CIRAD

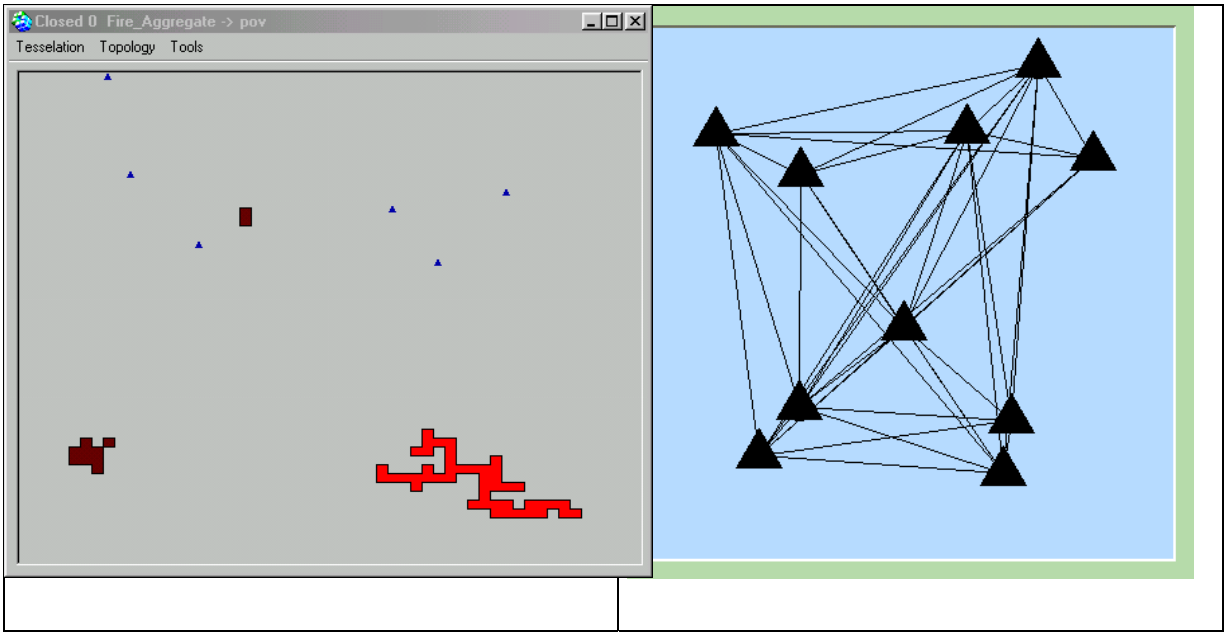
Land and Resources Programme

CORMAS

COMMON-POOL RESOURCES AND MULTI-AGENTS SYSTEMS

Tutorial 1

Manipulation of an existing example and creation of a new model



January 2003

1. INTRODUCTION

1.1.Purpose

The purpose of this tutorial is to help you to build your own multi-agents model, using Cormas. It shows how to use an existing model in Cormas (Conway model) and guide you to build a simple model called "Fire".

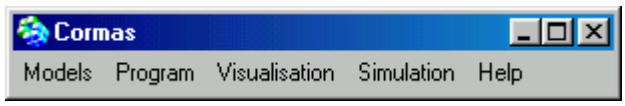
But first of all, it starts with a short presentation of the Cormas interface.

1.2.Prerequisites

The reader of this tutorial is supposed to be familiar with the Smalltalk language and the VisualWork environment, in particular the browser and the inspector.

2.THE CORMAS INTERFACE

In order to start the Cormas environment you have to select Cormas from the Tools menu of the VisualWorks launcher (main window). You have the choice between the French or English version.



The CORMAS window is divided into five parts:

1. The management of your models ('Models' menu) where you can import models, create a new model, export, close the existing model.
2. The definition of the model ('Program' menu) where you can describe the entities of your simulation ('the class for each entity' sub-menu), the methods to activate the entities and hence control the simulation ('the simulation organisation' sub-menu) and the points of view on your simulation ('the observer' sub-menu);
3. The various kind of visualisation, essentially the grids, direct communication graph and the chart in the 'Visualisation' menu;
4. The simulation control itself ('Simulation' pane).
5. The 'Help' menu.

At this stage, you are ready to work with Cormas and will learn first to manipulate an existing model and, second, to build your own model from scratch on a simple example.

3. MANIPULATING AN EXISTING EXAMPLE

To enter into Cormas, we will first import an existing model called Conway.

3.1 Getting the Conway model

To verify if the Conway model is already present into your computer, just see if a directory called "Conway" exist in vw7/cormas/Models/.

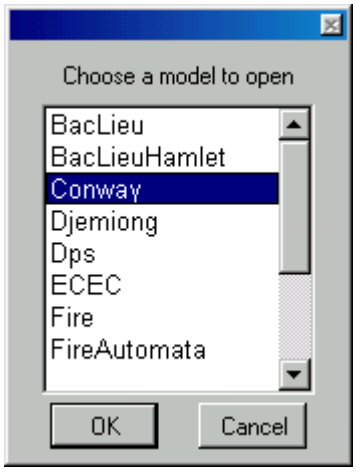
If not, you must download it from the Cormas web site:

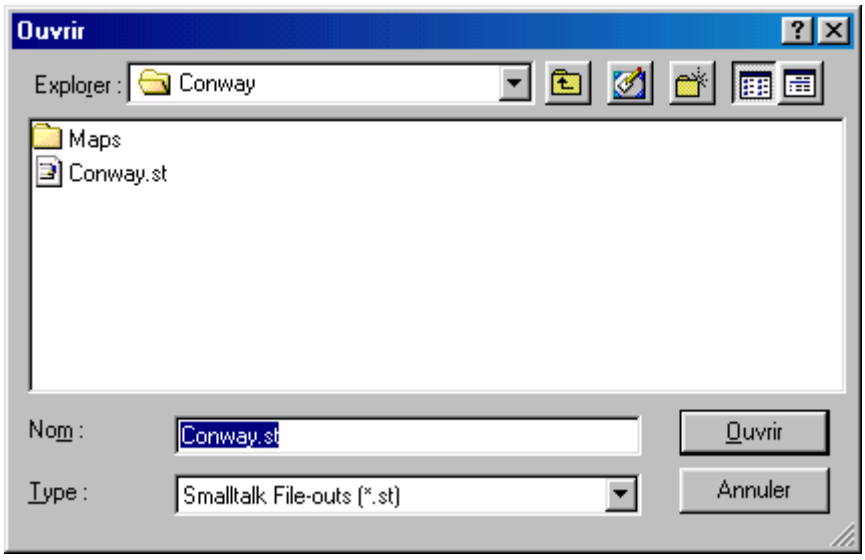
http://cormas.cirad.fr/logiciel/Conway.zip

You must unzip this file into the directory vw7/cormas/Models. A new directory with the same name as the model is created. It contains the "Conway.st" file and other subdirectories

3.2. Loading the Conway model

In order to load the model on Cormas, you select 'import' from the 'Models' menu of the Cormas window. A window opens with a list of models. Select the model Conway by double-clicking on it or by clicking on it and clicking on the OK button.

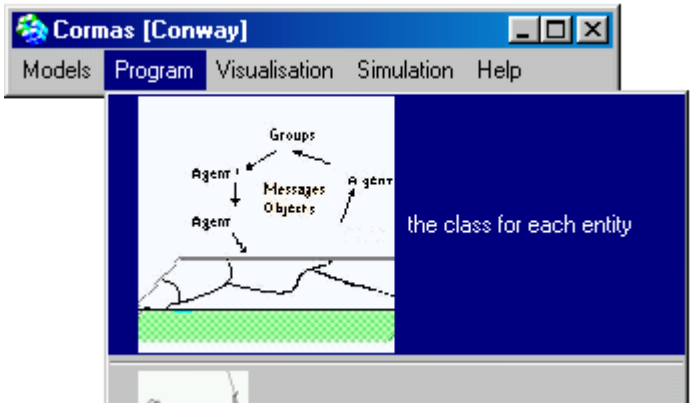


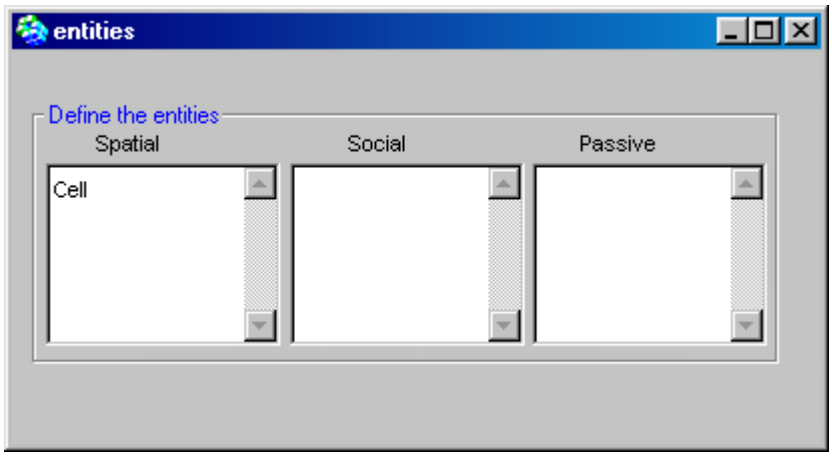


The Cormas models are under the directory Models.

Remark: If the Conway model doesn't appear in this windows, that means that you need to download it from the Cormas web site : http://cormas.cirad.fr/logiciel/Conway.zip

Then, the 'Cellule_Conway' appears in the spatial entity list.





3.3 Inspect the code

3.3.1 The transition function

Double-clicking on it opens a browser displaying the hierarchy of classes leading to the Cell class and the various methods protocols. You will see that the Cell is a direct subclass of SpatialEntityCell which describes the behaviour and structure of a generic cell of a cellular automata (CA). A cell just have to describe the current state (attribute 'state') and the next state for the next step updating (attribute 'bufferState'). These attributes are already defined in the superclass with its accessors. For each cell, you have to describe the *transition functions* :

'newState' to compute the next state given the current state;

'updateState' to update the current state with the next state.

