Object-Based Environment for Urban Simulation

OBEUS

User's Guide

by

Itzhak Benenson and Vlad Harbash

bennya@post.tau.ac.il, vlad@eslab.tau.ac.il

Tel Aviv University

2004

Content

0. From the developers

We don't like to write code. We do like to explore the world — with models that simulate the world as a society of living as well as lifeless objects of different types, objects that act and interact. After devoting 90% of our time to writing and debugging the code used in our models – once in C and Pascal, then in Delphi and C++, now in Java, and after staying awake the night before a conference with the clear understanding that last month's results were all caused by an improper coding, we have decided to develop a system that reduces coding to just the things we think are inventive: objects' actions and interactions. This system that forces us to make everything perfectly clear to others just as to ourselves, but without imposing any limitations on our creativity.

We have based OBEUS on the achievements of computer science and state-of-the-art software technology. Yet, you will not feel the presence of all these influences when developing your model because OBEUS is intended for modelers' convenience, not to teach you computer science. The goal of object-based simulation is to make a science out of a commonsensical view of the world as a collective of behaving and interacting objects. For that purpose, we need commonsensical software – and here it is. We will be happy to share the principles of OBEUS's development with you, but only if you are interested.

OBEUS© was designed by Itzhak Benenson and developed by Vlad Kharbash. It is a product of work done at the Environmental Simulation Laboratory, the Porter School of Environmental Studies, Tel Aviv University. Slava Birfur, also from ESLab, took part in developing the current version.

OBEUS is based on the novel theory of Geographic Automata Systems, developed by Itzhak Benenson and Paul Torrens (Utah State University) in their book "Geosimulation – Automata-Based Modeling of Urban Phenomena" (Wiley, 2004). The book presents the wide view of the modern urban modeling and simulation and can serve an up-to-date guide for those who want to enter this exciting field.

We wish to express our gratitude to Shai Aronovich and Saar Noam, both from ESLab, who conducted the initial experiments in OBEUS implementation during 2002.

Itzhak Benenson, Vlad Harbash

Tel-Aviv, December 2004

1. OBEUS installation and setup

1.1. Download

Download OBEUS_Istall.exe from www.geosimulationbook.com. It is a large 250Mb file that contains the components necessary for OBEUS installation, most of which are third-party shareware software programs (Table 1). You can download them yourself, one by one, form Microsoft and Borland – we have simply combined them into one file and built an interface that installs them one after the other.

Software component	What is it? (Download file size)	Provider
Windows Installer 2.0	***Installation engine (1Mb)***	***Microsoft***
http://www.microsoft.com/downloads/details.aspx?FamilyID=4b6140f9-2d36-4977-8fa1-6f8a0f5dca8f&displaylang=en		
Internet Explorer 6.0	***Microsoft Internet Explorer (15Mb)***	***Microsoft***
http://www.microsoft.com/downloads/details.aspx?FamilyID=1e1550cb-5e5d-48f5-b02b-20b602228de6&DisplayLang=en		
.NET Framework 1.1	***.NET Environment (30Mb)***	***Microsoft***
http://www.microsoft.com/downloads/details.aspx?FamilyID=262d25e3-f589-4842-8157-034d1e7cf3a3&displaylang=en		
.NET SDK 1.1	***.NET Developer Kit (100Mb)***	***Microsoft***
http://www.microsoft.com/downloads/details.aspx?familyid=9B3A2CA6-3647-4070-9F41-A333C6B9181D&displaylang=en		
MSDE	***MS SQL Server desktop edition (70Mb)***	***Microsoft***
http://www.microsoft.com/sql/downloads/2000/sp3.asp		
MDAC 2.7	***SQL Server driver for the .NET (10Mb)***	***Microsoft***
http://www.microsoft.com/downloads/details.aspx?FamilyID=9ad000f2-cae7-493d-b0f3-ae36c570ade8&displaylang=en		
Borland C# Builder	***Borland C# shareware compiler (25Mb)***	***Borland***
http://www.borland.com/products/downloads/download_csharpbuilder.html#		

With standard ADSL connection, download takes about 15 minutes - enough time to have a glance at this manual.

1.2. Uninstall

The instruction is very short: after the installation, each third-party component of OBEUS, as well as OBEUS itself will appear in the list of the programs in **Add/Remove programs** control panel. You uninstall OBEUS components in a standard way, then.

