the control panel screen shot. The initial population comprised 25 (sterile) agents including 14 of gender 'F'. The content of the AGENT table (Figure 24.22) describes the state of the 52 agents at the completion of the 24th step. This state results from 35 births and 8 deaths among the initial population.

The graphical representation makes more explicit the spatial configuration of the agent population (Figure 24.23). It shows that,

- 29 cells contain one sterile agent (black circle)
- 6 cells contain one fertile agent of gender F (red circle)
- 6 cells contain one fertile agent of gender M (blue circle)
- 2 cells contain several sterile agents (green diamond)
- 1 cell contains fertile agent(s) of gender F but none of gender M (red diamond)
- 1 cell contains fertile agent(s) of gender M but none of gender F (blue diamond)
- 1 cell contains both fertile agent(s) of gender F and fertile agent(s) of gender M (magenta diamond)

Figure 24.21 - Color code of the content of cells

AgentID	Y	X	Ticks	Age	Gender	Latency
1	6	22	0	25	M	0
2	6	3	0	28	F	2
4	4	3	2	20	M	2
5	7	6	2	21	F	8
6	1	7	0	27	F	14
8	7	10	0	26	M	2
10	8	15	1	26	F	0
12	4	3	0	25	F	10
13	5	17	0	25	F	6
14	2	16	2	29	M	4
17	5	8	2	24	F	18
18	4	7	1	25	F	2
20	1	1	2	18	M	9
21	3	8	0	29	M	1
23	7	19	0	20	F	2
24	3	11	2	29	F	25
25	8	20	0	20	F	0
26	7	15	0	7	M	2
27	2	10	0	7	F	1
28	6	15	0	7	M	1
29	4	9	0	7	F	2
30	1	24	0	7	M	21
31	2	7	0	7	F	21
32	5	11	2	6	M	20
33	7	2	1	6	M	19
34	2	16	0	6	F	18
35	3	7	0	6	M	18
36	6	22	0	6	M	18
37	7	23	0	6	M	18
38	4	22	2	5	F	17
39	5	13	0	5	F	15
40	2	5	0	5	M	15
41	1	20	2	4	F	14
42	4	21	2	4	M	14
43	7	24	1	4	F	13

44	1	13	0	4	M	12
45	7	13	1	3	M	10
46	1	18	0	3	F	9
47	4	4	0	3	F	9
48	8	10	2	2	M	8
49	7	8	2	2	F	8
50	2	19	0	2	F	6
51	6	17	0	2	F	6
52	2	8	2	1	M	5
53	3	17	1	1	M	4
54	7	4	2	0	M	2
55	4	7	2	0	M	2
56	6	19	2	0	F	2
57	8	8	2	0	M	2
58	4	9	1	0	M	1
59	7	16	0	0	F	0
60	6	22	0	0	M	0

Figure 24.22 - State of the AGENT table after step 24

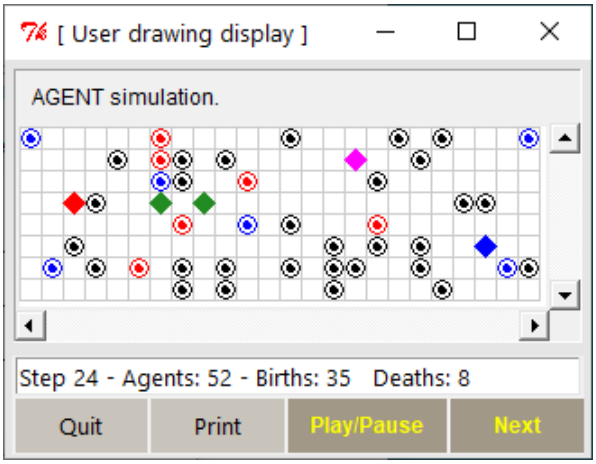


Figure 24.23 - State of the space after step 24

The application code

The *Basic Life cycle* simulator is implemented through the MAIN-Lifecycle.sql script The main script calls the same 7 procedures as the preceding simulators:

- _AGENT-Init-Parameters
- _AGENT-Create-Tables
- _AGENT-Get-Parameters
- _AGENT-Create-World
- _AGENT-Populate-World
- _AGENT-Generate-Current-State
- _AGENT-Process-History

24.13 Agent life cycle with birth control

Running the *Life cycle* model built in the previous section shows that maintaining a stable population can be difficult, as the evolution is highly sensitive to values of the parameters. Let us analyze its behavior and try to better control it.