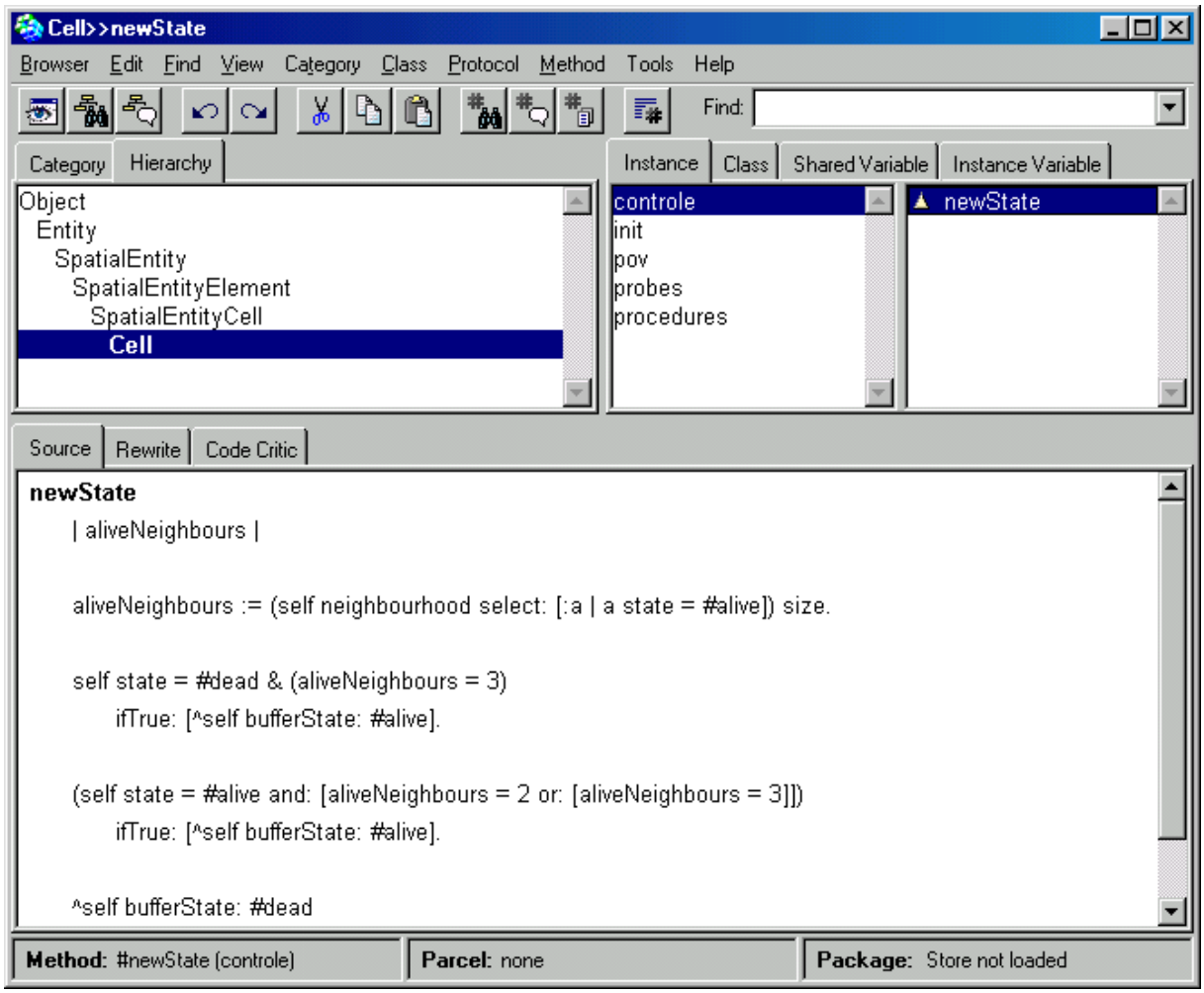
In the Cell the state is either '#alive' or '#dead'. Only 'newState' is redefined because 'updateState' does the default behaviour. By clicking on the method (in the 'controle' protocol), you will discover how the computation of the next state is performed:

It computes the number of alive neighbours (neighbourhood is defined in the 'Entity' class);

If the number is 3 and the cell is dead, the cell becomes alive;

If the number is 3 or 2 and the cell is alive, the cell stays alive;

Otherwise the next state becomes dead.



In order to run a synchronous simulation, you can see that in this method the state of the cell is not changed. This action will occur in the "updateState" method, inherited from SpatialEntityCell class :

```
updateState
    self state ~= self bufferState ifTrue: [self state: self bufferState]
```

Now, you can close the browser.

3.3.2. Code of the Controller

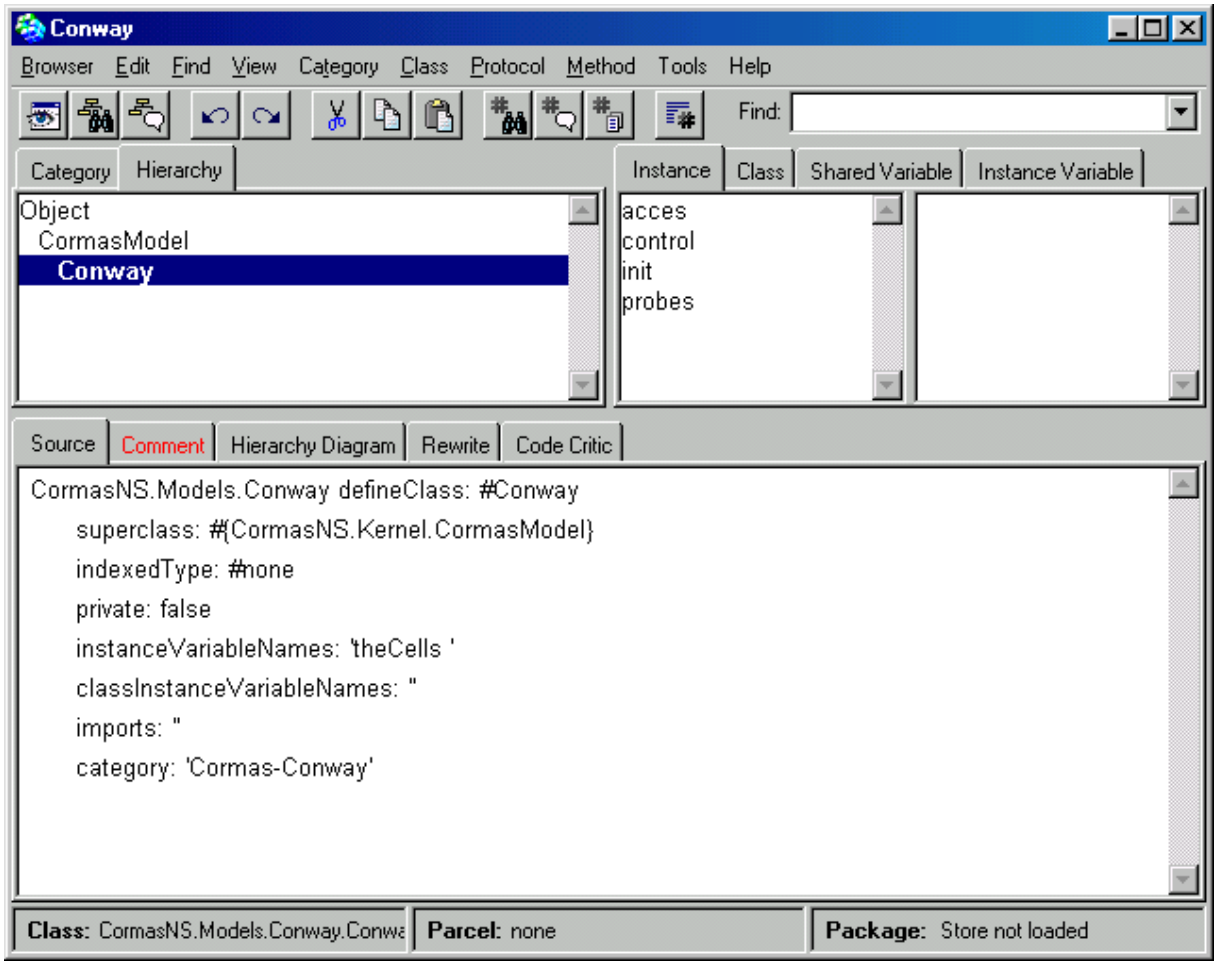
By clicking on the 'Prepare and Schedule' button of the Cormas window, you open a browser on the Conway model. This class is a subclass of CormasModel. It is the main class of your model and, sometimes, it is called the Controller or the Scheduler.

Here, you can see the attribute "theCells" which is a collection of pointers to each cell. The other attributes (alive, dead and so on) are used to build the charts.

Any model must have at least two protocols:

The 'init' protocol for initialising the automata;

The 'control' protocol for running the model.



The 'init' method of the model sends a message to every cell which can be the 'init' message by default but any method you specify by calling:

```
self initCells: #<the init message name>
```

This method must be defined in the cells (in this case in Cell). For example, let's see the initRandomly method of the controller (in the 'init' protocol) :

```
initRandomly
    self initCells: #initRandomly.
    self initData
```

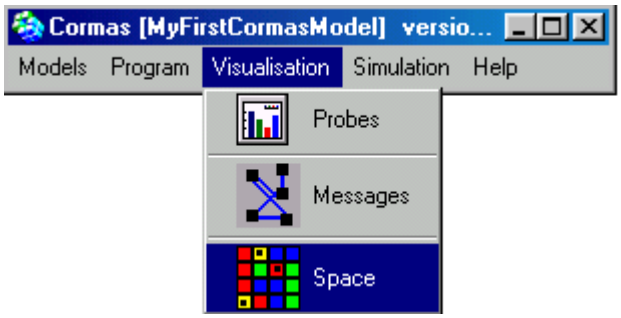
This method will call the initRandomly method of each cell :

```
initRandomly
    super init.
    Cormas random < 0.5
          ifTrue: [self state: #dead]
          ifFalse: [self state: #alive]
```

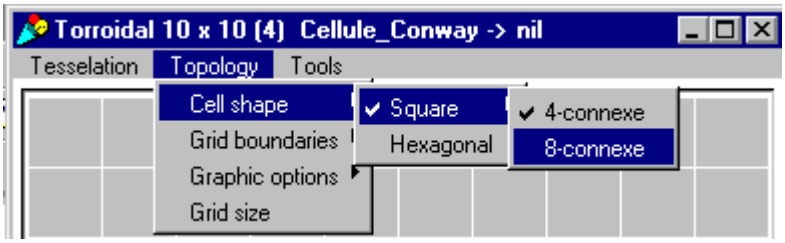
To run the model, the 'CormasModel' class (the super class) defines the method 'stepSynchronously:' which calls 'newState' on every cells and then 'updateState' for each cell.