1.3. Third-party components setup

OBEUS_Istall.exe is a self-extractable file; you have to click it twice and extract to some temporary directory. Execute the setup.exe file. The file scans the computer for third-party applications listed in the above table. It might happen that some of those applications, Internet Explorer 6.0, for example, are already installed in your PC. The OBEUS setup presents the results of the scan in the dialog box (Figure 1), where the necessary installation components are enabled and checked.

Warning: To execute .NET programs you need to install Internet Explorer 6.0 or above. This is not that bad even for those who hate Microsoft. Simply don't set MS Explore default browser after the installation, if you're in favor of Mozilla&C^0.

Click the "install" button (Figure1.1), and Setup installs the applications selected, one by one. You will have to click the **Next** button several times during the installation and answer **No** to the question "Would you like to reboot the computer right now?" Clicking "Yes" does not endanger the install process – should you do so, activate execute setup.exe after rebooting and it will install the remaining components.

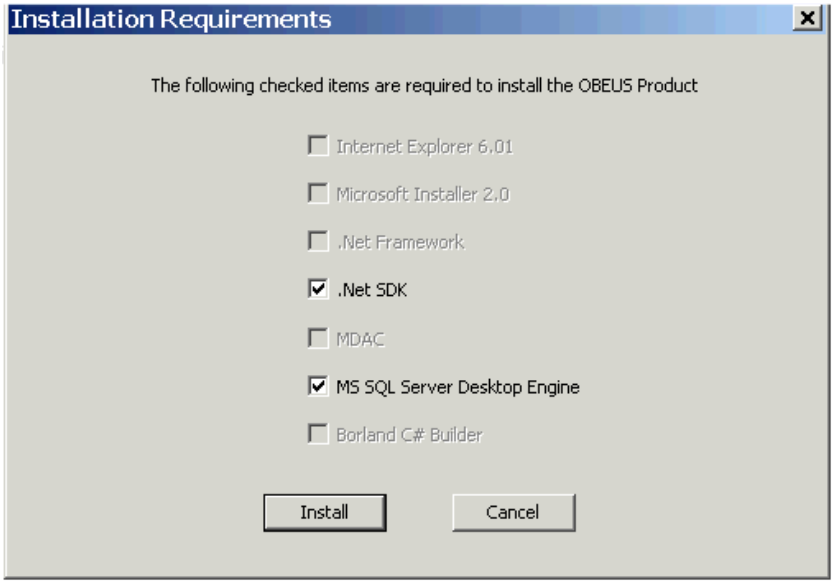


Figure 1.1: Initial installation screen

It takes about 20 minutes to install all third-party components.

1.4. The OBEUS core setup

After the third-party components are installed, you have to set up the location of OBEUS itself (Figure 1.2). Select the installation folder. To make it accessible to all users of your PC, choose **Everyone** and then click **Next**.

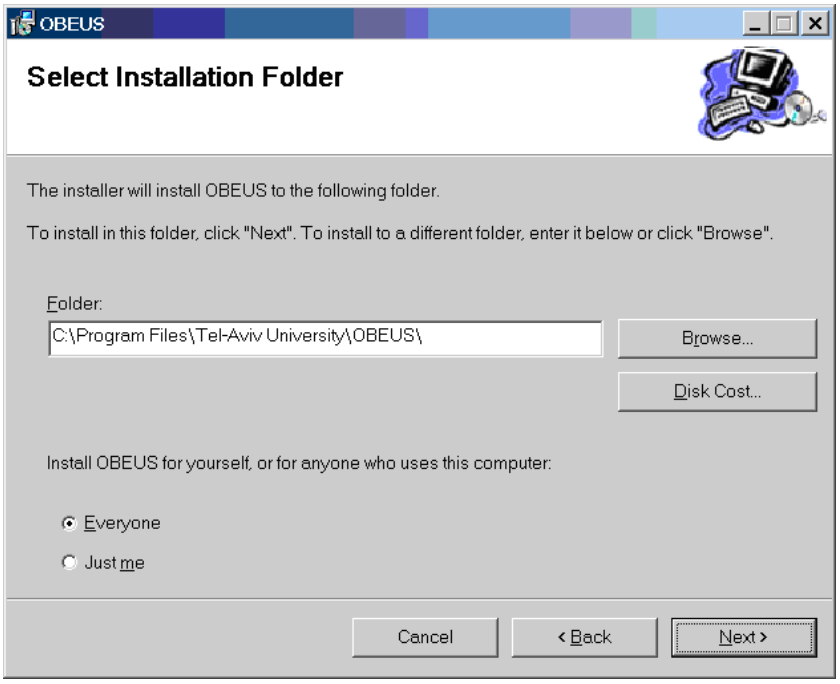


Figure 1.2: Select Installation folder

It takes about a minute and 32Mb of disk space to install OBEUS (Figure 1.3).