24.13.1 The LifeCycle model is unstable

Most of the time, the population grows continuously until it completely fills the world, or, on the contrary, decreases until extinction, leaving an empty world. This what the curves of Figure 24.24 show.

Let Ncell be the number of cells and Nag the number of agents. Ncell is a constant of the simulation while Nag is time-dependent (i.e., step-dependent). Let also density denote the ratio 100*Nag/Ncell. Since a cell can host several agents, the density has no upper limit and can get far greater than 100.

Each curve plots the evolution of density during a 200-step simulation for a definite value of matchingProba, from 100 (each meeting generates a new agent) to 20 (only 20% of the meetings generate an agent). The five simulations have been run with these parameter settings:

```
height,width = 8,16       ticksPerAge   = 3
density      = 10         maxAge        = 35
maxStep      = 200        maturity      = 10
trajectory   = 4          latency       = 6
                          matchingProba = 100,80,60,40,20
```

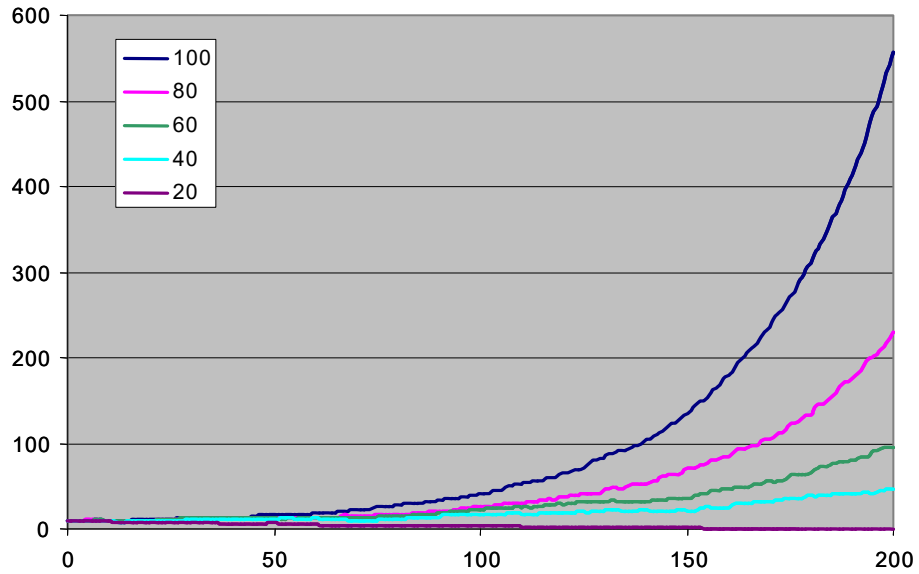


Figure 24.24 - Unstable growth of the agent population

For the largest values of matchingProba (from 100 down to 40) the curves exhibit an exponential growth, while the smallest values (e.g., 20) lead to the extinction of the agent population. Choosing a value that gives a stable evolution on the long run is quite difficult, if not impossible. All the more so since experimenting with the other behavior parameters, namely ticksPerAge, maxAge, maturity and latency lead to the same conclusion: ***this model is intrinsically unstable***.

Given this property, the simulator stops when the number of agents drops to 0 or exceeds twice the number of cells.

24.13.2 Adding birth control to the LifeCycle model

Modeling real systems of this kind requires additional regulation rules that limit the increase and the decrease of the population. Introducing in the model a resource that is naturally limited, such as food, can be a good example. If the world can supply a quantity of Q_f food units per step and if each agent needs Q_a units per step, the world cannot feed more than Q_f/Q_a agents in each step[33].