Another method is called 'stepAsynchronously:' to call 'newState' and 'updateState' on every cells consequently having each cell changing state immediately.

3.4 Opening of the spatial grid

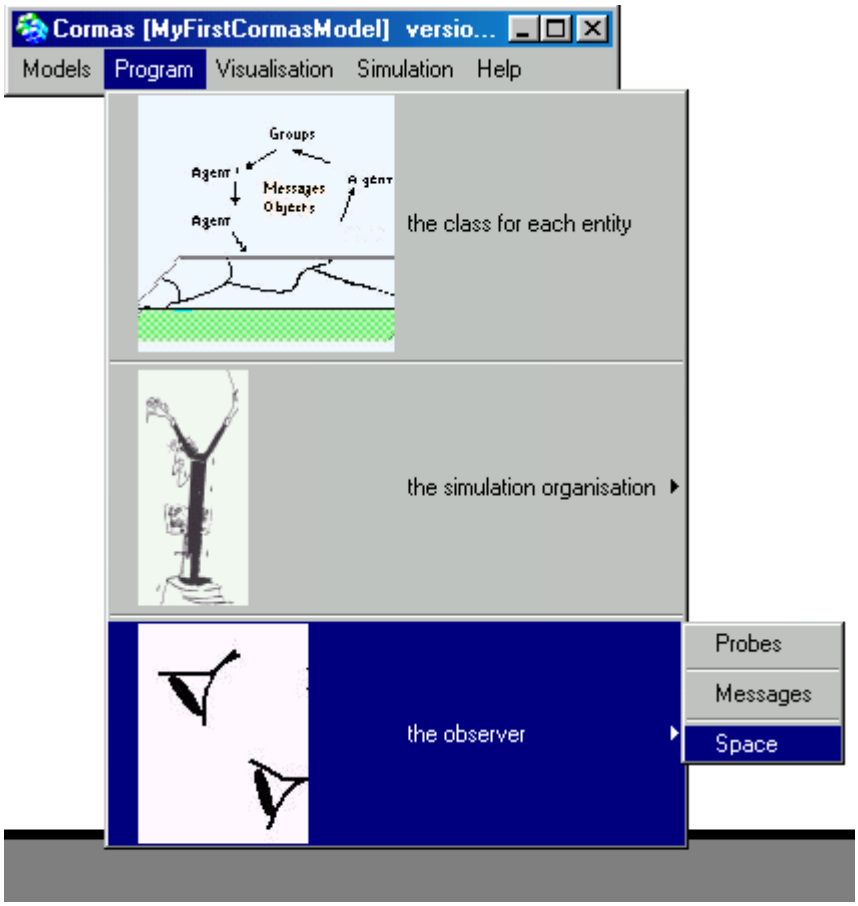


By clicking on the grid button of Visualisation menu of the Cormas window you can open the grid window where you will see a 10 by 10 grid. You can extend it to 30 by 30. In the 'Topology' menu, you select 'Grid size'. A window opens where you can change the number of lines and columns. It is also important to set the connectivity (number of neighbours) to 8 by selecting in the 'Topology' menu, 'Cell shape', 'Square' and '8-connexe'. It is the case when we consider the cells not only north, south, east and west but also the cells in diagonal.



The 'Grid boundaries' must also be 'toroidal' (same menu: Topology → Grid boundaries → Close).

3.5 Defining a point of view



By clicking on the 'The observer -> Space' submenu of 'Program' menu, you open a window to define a point of view.

By clicking on the 'Cell', you will find the possible methods to observe the 'Cellule_Conway' state. By double clicking on a method you will find the possible states (symbols) returned by the corresponding method :

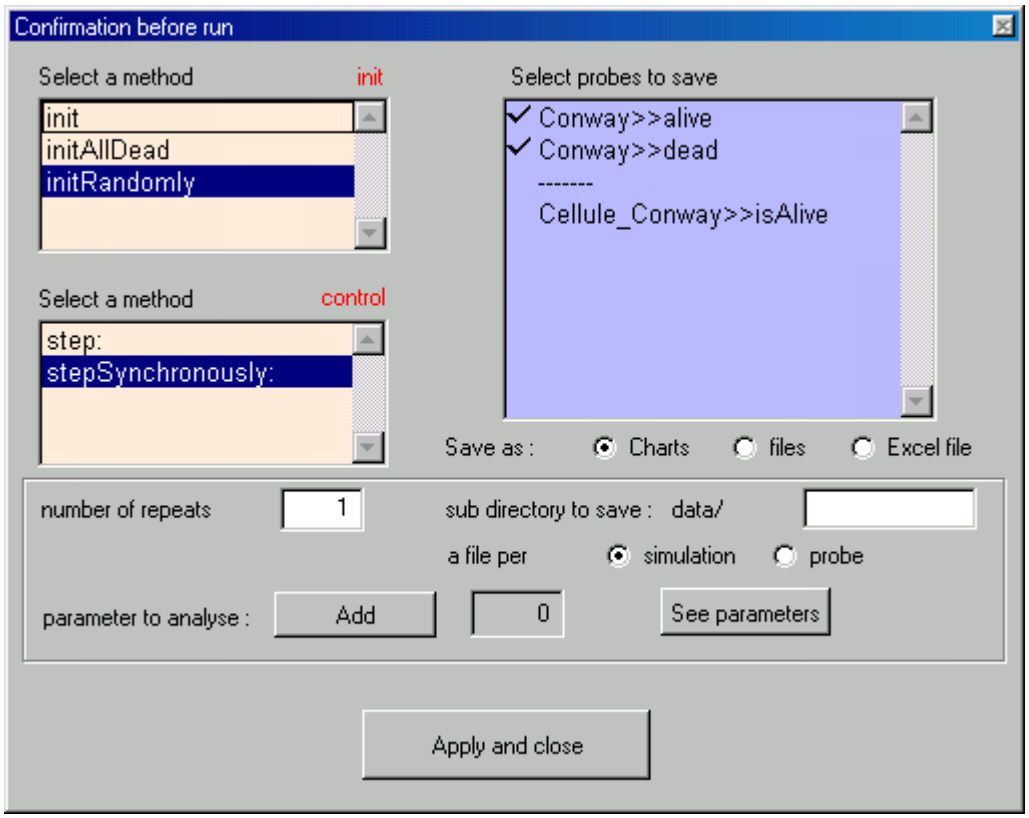
```
pdv
^state
```

You can remember that the state of a given cell is a Symbol which can always be #dead or #alive, in our case.

It is possible to associate a colour to each value. In our case, alive is black and dead is white. For each new colour associated to a symbol, don't forget to click on the Apply button.

3.6 Choosing a scenario for a simulation

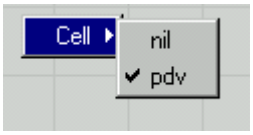
When clicking on the 'Initialise…' button in the Cormas window, you will have the list of initialisation methods (those defined in the 'init' protocol of the model) and the list of possible methods to make the automata evolve (those defined in the 'control' protocol of the model).



Select both' initRandomly' and 'stepSynchronously:' . Then select the probes which will compute the charts. Then click on the button 'Apply and close'. The first method will be called immediately.

3.7 Choosing a point of view

If you right-click on the grid window, you will have the choice between different points of view, in this case only 'pdv' or 'nil' (nothing).



Select 'pdv' and the states of the cells will appear on your grid (black for alive and white for dead).

3.8 Running the simulation

3.8.1 Simulation of the current scenario

At this stage you can either:

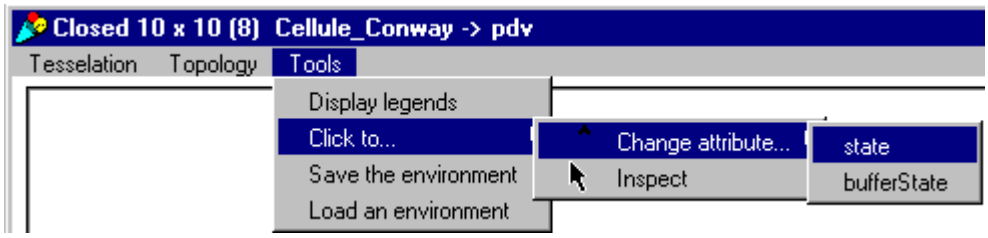
- Click on the 'Step' button as many times as you want and observe the evolution after each call to 'stepSynchronously:';
- or set the final time (in number of steps) and click on the 'Run' button; the simulation will run until the final time is reached.

3.8.2 Choose another initial state

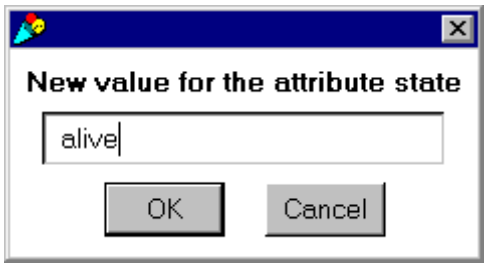
If you re-initialize the state of the cells by clicking on the "Initialize…" button, the state of each cell will be determined randomly.