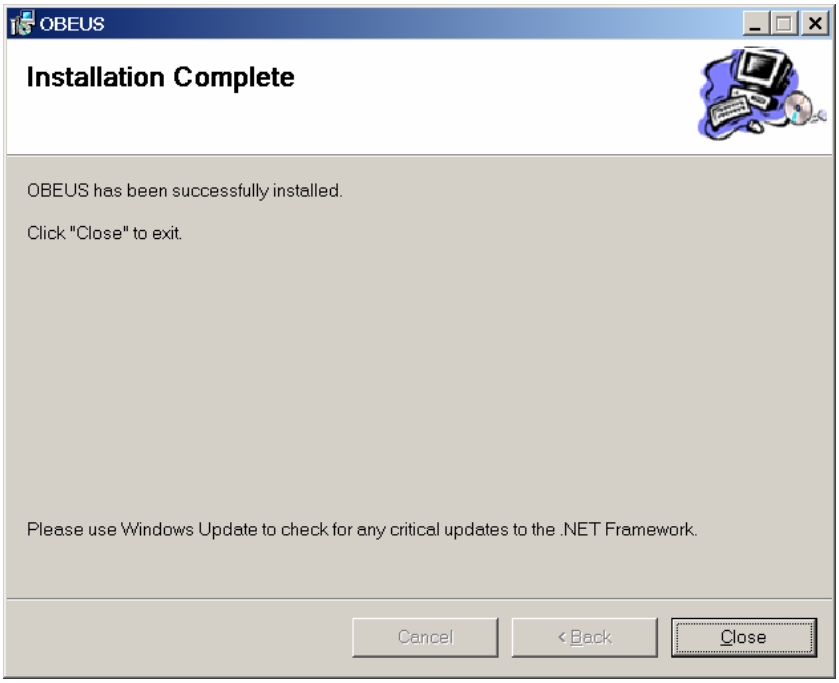


Figure 1.3: Installation Complete dialog

Click the “Close” button to exit the installation. Click **Restart** when OBEUS asks you to at the end of installation.

Shortcuts to OBEUS can be placed automatically on the desktop and in the Start>Programs menu.

1.5. Try it — It works

Click the OBEUS icon and run the OBEUS. Then, click **Project→Import project** in the OBEUS menu. Go to the **OBEUS /Hello/** directory and when importing call the project as **Hello**. Select **Project→Open** and choose **Hello** – the only entry in the list. You now see the **Model Tree** window of the **Hello** project (Figure 1.4).

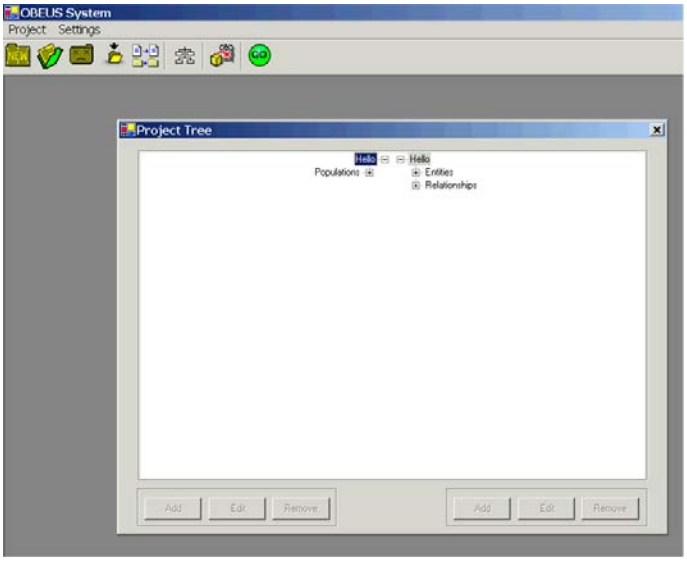


Figure 1.4: Model tree for the **Hello** project

Push **Go** button and OBEUS will prepare environment for the **Hello** project. Then push **Run** button on the new tool bar that appears. The result should be as shown in Figure 1.5. Contact us if something seems you wrong.