Such a rule can result in the dynamic adaptation of some of the global parameters of the model. For example, the matchingProba probability, which was a constant in the previous experiments, can be computed in each step: reduced when the population size becomes too large, and increased when the population shrinks to a dangerous level. Similarly, the global parameters latency, maturity and maxAge can also be dynamically adjusted to regulate the population. By fine tuning these parameters, we can develop a better model, closer to the way the target system behaves. For instance, in a close world:

too many agents ⟶ less food ⟶ higher death toll ⟶ fewer agents
too few agents ⟶ more food ⟶ enthusiasm to reproduce ⟶ more new agents

These arguments are valid for models with only one type of agent. The introduction of another competing type will change the game, particularly in the case of a *predator-prey* scheme:

too many predators ⟶ preys decrease ⟶ predators decrease ⟶ fewer predator
too few predators ⟶ preys increase ⟶ predators reproduce ⟶ predators increase

Working on the expression of the matchingProba parameter, as suggested above, seems a good idea.

Let us adopt a new constant parameter, maxDensity, that specifies the maximum density that the agent population can never exceed. At any step of the simulation, the generation process, implemented by the T_AGENT_GENERATE trigger, must ensure that this condition always be satisfied:

$0 < 100*Nag/Ncell \leq maxDensity$

33. Simplified hypothesis of the reality: an agent can make do with just under Q_a food units for a limited time before dying.

Let us try to convert the matchingProba probability into a dynamic parameter. Instead of assigning it a constant when the world is built (see Script 24.25),

```
set matchingProba = 100;
```

we *compute* its value at the beginning of each step:

```
select <f(Nag)> into matchingProba from AGENT;
```

Let us consider the following formula

$$matchingProba = max(100 \times (1 - 100/maxDensity \times Nag/Ncell), 0)$$

Let us consider that the space is made up of Ncell = 8*16 = 128 cells and the maximum density is maxDensity = 60%. The population size cannot exceed Nag = 0.6 x 128 = 76.8 ≈ 77 agents. As the number of agents tends to that limit, the probability of two fertile agents generating a new agent tends to zero. On the contrary, when this number decreases, this probability augments.

The table of Figure 24.25 gives the values of matchingProba for increasing values of Nag. The SQL expression of matchingProba is shown in Script 24.34.

Nag	matchingProba
2	97
10	87
20	74
30	61
40	48
50	35
60	22
70	9
76	1
77	0

Figure 24.25 - Values of matchingProba as a function of Nag

```
select max(100*(1 - (100/$maxDensity$) * (count(*)/$Ncell$)),0)
       into matchingProba
from   AGENT;
```

Script 24.34 - SQL expression of matchingProba

The T_AGENT_GENERATE trigger described in Script 24.32 is slightly modified, matchingProba being now a dynamic variable. Its new version is shown in Script 24.34.

```
create trigger T_AGENT_GENERATE
after update of Message on AGENT
for  each row
when new.Message = 'generate'
     and random_i(1,100) <= :matchingProba
begin
    <1. The agent creates a new agent>
    <2. The agent adds a unit in its own cell>
    <3. The agent becomes sterile>
end;
```

Script 24.35 - New version of the T_AGENT_GENERATE trigger

24.13.3 Experimentation: the Agent *LifeCycle with birth control* simulator

The control panel of the *LifeCycle* simulator of the previous experiment has been modified to add the absolute maximum density (Figure 24.26).

The figure 24.27 shows the result of five simulations for different values of maxDensity. They have been run with these parameter settings:

```
height,width = 8,16      ticksPerAge = 3
density      = 10        maxAge      = 35
maxStep      = 400       maturity    = 10
trajectory   = 4         latency     = 6
                         maxDensity  = 100,80,60,40,20
```

Each curve plots the evolution of density during a 400-step simulation for a definite value of maxDensity, from 100 to 20.