But if you prefer to define yourself the initial state of the cells, you can follow this way:

Click on the "Initialize…" button and select 'initAllDead' (this method will call the "initDead" method of each cell). Then, by selecting in the 'Tools' menu 'Click to…' then select 'state'



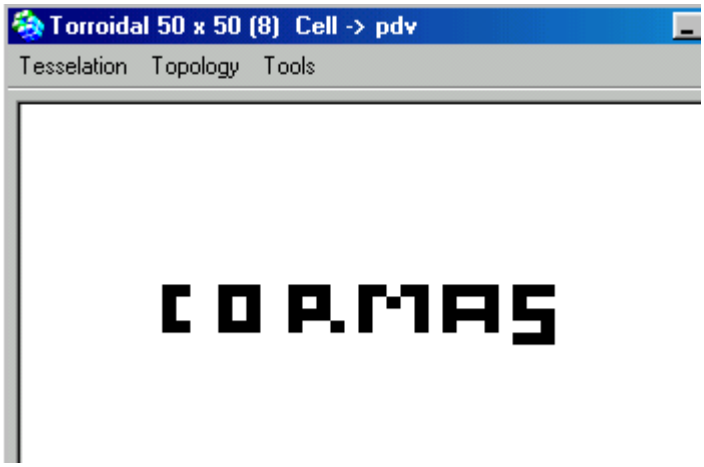
Then a window pops up. Write the new state you want to assign to the cells and ok



You can now design the new leaving cell by clicking on the grid.

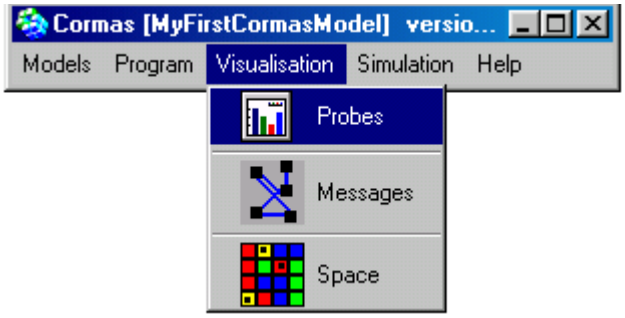
Or, you can load an existing initial state, saved on the disk, by clicking on the 'Tools' menu and selecting the "Load an environment" label.

You can also select one of the 2 files that predefine a given grid. For example :

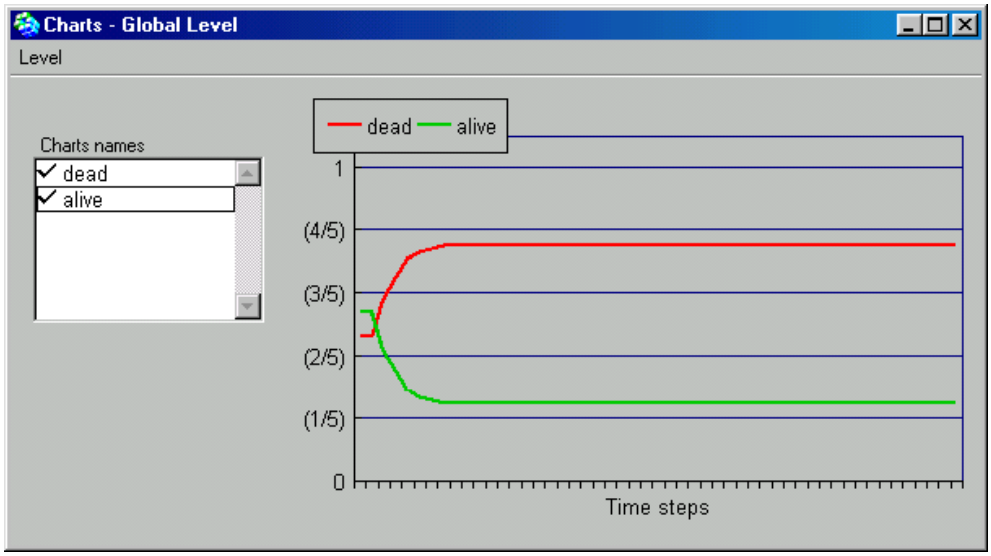


3.9. Viewing the charts

Click on the visualisation of the charts.



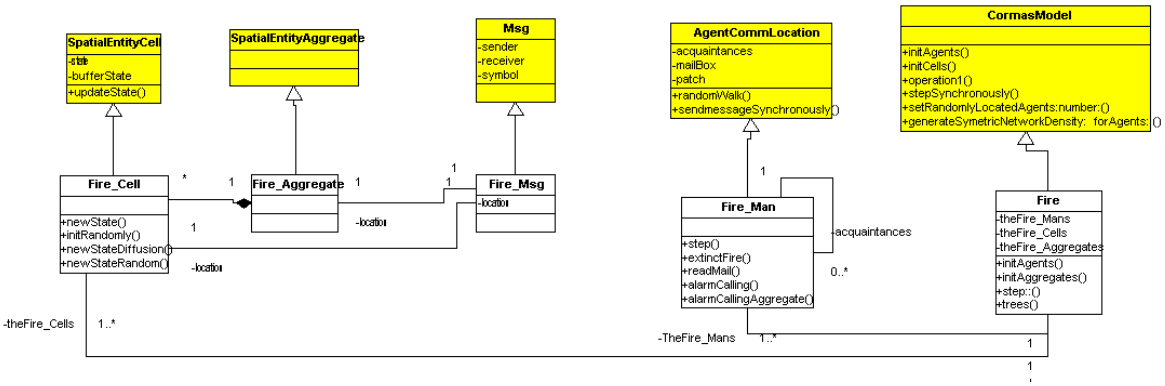
You can observe the global evolution of your system (deads and alives) or the evolution of a single entity (cellByCell).



To view two curves simultaneously on the same window, “control – click” on the second parameter.

4.CONSTRUCTION OF AN EXAMPLE

In this chapter, you will build a simple model: "Fire". The aim of this model is to simulate fire diffusion in a forest and firemen trying to extinguish the fire. Below, you can see a simple class diagram of this model :

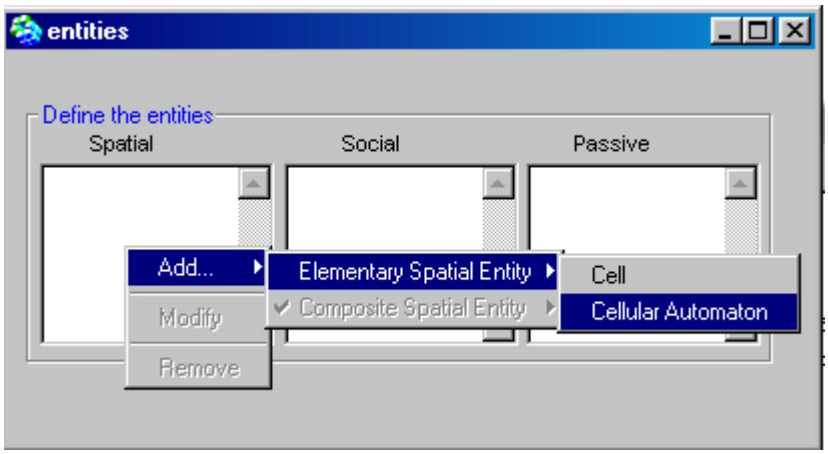


In this diagram, the information on the attributes is redundant : it is designed as field and as relationship. The yellow classes are classes from Cormas. You can see a class diagram of the CormasKernel-Entities category at the end of this document.

In the Models menu you select "new" model and you enter the name Fire as the new model name.

4.1. Defining a spatial entity

The second step consists in defining the spatial entities. By right-clicking in the list of spatial entities you get a popup menu where there is the add functionality. You have the choice between elementary entities or composite entities, choose the first one. You then have the choice between cell and Cellular Automaton. The first one do not define any state neither bufferState and the cellular automata methods, so choose the second one.



You are then requested to enter the name and the proposed name is Fire_Cell. Just accept it and a browser will be open in which the init method is already created with a default behaviour.

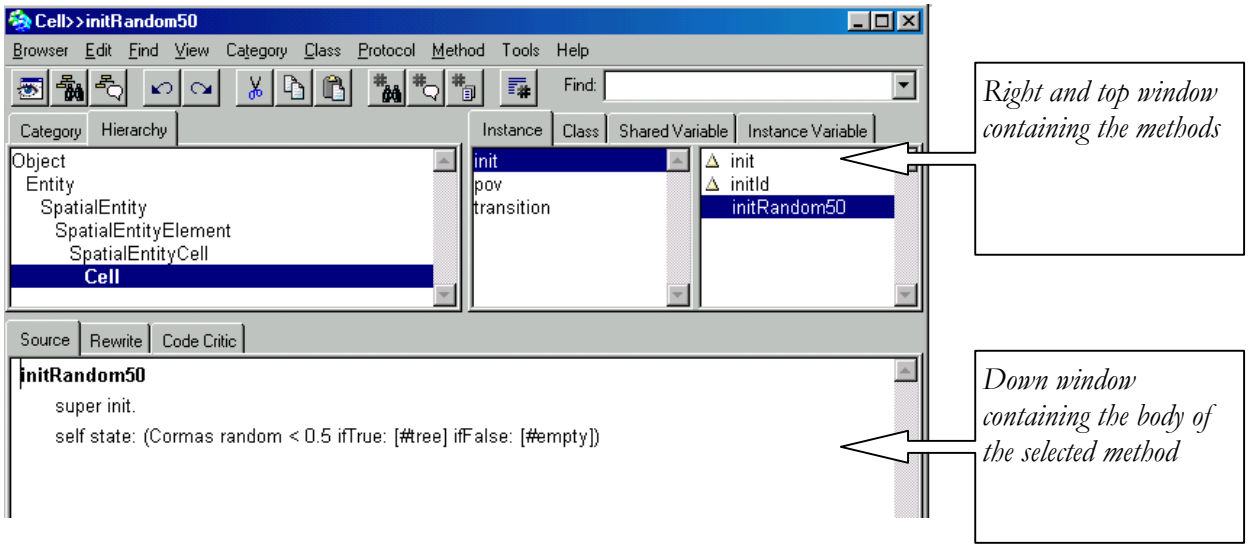
Select the 'init' method and at the end of the method, write the instruction:

(Cormas random < 0.5 ifTrue: [#tree] ifFalse: [#empty])

or even better, create a new method with the following code:

```
initRandom50
    super init.
    self state: (Cormas random < 0.5 ifTrue: [#tree] ifFalse: [#empty])
```

To create a new method, just select an existing method and over write the old code on the body of in the down window with the new code. Then accept : To accept a new method, right-click on the down window and select "accept". The new method is then add in the right and top window. The old method has not been changed.