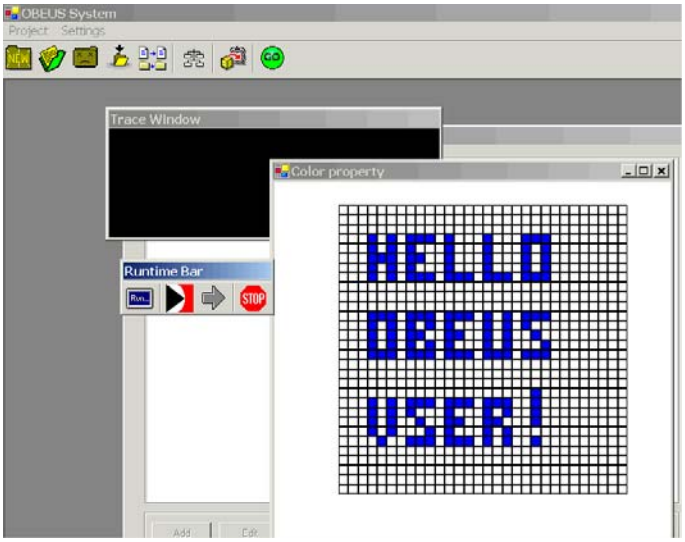


Figure 1.5: If you have reached this stage – OBEUS is properly installed and works

2. Before modeling with OBEUS

2.1. The concept of Object-Oriented Programming

Object-Oriented Programming (OOP) is a computer programming paradigm that emphasizes the following aspects:

- Objects - The basis of modularity and structure in an object-oriented computer program.
- Abstraction - Each object in the system serves as an abstract "actor" that can perform work, report on and change its state, and "communicate" with other objects in the system, all without revealing how these features are implemented.

A variety of techniques are required to extend an abstraction. The most important for OBEUS users is

- Encapsulation – Only the objects' own internal methods are allowed to access its state. Each class of object exposes an interface to other objects that specifies how the other objects may interact with them.
- Polymorphism - Invoking an operation that refers an object will produce behavior depending on the actual type of the referent, that is different behavior for referents of different types
- Inheritance – New classes can be defined as extensions of existing classes

OOP is often called a paradigm to stress that it changes the way in which programmers and software engineers think about software and develop it.

To implement the OOP approach, one needs an *OOP language*. C# is a modern OOP language that provides all the expected features: classes, interfaces, inheritance, polymorphism, encapsulation, etc. In addition, the C# language offers some new and powerful innovations.

For more information about OOP see:

http://www.samspublishing.com/articles/article.asp?p=101373

http://www.brpreiss.com/books/opus6/html/page588.html

http://www.intel.com/cd/ids/developer/asmo-na/eng/technologies/dotnet/using/dotnetapps/20012.htm

2.2. Geographic Information Systems (GIS) and Database Management Systems (DBMS)

OBEUS is based on GIS/DBMS view of the reality, which can be illustrated by the urban GIS, with layers (tables) of houses, public spaces, streets, regions and citizens, all related spatially via location and non-spatially via common keys (Figure 2.1, taken from Benenson, Omer, 2003). OBEUS can be considered as a dynamic GIS/DBMS – in addition to standard functionality, the objects of the OBEUS models can change their properties and location it time. New layers/tables of data, as well as aggregate views of the results are constructed, modified and stored by the model.