This model protects the agents from overpopulation. However, it does not prevent them from extinction for certain combinations of evolution parameter values. As in the real world, extinction threatens very small populations, which are particularly vulnerable to uneven age and gender distribution. For example, the disappearance of one gender automatically leads to the extinction of the entire population.

The application code

The *Basic LifeCycle with birth control* simulator is implemented through the script MAIN-Lifecycle-Control.sql. It calls the same 7 procedures as the *LifeCycle* simulators:

- _AGENT-Init-Parameters
- _AGENT-Create-Tables
- _AGENT-Get-Parameters
- _AGENT-Create-World

– _AGENT-Populate-World
– _AGENT-Generate-Current-State

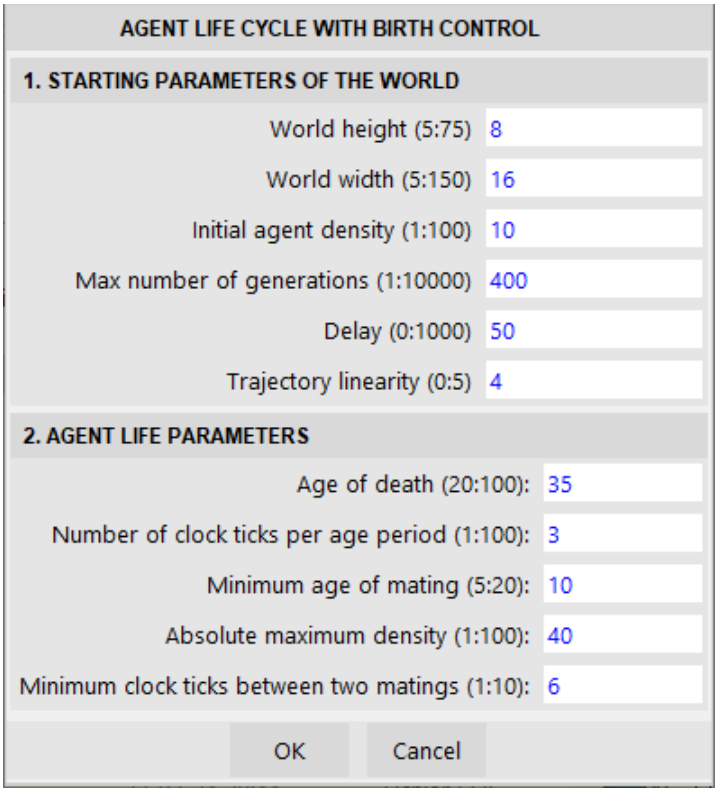


Figure 24.26 - Control panel of the Basic *Agent Life Cycle with birth control* simulator

24.13.4 Refining the birth control model

The formula for calculating the matchingProba parameter is an abstract rule that stands out in a model whose rules, so far, describes as closely as possible the concrete behavior of a population of living entities in the real world. The result is a hybrid model in which certain components, because of their abstract nature, are no longer representative of the real system.

Wouldn't it be possible to replace this equation with an explicit representation of the concrete objects and behaviors that govern the evolution of populations?