You can create other methods for various values of the probability.

In order to test the new cell, close the browser and in the Cormas window, click on 'Program the simulation organisation-> the initial instanciation' button. A browser is opened with the 'Fire' class already created. The 'ini't method is already defined with the initialisation code. The line:

```
self initCells.
```

Calls 'init' on each cell. Because we want to use the method 'initRandom50' instead of 'init', you have to define a new 'init' method this way:

```
initRandom50
    self initCells: #initRandom50.
    self initAgents.
    self initData.
```

The argument of the 'initCells:' message is the name of the message to send to each cell.

4.2 Defining a point of view for the cells

After closing the browser (after "Accept", of course), you define the observation of space by selecting the space in the 'Program -> the observer -> Space' menu.

Select the 'Fire_Cell' and by right-clicking on the 'Methods' list, you get a window to define an reader method in the following way:

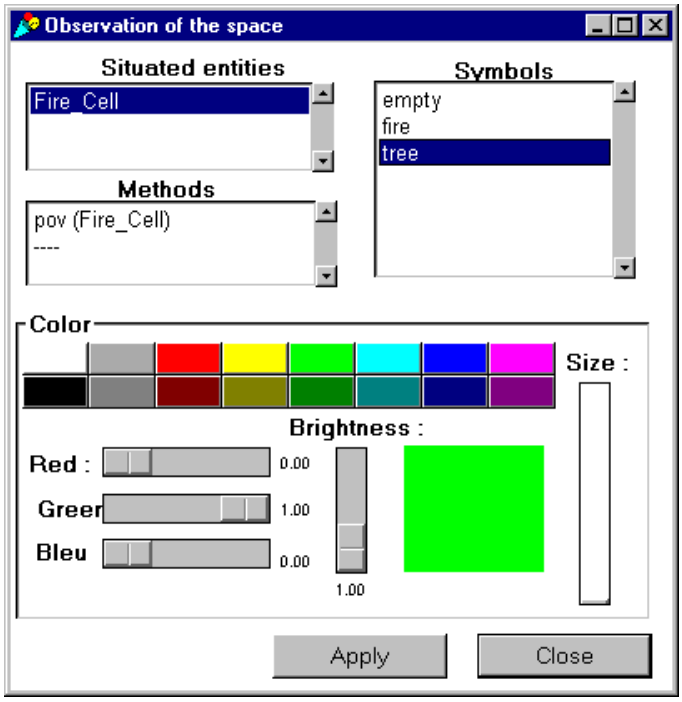
```
pov
    ^self state.
```

By accepting this definition, the name of this new method will appear in the list of methods. In the 'Definition' list you have to define the possible values returned by the selected method.

Just do it by right-clicking in the window and select 'Add'. A window opens in which you have to enter the name (without #!). Do it for each of the possible values:

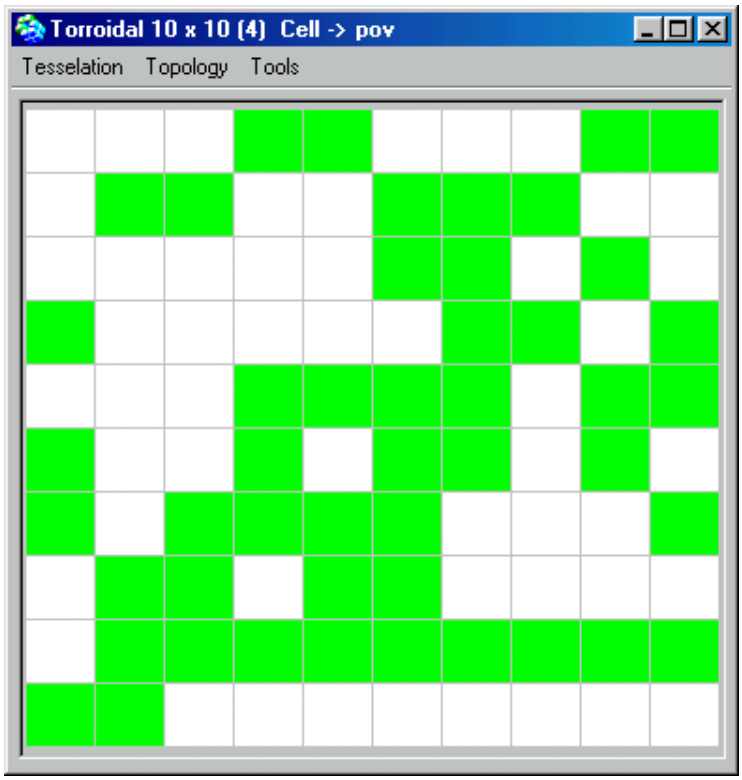
```
empty
tree
fire
```

Then, you must associate a colour to each of these values. Just select the value, select the corresponding colour and then click on the 'Apply' button. For example, use green for the tree, white for empty and red for fire. When you are finished with this work, close the window.



Now to visualize the resulting initial state of the cells, click on the grid button  of the 'Visualisation' menu of the Cormas window. The grid window will appear.

Select the 'Simulation->InterfaceSimulation' button and click on the button Initialise. Then select the method 'initRandom50' for initialisation and 'step:' as control method (it does not do anything at this stage). The grid window should display randomly put green tiles on white cells.



IF NOT, CHOOSE A POINT OF VIEW (CF. CHAPTER 3.7 Choosing a point of view) **:**

4.3. Defining the spatial entity dynamics

Double-click on the 'Cell' to open again the browser. Add a 'control' protocol and add the method 'newStateRandom':

```
newStateRandom
    "fire chance for a tree cell"
    (self state = #tree and: [Cormas random < 0.001])
          ifTrue: [self bufferState: #fire]
          ifFalse: [self bufferState: self state]
```

This means that a tree has a low probability to spontaneously put itself into fire. Because Cormas expects the method 'newState' to be defined as the standard behaviour, you must also define this method which simply calls your transition function:

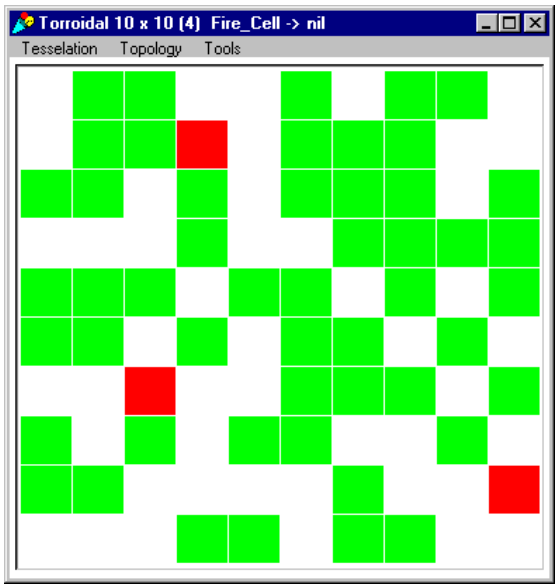
```
newState
    self newStateRandom
```

Click again on the 'Prepare and Schedule' button to open the browser on the 'Fire' model. We will now define the dynamics. In the class 'Cormas_Model', two methods are already defined: *StepSynchronously* and *stepAsynchronously* which automatically sends the appropriate messages ('newState' and 'updateState') to all the cells. In the class 'Fire', select the 'step:' method in the control protocol and make the following modification:

```
step: t
    "The main method of the model. t parameter is the current time (integer)"
    self stepSynchronously: t.
    self updateData: t.
```

Now you can click on the 'Initialise…' button of the Cormas window to select the initialisation you want (initRandom50) and the stepping function (step:).

By clicking repeatedly on the 'Step' button, you will have some cell changing to red, showing the method is correct.