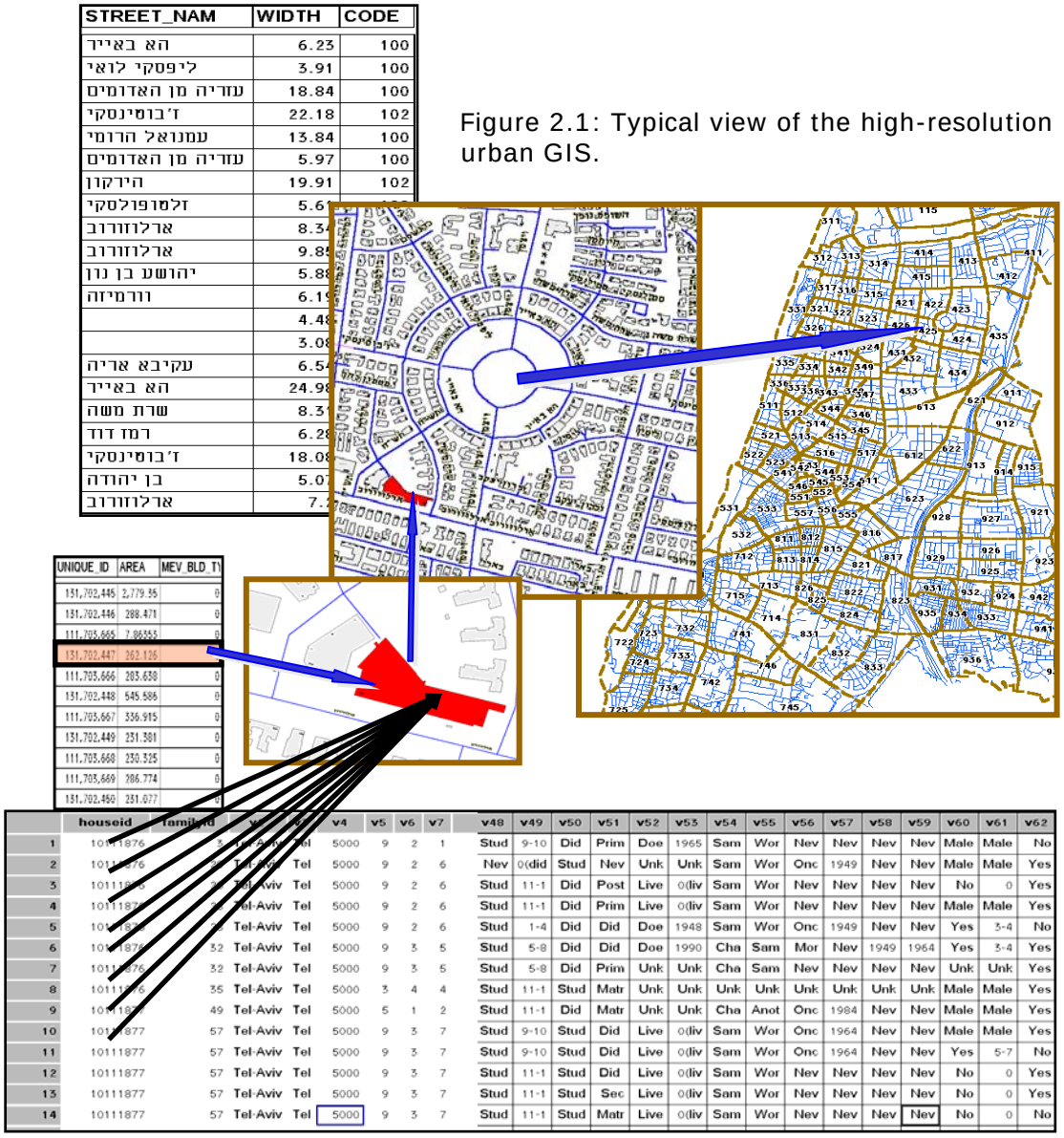


Figure 2.1: Typical view of the high-resolution urban GIS.

2.3. Why we choose .NET and not Java

This is an extension of Java idea, but better

3. Nearest future of OBEUS development

You have downloaded the alpha-version of the system. While we are applying it for our own modeling work for several month already, its functionality is yet ‘almost completed’, say nothing regarding bugs we ‘still have’, poor output, etc., etc. The main issues we plan to develop during nearest months (in order of our preferences) are as follows:

- Improvement of the map and graph output, including Replay option
- Enabling work with point and line entities;
- Development of additional general methods, such as a distance between two entities
- Development of built-in methods for the specific components of OBEUS, such as establishing initial conditions.
- Finalizing OBEUS components that deal with spatial self-organization;
- Development of built-in methods for specific types of models, such as the models of vehicle and pedestrian traffic.

4. Quick start with the Game of Life and Schelling model

Usually, we decide if the system is worth using after one or two attempts to do something simple but meaningful. If it goes smoothly, we continue to study the system; the chance that we won't use it drops to zero after less than an hour of experimentation. So, let's try doing something simple in OBEUS now, right after its installation. Our plan is to present the system in an hour, and then you could decide yourself whether to proceed or to uninstall (see 1.2 for instructions).