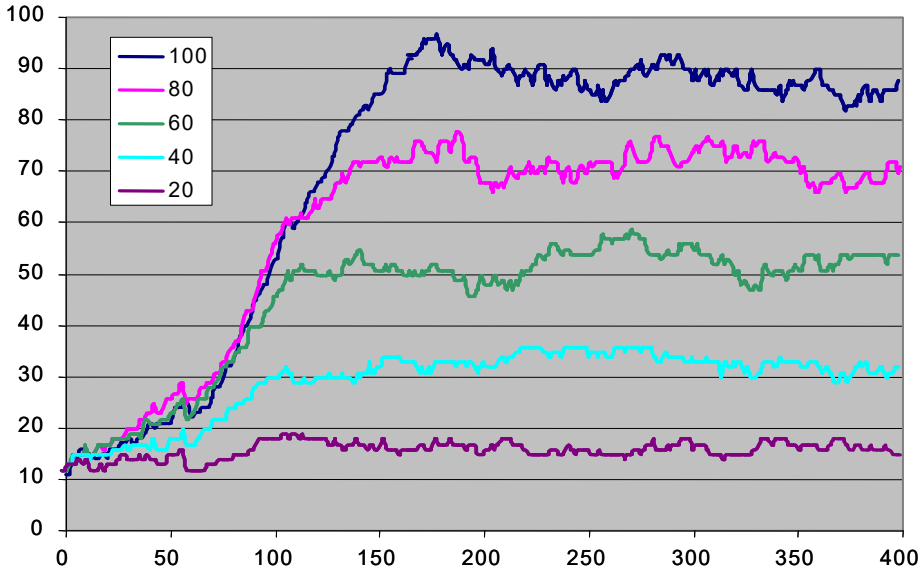


Figure 24.27 - LifeCycle simulations with birth control

Let's consider one of the most obvious sources of this phenomenon: the *limited availability of food*, as suggested in the introduction. An agent must feed to remain active. It finds food in every cell it visits. Once the food in a cell has been consumed, it is replenished by a certain amount with each step. An agent's ability to reproduce depends on its state of satiety: a better-fed agent tends to reproduce more *dynamically*. A malnourished agent, on the other hand, will not reproduce and may even die of starvation.

Here are a few ideas for refining the agent model (Figure 24.28). Their implementation is left to the reader's initiative.

Facts about the system	**Modeling suggestions**
each cell contains a variable quantity of food,	add column **Food** to table **CELL.**
this quantity increases with each unit of time (i.e., each step); it decreases when an agent consumes it,	static parameters **foodInc** (increase per clock tick) **foodDec** (consumption by agent per clock tick).
the maximum amount of food in a cell,	static parameter **maxFood**.
every agent has a health capital,	add column **Health** to table **AGENT.**

this health level is limited to a maximum. It increases with the amount of food consumed. It decreases with each unit of time,	static parameters **maxHealth**, **foodHealthRatio** (level increase per unit of food consumed), **healthPerStep** (level loss per step).
when an agent's health level decreases, its ability to reproduce decreases,	dynamic parameter **matchingProba** is no longer global but depends on the health level of each agent; it is also possible to act on the **maturity** and **latency** parameters; the value of these parameters is calculated at each step by each agent.
as well as its life expectancy,	*ditto* for the **maxAge** parameter (premature death) or perhaps **ticksPerAge** (accelerated aging).
an agent dies as soon as its health level drops to zero.	

Figure 24.28 - A better description of the evolution of the population

24.13.5 Toward learning agents

The behavior of agents, no matter how sophisticated, obeys rules set in advance and applied in the same way by each agent. In other words, agents' knowledge of their environment is constant from birth to death. This knowledge is therefore non-evolving, which raises the question of *learning*: can an agent, through experience or exchanges with other agents, enrich the knowledge it has of its environment and thus behave more efficiently or more safely?

To illustrate this point, let's imagine that the food found in the cells are plants of two varieties, which the agent initially ignored. Some are edible, others toxic. When an agent eats a toxic plant, it gets sick, resulting in an abnormal decrease in its health level. While the standard decrease is one unit per step, ingesting a toxic plant results in a loss of, say, three units, which weakens the agent and can even lead to death. The combination of these two events - the ingestion of a toxic plant and the drop in health - can provide the agent with new knowledge that will help it survive.

We can imagine several ways to implement the reaction of the agent to this danger. One, let’s say more *agent-oriented*, would be to interpret a health level loss of three units as an event that activates a specific trigger. The latter would analyze the environment of the agent, infers the rule *toxic food in the current cell* $\rightarrow$ *danger* and records it in the state of the agent.

What's more, when two agents meet (they visit the same cell, for whatever reason), they can share their knowledge of which plants to avoid, thus contributing, through a peer-to-peer process, to the development of a collective knowledge.

24.14 Logging elementary operations and debugging

Being more technical, this section is intended for experienced experimenter.