To complicate the behaviour you can add the diffusion of fire over the grid. It is enough to say that if at least one neighbour is in fire, the cell begins burning if it is a tree. The resulting method is the following:

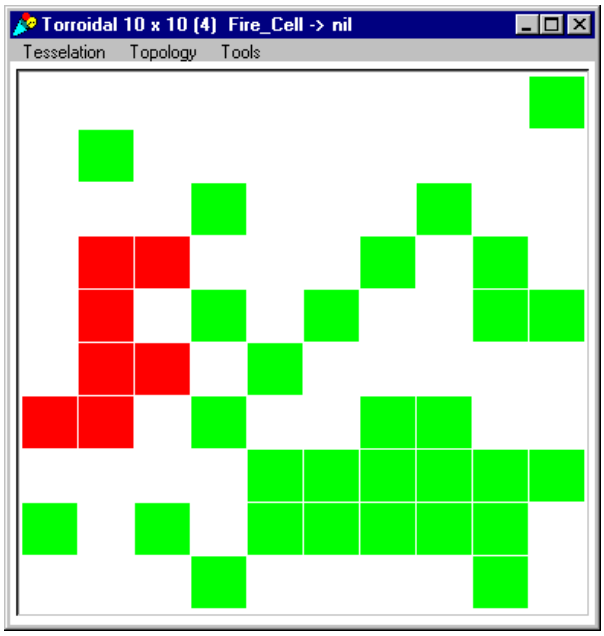
```
newStateDiffusion
    "The cell has 10% chance to become in fire if there is fire in its neighbourhood "
    | fireAround |
    fireAround := self neighbourhood contains: [:aCell | aCell state = #fire].
    (self state = #tree and: [fireAround and: [Cormas random < 0.1]])
        ifTrue: [self bufferState: #fire]
        ifFalse: [self bufferState: self state]
```

To test it, just modify the 'newState' method:

```
newState
    self newStateDiffusion
```

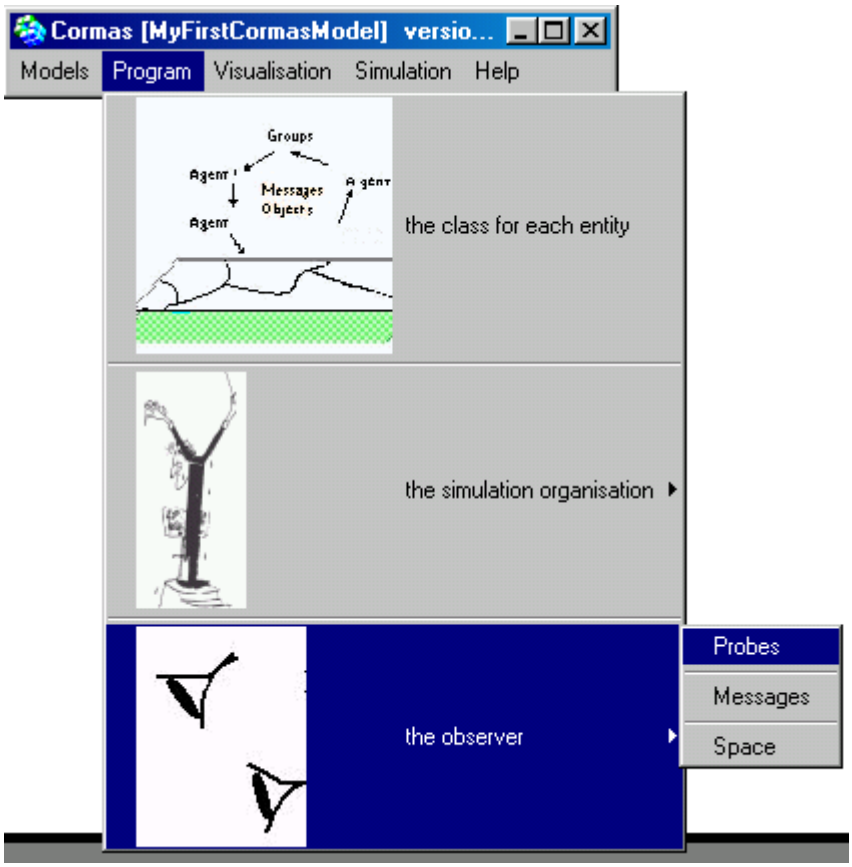
You will need now to locate the fire by yourself. On the grid interface select ' 'Click to…Change attribute… state'. Type fire and click anywhere on the grid. Fire should appear.

By clicking the 'Initialise…' button and then the 'Step' button repeatedly, you will be able to observe the fire propagation:



4.4. Defining a chart

To define a chart for your simulation, you define an observation on probes.



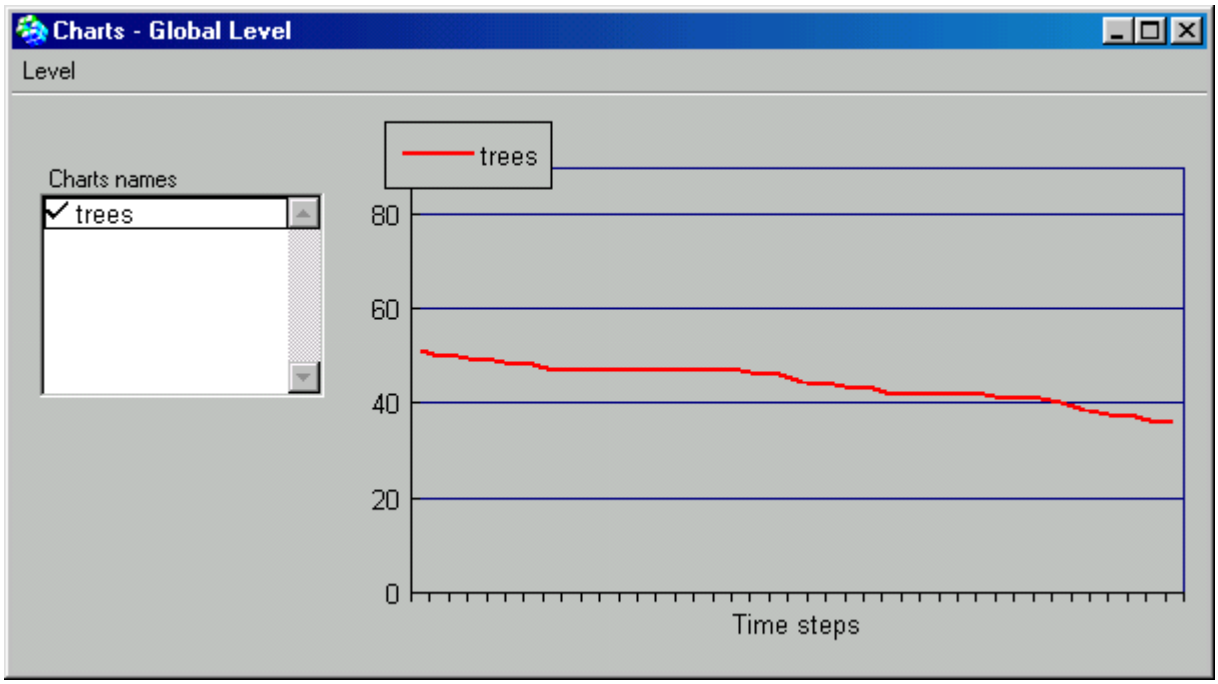
A window open with an empty list of charts. By right-clicking in the window, you can select 'Add'. You can either define a chart at the level of the population or at the level of the entity. Select the global level and enter a new name, for example 'trees'. A browser opens in which you must define a method returning the ratio of trees on the total number of cells under the method trees:

```
trees
```

"return the data (a number) to be plotted with the tree chart"

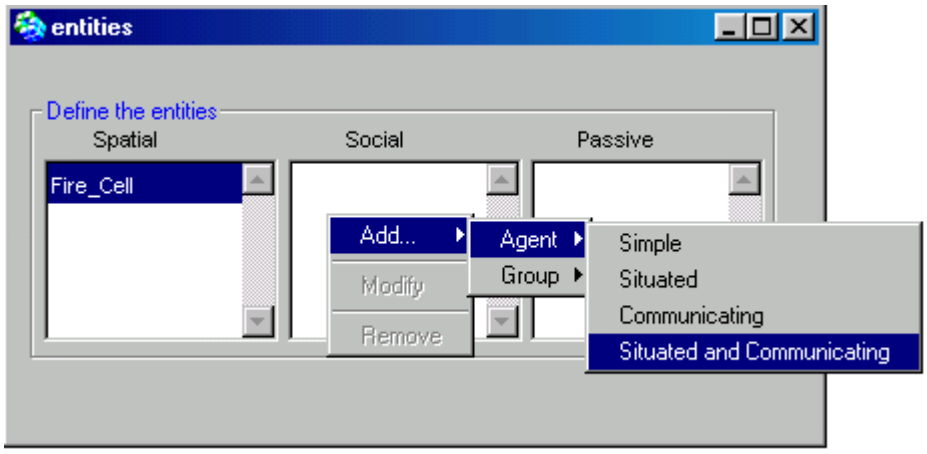
```
^(self theCells select: [:aCell | aCell state = #tree]) size
```

Note that the method name is imposed and that we access the collection of cells created by the simulator by an automatically generated attribute: 'theCells'. If you run the simulation again for a number of steps, open the chart window ('Visualisation' pane of the Cormas Window) and click on the 'tree' chart, you will have a result similar to this:



4.5. Defining the agents

In the 'Define the entities' pane, right-click on the social entity list and select an agent which is both situated and communicating.



Enter the name 'Fireman'. Situatedness implies the knowledge of where the agent is located as the value of the 'patch' attribute.

Several methods are already defined in the super classes:

- In the 'procedure' protocol:
  - 'leave' to leave a cell;
  - 'moveTo:' to go to a cell;
  - 'randomWalk' to let the agent walk around.

In the 'Fireman' class, define the 'step' method:

```
step
    self randomWalk
```

4.6 Initialisation and scheduling of the model

If you click on 'Prepare and Schedule' button, you will find the 'Fire' model with new attribute: 'theFiremans' (sorry for the bad plural). In the 'instance-creation' protocol, the 'initAgents' method is already created and must be updated to create a number of agents, locate them and add them to the collection:

```
initAgents

    super initAgents.
    "create the population"
    self setRandomlyLocatedAgents: Fireman n: 5
```

Just check that your init method in the 'init' protocol actually calls the 'initAgents' method. In the 'control' protocol, select the 'step:' method in order to make the agents behave:

```
step: t
    self stepSynchronously: t.
    self theFiremans do: [:fm | fm step].
    self updateData: t.
```

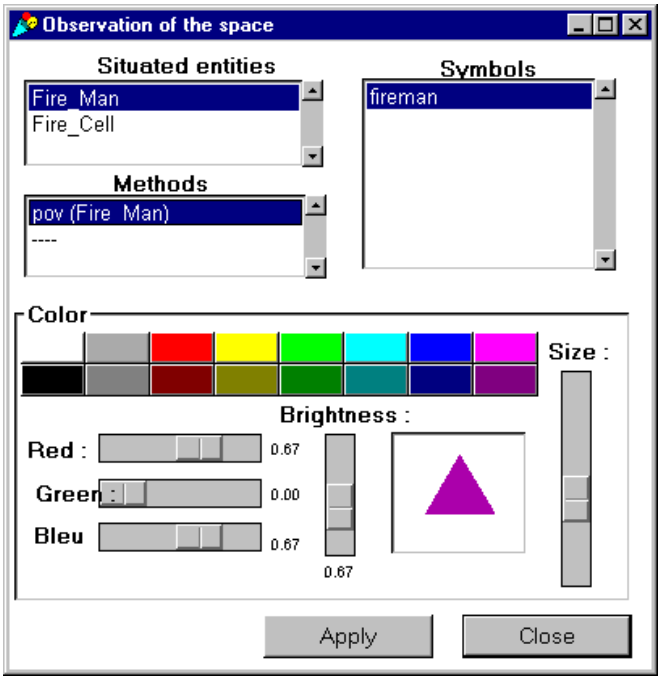
Note that if the method 'stepSynchronously:' exists to evolve the cellular automata, you have to define the way the agents are executed yourself.