We have chosen two well-known models for this quick start. We begin with the Game of Life just because it takes one page to remind you of this simple but intriguing model. The second OBEUS test is done with Schelling's model of residential segregation (Chapter 5).

Note the numerous screen images inserted in the text below. These images are often only part of the full display – don't worry about something important beyond the image presented - we present all the relevant information.

4.1. Game of Life – short introduction

The Game of Life was invented by John Conway. His original intention was to design a simple set of rules that would produce a population's non-trivial spatial dynamics on the microscopic level (Berlekamp, Conway et al. 1982). Although aware of the computational universality of Cellular Automata (CA) and its ability to generate complex spatial structures, Conway looked for rules that while simple, could generate population dynamics that were not easily predicted or anticipated.

After a great deal of experimentation, Conway settled on a set of rules for a 2D 'population' of cells in a CA model; these cells could be in one of two states—dead (0) or alive (1). There are three rules to the Game of Life: A cell remains alive, dies, or is reincarnated depending on the number of live neighbors within its 3x3 Moore neighborhood (Figure 4.1).

Rule 1: Survival—a live cell with exactly two or three live neighbors stays alive.
Rule 2: Birth—a dead cell with exactly three live neighbors becomes a live cell.

Rule 3: Death owing to overcrowding or loneliness—in all other cases, a cell dies, or if already dead, stays dead.

These rules should be applied to all cells simultaneously.

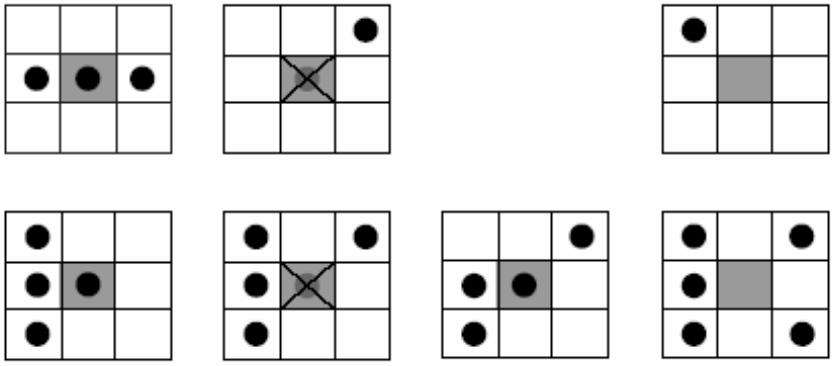


Figure 4.1: Illustration of Game of Life rules: A living cell in the center of a 3x3 Moore neighborhood (a) survives if it has two or three neighbors; (b) dies from overcrowding if it has four or more neighbors or from loneliness if it has only one neighbor. A dead cell in the center of a 3 x 3 Moore neighborhood (c) is born anew if it has exactly three neighbors; (d) remains dead otherwise

Surprisingly, the simple rules of the Game of Life support fantastic variation in simulations of growth patterns. Because the rules are so simple, the model can be reproduced quite easily and an amateur can get a taste of the complexity of the model from countless Internet sites devoted to 'Life' (Summers, 2000; Silver, 2003). I recommend www.math.com/students/wonders/life/life.html for a general introduction, while a good stand-alone version of the model can be obtained online at www.psoup.math.wisc.edu/Life32.html

4.2. Define new OBEUS project for the Game of Life

We will now formulate the Game of Life in OBEUS. As in any system, beginning a new model means first defining a new project and giving it a name - 'Game of Life' in our case. To do that in OBEUS, you click **New** button and then type the project's name in a dialog box (Figure 4.2).