So far, we've designed and developed the code for the *Basic* simulators. The existence of another variant, called *Extended*, has only been mentioned. We will now give more details on this version.

The code of the controller and of the triggers of the Basic version has been enriched with instructions that inform the user/developer about the intermediate states of the world (agent population and space) and of the elementary operations of the agents at each step of the simulation. This additional code is activated or disabled in the INFORMATION part of the control panel. It is intended,

- to better understand the way the agents behave,
- to track and debug the logic errors, in particular in the body of the triggers.

This practice is of general use in the development of complex programs, where it is called *code instrumentation*.[34]

Let us examine how it works by running the *Attractors-Repellers* simulator to create a small world populated with 5 agents and 4 distant repellers. Its starting state is shown in Figure 24.29

Through the settings of the INFORMATION part of the control panel (Figure 24.30), we ask the simulator to print in the output window,

- the initial and final states of the cells,
- at each step, the resulting state of the cells (Show grids),
- at each step, for each agent, the sequence of the elementary actions carried out (Show moves).

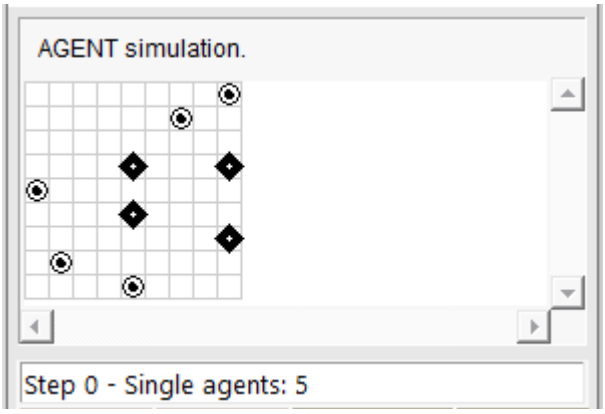


Figure 24.29 - Starting state of the simulation

34. https://en.wikipedia.org/wiki/Instrumentation_(computer_programming)

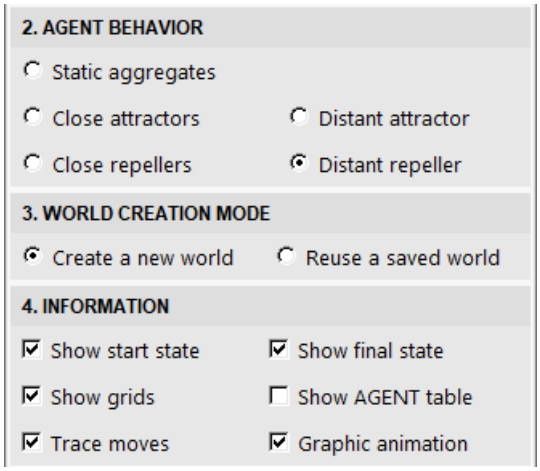


Figure 24.30 - Information settings

From the (fairly long) trace of the simulation, we examine what happened during the first step. The trace, an excerpt of which is shown in Figure 24.31, comprises the initial state of the space[35], the detail of the actions performed by each of the five agents during the execution of step 1 then the new state of the cells at the end of this first step. Actually, we focus on the behavior of agent #2 (fragment in blue color), initially located in cell (Y,X) = (2,7). Its trace, entitled Processing Agent #2 in (2,7) [0,0][36], comprises four paragraphs:

1. *Identifying distant repellers*

   In this action, the agent identifies its two distant repellers and collects their position in relative distances: `[2,2;-2,2]`[37].

   Then, it computes the union of their vicinity: [2,2;-2,2;-3,1;-2,1;-1,1;1,1;2,1;-3,2;-1,2;1,2;-3,3;-2,3;-1,3;1,3;2,3],

2. *Selecting the next move*

   The agent computes its own vicinity: [-1,1;1,1;-1,-1;0,-1;1,-1;-1,0;0,0;1,0;0,1] ,

   By computing the set difference of these cell coordinates, it derives the *clean cells* of its vicinity, that is, those that do not belong to the vicinity of the repellers [-1,-1;1,-1;0,1;0,0;1,0;-1,0;0,-1].