4.7. Defining a point of view for the firemen

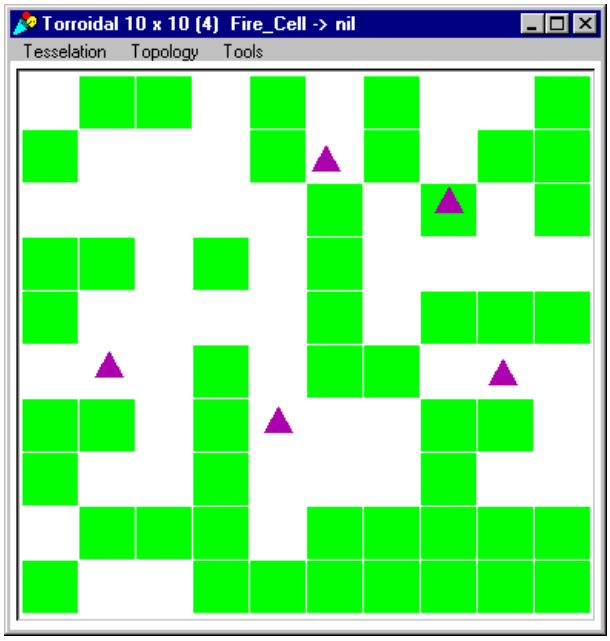
Now we have to define the point of view on the agent by selecting Fireman in the space observation window defining a pov method:

```
pov
^#fireman
```

Here the symbol returned does not have any meaning: it is just to have something to display. Define the fireman symbol and associate it with a color (the default shape is a triangle). The size of the triangle can be changed by the slider on the right of the shape:



Finally, you can "Apply" and close the window and run the simulation where you should see the firemen crowling around:



4.7 Defining new agents dynamics

The next step consists in allowing a fireman to extinct the fire when he is on a cell on fire. In order to do it, the attribute 'patch' contains the cell where the agent is located. The code may look the following:

```
step
```

```
"The main method of the fireman.
It walk randomly; if it arrives on a cell in fire, it extinct it"
    self randomWalk.
    self extinctFire
```

with the "extinctFire" method like :

```
extinctFire
    self patch state = #fire ifTrue: [self patch state: #empty]
```

If you test this new behaviour, you will observe that the agents are just changing the fire put a long time ago.

Of course, this behaviour is quite simple-minded. An additional complexity is to allow the agent to go to a fire cell if he perceives it in its immediate surrounding:

```
step
| firedNeighbours |
firedNeighbours := self patch neighbourhood select: [:cell | cell state = #fire].
firedNeighbours isEmpty
    ifTrue: [self randomWalk]
    ifFalse: [self leave. self moveTo: (Cormas selectRandomlyFrom:
firedNeighbours)].
self extinctFire
```

If you want your fireman to be very efficient you can have all the neighbouring cells within a given radius:

```
    (self perception: 3)
```

We can see that the role of the first lines of this method is to move the fireman, either randomly or towards the fire. It seems so preferable to group them together in a new method :

```
move
| firedNeighbours |
firedNeighbours := self patch neighbourhood select: [:cell | cell state = #fire].
firedNeighbours isEmpty
    ifTrue: [self randomWalk]
    ifFalse: [self leave. self moveTo: (Cormas selectRandomlyFrom:
firedNeighbours)].
```

The step method is much more clear now :

```
step
 "It walks towards fire or randomly; if it arrives on a cell in fire, it extinct
it"
self move.
self extinctFire
```

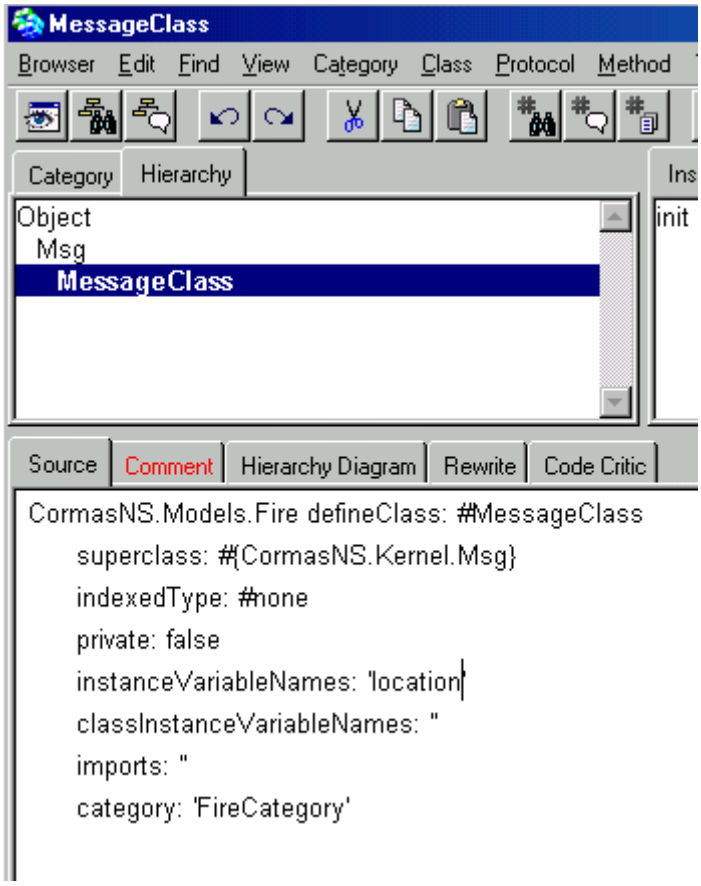
4.7. Sending messages

The behaviour of your firemen is very individualist for the moment. An additional complexity is to allow the agent to co-operate. If you want your fireman to be more co-operative, you can add them the ability to call the other in case of extensive fire. For that, you can use the Message class.

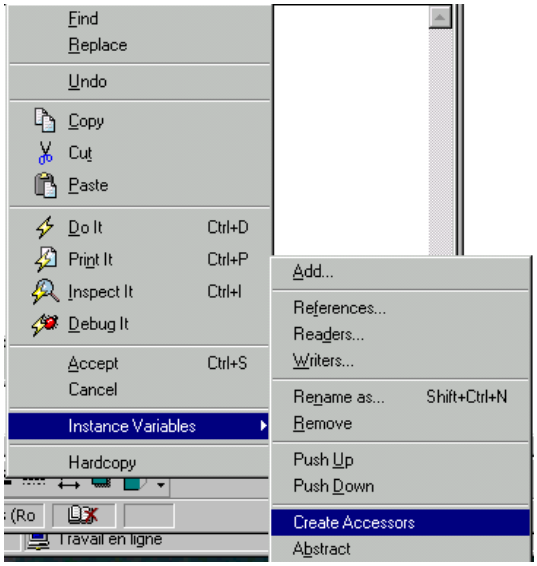
4.7.1. Create a new message

By right-clicking in the list of "Passive" entities on the Cormas interface, you get a popup menu where there is the add functionality. You have the choice between "Message" or "Object", choose the first one. You are then requested to enter the name and the proposed name is MessageClass. Just accept it and a browser will be open.

Now, you can add a new attribute (location), which is the place where the receiver-fireman can go to give help. For that, just write 'location' after the "instanceVariableNames" field, like in figure below :



Then to create the accessing methods, double click on location and with the right click choose 'Instances variables ->create accessors'



4.7.2. Connecting the Fireman to its acquaintances

And now, you have to inform each fireman of its colleagues. For that, you need to come back in the main class of the model: Fire and add the following instructions at the end of the 'InitAgents' method :

```
    self generateSymetricNetworkDensity: 1 forAgents: Fireman
```

Your method should now look like that :

```
initAgents
    super initAgents.
    self setRandomlyLocatedAgents: Fireman n: 6.
    self generateSymetricNetworkDensity: 1 forAgents: Fireman
```

4.7.3 Allowing the Fireman to send and read Message

A fireman agent (an instance of Fireman class) must know the other agents to send them messages. Any agent that has been declared as communicating agent has an attribute acquaintances which is to be used as an address book. It has also an attribute mailBox in which he receives messages sent by the other agents.

Open a browser on Fireman class.