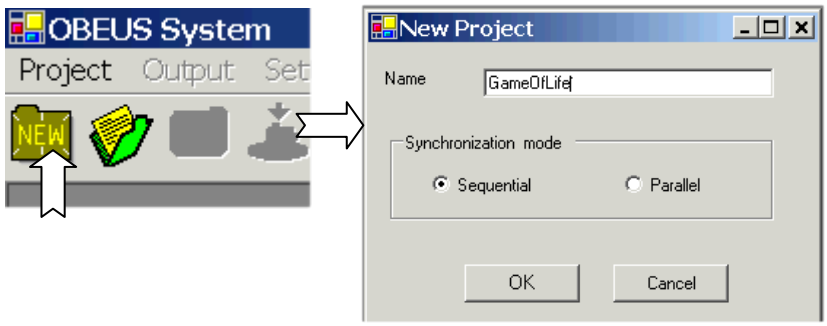


Figure 4.2: Open new project - sequence of action

The default synchronization mode of OBEUS models is **Sequential**. We will talk about this later; at this stage simply don't change the choice. If you follow this text with OBEUS turned on, you probably noticed that it took OBEUS quite a time (several seconds with standard configuration of your computer) to define a new model. What happens during these seconds of computer time? All this time is spent on construction of the database where all the future model's elements will be stored. We will discuss the advantages of this form of model storage in the next chapter. OBEUS is based on the Microsoft SQL Server database management system, and all its tables are created in SQL format.

4.3. Model Tree

After the database tables are defined, we can begin formulating the **Game of Life** model. It is done via **Model Tree** that opens automatically when you define a new model (Figure 4.3).

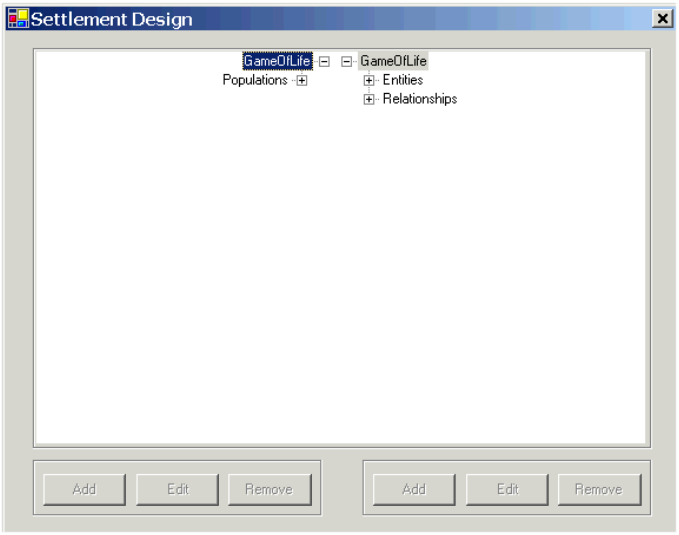


Figure 4.3: Initial state of the **Model Tree** of the **Game of Life**

4.4. Cells – entities of the Game of Life

In OBEUS, modeled systems are described as consisting of automated objects – *entities* - of different types that *behave* and *interact*. Note, that OBEUS is an inherently Object-Oriented system; consequently, in what follows you will always define *classes* of entities. The automated objects of the Game of Life are cells; you

thus have to define the entities of a type ***Cell***.

To define class of Cells, you click the *Entities* entry in the right branch of the model tree and then push the Add button at the bottom. A new dialog box pops up; you have to type in it the name of the new class of entities. Let us follow programming tradition and call new entity just 'Cell'.

In the same dialog box you have to set up the entity *type*, which is the basic property of OBEUS objects. Entities in OBEUS can be of one of two types – Mobile, which are generally labeled *Non-fixed*, and immobile, which are labeled *Fixed*.

OBEUS is an environment for spatial modeling, and you work in it with entities located in space. This is the reason why we have to distinguish between fixed entities, whose location does not change in time (say, houses) and non-fixed entities, which can relocate (say, householders). We will return to the reason why we prefer the abstract labels "fixed/non-fixed" to labels such as, say, "mobile/immobile" in the next chapters.

'Game of Life' cells are fixed. OBEUS allows two ways of getting information about fixed entities. The first is to base on an already existing GIS layer, the features of which can be considered as separate entities of a class you defined. The second is to build and spatially arrange fixed entities on the spot, when defining them. In the latter case, you will be limited to a rectangular grid in the spatial arrangement of the entities. In OBEUS, these two ways are labeled as **Fixed (New)** and **Fixed (Open)** (Figure 4.4).