   Finally, it selects, at random, one of these cells: **[0,1]**.

35. Remember that repellers are coded by Content = 9.
36. This title indicates the current (Y,X) position and the values of (OldX,OldY) of the agent.
37. (X offset,Y offset)

3. *Selecting deltas*

   Since avoiding the repellers is of highest priority, the trajectory style is ignored. The next move is the direction selected above.

4. *Final decision*

   The destination cell is computed: (Y,X) = (3,7).

The resulting space of the first step closes this trace.

```
   1 2 3 4 5 6 7 8 9
 1 . . . . . . . . 1
 2 . . . . . . 1 . .
 3 . . . . . . . . .
 4 . . . . 9 . . . 9
 5 1 . . . . . . . .
 6 . . . . 9 . . . .
 7 . . . . . . . . 9
 8 . 1 . . . . . . .
 9 . . . . 1 . . . .

-- Execute generation 1 --

Processing Agent #1 in (1,9) [0,0]
...
- Final decision:
  Agent #1 moves from (1,9) to (1,9)

Processing Agent #2 in (2,7) [0,0]
- Identifying distant repellers:
  Distant repellers:           [2,2;-2,2]
  Vicinity of repellers:       [2,2;-2,2;-3,1;-2,1;-1,1;1,1;2,1;
                                -3,2;-1,2;1,2;-3,3;-2,3;
                                -1,3;1,3;2,3]
- Selecting the next move:
  Vicinity of the agent:       [-1,1;1,1;-1,-1;0,-1;1,-1;
                                -1,0;0,0;1,0;0,1]
  Clean vicinity of the agent: [-1,-1;1,-1;0,1;0,0;1,0;-1,0;0,-1]
  Selected direction:          [0,1]
- Selecting deltas:
  Selected deltas of (2,7): 0,1
- Final decision:
  Agent #2 moves from (2,7) to (3,7)

Processing Agent #3 in (5,1) [0,0]
...
- Final decision:
  Agent #3 moves from (5,1) to (4,2)

Processing Agent #4 in (8,2) [0,0]
...
- Final decision:
  Agent #4 moves from (8,2) to (8,3)

Processing Agent #5 in (9,5) [0,0]
- Final decision:
  Agent #5 moves from (9,5) to (8,4)

...
```

```
  1 2 3 4 5 6 7 8 9
1 . . . . . . . . 1
2 . . . . . . . . .
3 . . . . . . 1 . .
4 . 1 . . 9 . . . 9
5 . . . . . . . . .
6 . . . . 9 . . . .
7 . . . . . . . . 9
8 . . 1 1 . . . . .
9 . . . . . . . . .
```

Figure 24.31 - Excerpts of the trace of a simulation (distant repellers)

24.15 SQL coding in SQLfast

This material has been borrowed from the section ***Coping with the limitations of the trigger language*** *of Case study* ***Active Databases****.*

The body of the triggers developed in this study is written in some kind of *pseudocode* that makes the algorithm structure readable but is not executable as such. The reader of this study may be surprised when comparing this code with that of the SQLfast applications. There are two reasons. First, the syntax of the triggers in SQLite, which is also that in SQLfast, strictly complies with the standard that defines the body of a trigger as a *sequence of SQL statements*. So, usual features of programming languages such as *variables*, *if-then-else* control statements and *procedure calls* are not allowed[38]. This point is discussed below. The second reason is functional. The actual code of the triggers comprises ancillary functions without interest at the problem modeling phase. Such is the case of the *tracing functions* Show grids and Trace moves (enabled in the INFORMATION part of the control panels) executed by the triggers.

We show now that we can easily compensate for limited syntax of the trigger body.

Simulating variables

Usually, a value used in several statements of the trigger is stored in a local variable. This is particularly recommended if the acquisition cost of this value is high, for instance, if it is computed by an aggregation query.