For the Fireman, "Alarm calling" method.

```
alarmCalling
"Send a message to its colleagues if their is more than 1 cell in fire around
him"
| alarm firedNeighbours |

    firedNeighbours := self patch neighbourhood select: [:cell | cell state =
#fire].
    firedNeighbours size > 1 ifTrue:
                [alarm := MessageClass new.
                alarm sender: self.
                alarm symbol: #alarm.
                self acquaintances do:
                            [:x |
                            alarm location: (Cormas selectRandomlyFrom:
firedNeighbours).
                            alarm receiver: x.
                            self sendMessageAsynchronously: alarm]]
```

For the moment, the firemen are able to send and receive messages. But they don't treat them when they receive them. So, add the "readMail" method :

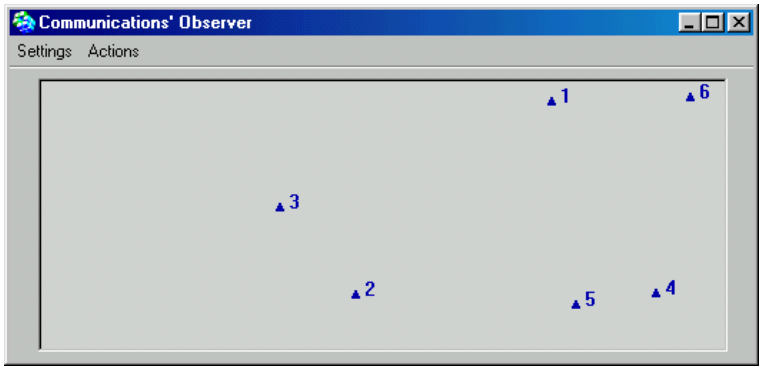
```
readMail
    | aMessage |
    aMessage := self nextMessage.
    aMessage notNil
          ifTrue:
                [self leave.
                self moveTo: aMessage location]
          ifFalse: [self move]
```

Now, you have to modify the Fireman step method in order to use this new functionality's:

```
step
    self alarmCalling.
    self readMail.
    self extinctFire
```

Then, you can now see if your messages are sent. For that, open a new windows by selecting in the Cormas interface on the button  in the middle of the "Visualisation" menu.

With this interface you can observe the communications and who communicates with whom. Have a look at the users guide to know more about the observation of exchanges of messages.



From object to agent

Agents are autonomous since they decide what to do with depending on the received stimuli, its own resources and its goals. The behaviour of an agent consists of a set of decision rules leading to action selection. In our example, firemen receive alarm message (external stimuli) but are not oblige to react to it. We want them to choose according to their goals : to achieve their current task if necessary or to "fly" to the rescue of the sender of the message. So you can complexify the readMail method.

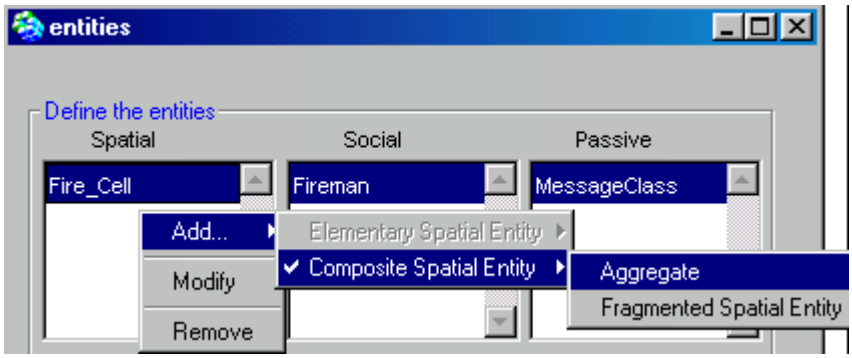
4.8 Define spatial entities at upper scale.

There are many ways to improve the previous model. It is an interesting model because you may improve it either by improving the individual skills of the firemen, or by improving the coordination among firemen by letting them communicate more.

In the following slides, to practice the spatial entities, we will introduce a new entity, the fire, as an aggregation of contiguous Cells in fire. The agents will decide to call for help if the fire is large enough, or depending on the spatial properties of the fire.

4.8.1.Creating the spatial entity

On the main interface of Cormas locate the mouse on Spatial entities and add an entity Aggregate.



Create the class Aggregate. There is no code to implement at the level of this class

4.8.2. Define a point of View on the Aggregate

Select Space on the observation. Then click on the Aggregate and on the method part right-click to add a point of view. Type and accept the following code.

```
pov
    ^self size > 10
          ifTrue: [#needHelp]
          ifFalse: [#noHelpNeeded]
```

Add these two symbols and associate colors.

4.8.3. Modifying Fireman perception

```
alarmCallingAggregate
    "Send a message to its colleagues if the fire it perceives is larger than 10
cells "
    | alarm firedCell aggregate |
    firedCell := self patch neighbourhood detect: [:cell | cell state = #fire]
ifNone: [^nil]..
    aggregate := firedCell theCSE at: #Aggregate.
    aggregate size > 10
          ifTrue:
                [alarm := MessageClass new.
                alarm sender: self.
                alarm symbol: #alarm.
                self acquaintances do:
                      [:x |
                      alarm location: (Cormas selectRandomlyFrom: aggregate
components).
                            alarm receiver: x.
                            self sendMessageAsynchronously: alarm]]
```

Then change the step method of the Fireman.

```
step
    self readMail.
    self alarmCallingAggregate.
    self extinctFire.
```

4.8.4.The dynamics of the aggregates

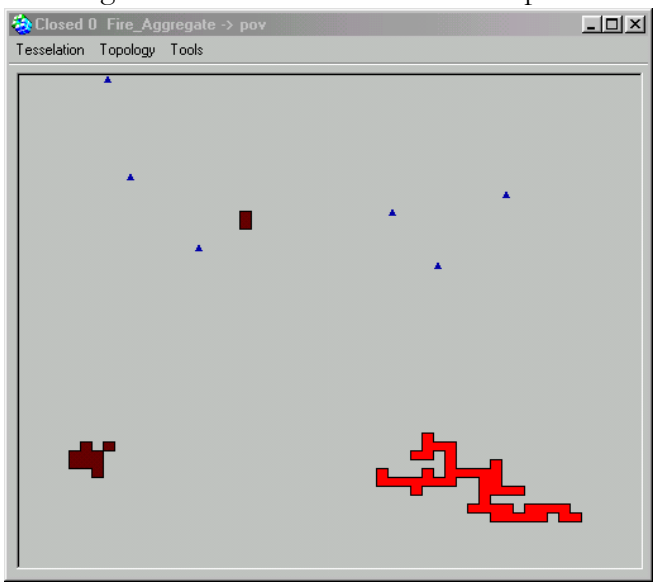
Cormas has procedures to create aggregates in many ways (See for instance the TSE exercice). To manage the aggregate, click on 'Prepare and Schedule' and in the instance-creation protocol add the following method.

```
initAggregates
    self spaceModel
          setAggregates: Aggregate
          from: Cell
          verifying: [:a | a state = #fire]
```

Then modify the step method consequently:

```
step: t
    self stepSynchronously: t.
    self initAggregates.
    self theFiremans do: [:fm | fm step].
    self updateData: t
```

Now you can simulate. On the spatial interface select the point of view on Aggregate. You will observe that the agents manage alone small fires but call for help when they perceive large fires.



4.9 Landscape indices.

Cormas provide some pre-computed landscapes indexes. Thus it is possible to observe the dynamics of these indexes.

Follow the same procedure than 4.3. You may add the following charts, one by one.

dominance

```
^self spaceModel dominance: #state
```

edgeDensity

```
^self spaceModel edgeDensity: Aggregate
```

fractalDimension

```
^ self spaceModel fractalDimension: Aggregate
```

meanCompactness

```
^self spaceModel meanCompactness: Aggregate
```

meanNearestNeigh

```
^self spaceModel meanNearestNeighbourDistanceAggregate: Aggregate attribute: nil
```

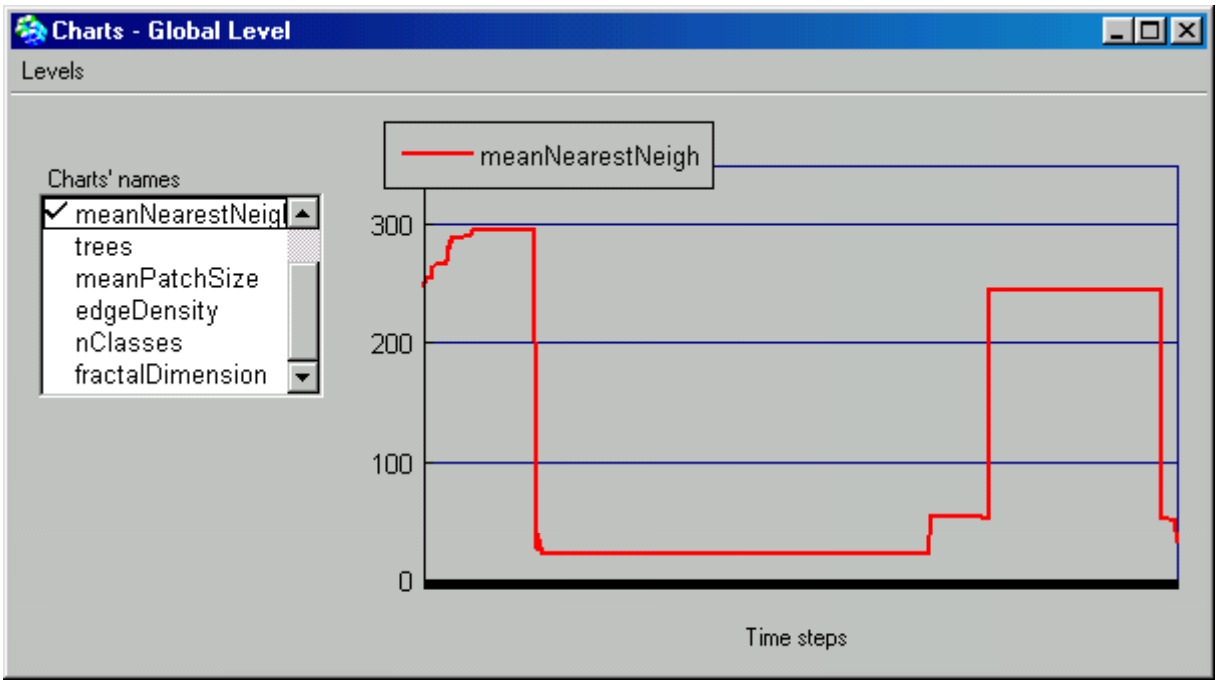
meanPatchSize

```
^self spaceModel meanPatchSize: Aggregate
```

nClasses

```
^self spaceModel nClasses: #state
```

For instance you will have the following chart for the mean nearest neighbour.


Charts - Global Level
Levels
Charts' names
meanNearestNeig
trees
meanPatchSize
edgeDensity
nClasses
fractalDimension
meanNearestNeigh
300
200
100
0
Time steps