If the concept of local variable is missing, a technical table containing a single row, where each column implements a variable can be used instead, as illustrated by table **Var1** in Script 24.36.

38. Most major DBMSs provide a richer programming language to write stored procedures and triggers.

```
create table Var1(DeltaX integer, DeltaY integer);
create trigger ...
begin
   ...
   insert into Var1(DeltaX,DeltaY)
      select ...
      from   AGENT
      where  AgentID = new.AgentID;

   update AGENT
   set X = X + (select DeltaX from Var1),
       Y = Y + (select DeltaY from Var1)
   where AgentID = new.AgentID;;
   ...
   delete from Var;
end;
```

Script 24.36 - Single-row table **Var1** acts as a series of local variables

This example also shows how the query form `select ... into DeltaX,DeltaY` is simulated with the form `insert into Var1(DeltaX,DeltaY) select ...` .

Simulating *if-then-else* statements

If the trigger language of the DBMS does not include *if-then-else* alternative control statement, then Script 24.37, in which <C>, <C1> and <C2> denote arbitrary SQL conditions, is invalid.

```
create trigger TRG_DEL_T
before delete on T
begin
   if (<C>)
      then delete from S where <C1>;
      else update S set B = null where <C2>;
   endif;
end;
```

Script 24.37 - Standard if-then-else alternative

We suggest two methods to transform this trigger into pure sequential code. The first one consists in adding respectively <C> and not <C> to each branch of the alternative (Script 24.38). If the evaluation of <C> is expensive, its result can be stored in a local variable, that will be used in next statements as a substitute for <C>.

```
create trigger TRG_DEL_T
before delete on T
begin
   delete from S where <C1> and <C>;
   update S set B = null where <C2> and not <C>;
end;
```

Script 24.38 - Transformation of if-then-else alternative - Method 1

According to the second method, each branch is translated into a *distinct trigger* the `when` clause of which decides whether it will fire (Script 24.39).

```
create trigger TRG_DEL_T1
before delete on T
when <C>
begin
   delete from S where <C1>;
end;

create trigger TRG_DEL_T2
before delete on T
when not <C>
begin
   update S set B = null where <C2>;
end;
```

Script 24.39 - Transformation of if-then-else alternative - Method 2

It is important to note that the validity of these transformations depends on whether the result of the execution of the first branch affects the evaluation of the second instance of condition <C>. For example, in Script 24.38, if <C> is true, the first statement is executed. However, this execution may make condition <C> false, in which case both branches of the alternative will be executed.

Calling an external procedure

Most DBMS allow the body of a trigger to ask the execution of a procedure written in some proprietary or general programming language. Such a statement will generally look like this one (Postgresql):

```
execute procedure printMessage('Agent #'||old.AgentID);
```

or (Oracle):

```
printMessage('Agent #'||old.AgentID);
```

The procedure `'printMessage'` is intended to write a message on the user console. Unfortunately, the body of an SQLite trigger must be made up of a sequence of SQL data modification queries only (`insert`, `update`, `delete`) plus a simple form of `select` query initially intended to (conditionally) raise an exception. So, no explicit procedure nor function call.

We will consider the following syntax of this variant of `select` query:

```
select <function> [where <condition>];
```

Where `<function>` denotes a native or UDF function that (optionally) returns a value (which is ignored) after executing some desired action (the interesting part). So, instead of

```
printMessage('Agent #'||old.AgentID);
```

we will write:

```
select printMessage('Agent #'||old.AgentID);
```

In the same way, instead of:

```
if (<condition>) printMessage('Agent #'||old.AgentID);
```

we will write:

```
select printMessage('Agent #'||old.AgentID) where <condition>;
```

If we find this syntax a bit awkward, we can improve it as follows:

```
set message = select printMessage;
```

Then, wherever in the body of a trigger we want to write a message in the output window, we simply write:

```
$message$('Agent #'||old.AgentID') where $showMove$;
```