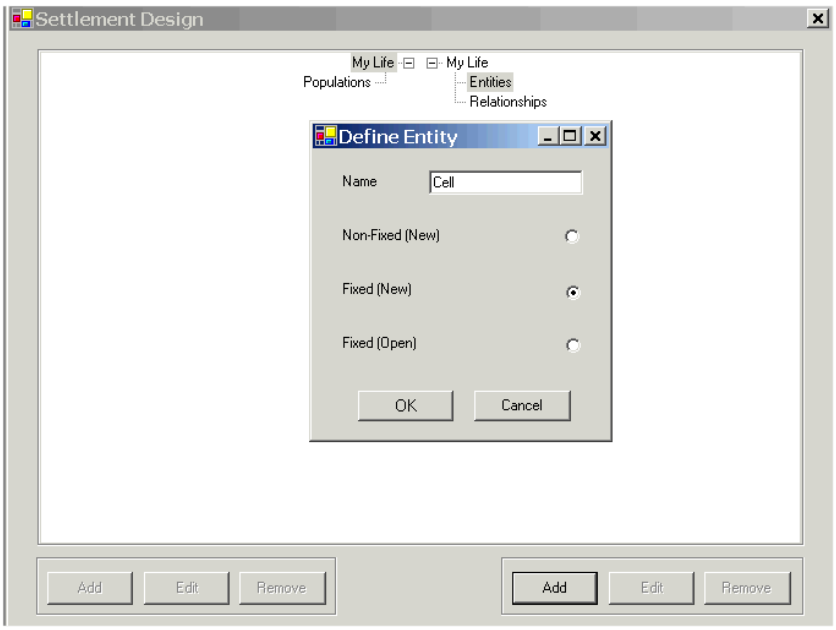


Figure 4.4: Dialog box of **Entity** definition

If an entity is defined as **Fixed (New)**, this means that we will spatially arrange the new type's entities as a rectangular grid; if defined as **Fixed (Open)**, it means we will work with the existing GIS layer.

In this starting example we will use **Fixed (New)** option and thus define new grid (Figure 4.5). When using square grids, we have to define two properties: first its dimensions, second the meaning of *neighborhood* relationships, necessary to implement the rules of the **Game of Life** as presented in Figure 1. Note that according to the definitions of the Game of Life rules (Figure 4.1), cell's neighborhood is defined as a Moore 3x3 one. **Thorus** option is explained in the next section.

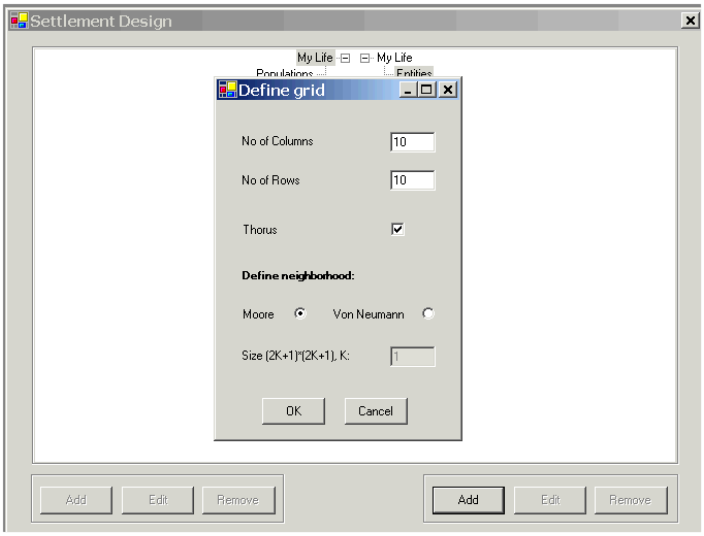


Figure 4.5: Fixed entities of the **Cell** class are arranged as the 10x10 grid using the **Fixed (New)** option.

4.5. Relationships

The definition of a cell neighborhood above is not a marginal activity related to the **Game of Life**, but is directly related to the very basic property of OBEUS – the way the system understands and works with relationships between the entities. Definition of the cell's neighborhood in the **Game of Life** is an example of defining these relationships.

The notion of the neighborhood is probably well known to you; there is also nothing impressive in the ability of OBEUS to construct a cell grid and to define the neighborhood relationships holding between the cells on the initial stage of **Game of Life** construction. Each system aimed at simulating Cellular Automata dynamics enables these options and, as a rule, displays the grid once constructed.

The difference between OBEUS and other systems lies in how these relationships are viewed and how your work with them. Standard CA models apply definition of

neighborhood each time it is necessary to retrieve data on a given cell's neighbors; they do that according to the grid coordinates of the cell. For example, if a Moore 3x3 neighborhood (as in Figure 1 of the **Game of Life**) is defined and the neighbors of the cell located in the left upper corner of the grid are considered, the system takes the cell's coordinates – (1, 1), calculates its neighbors' coordinates – (1, 2), (2, 1), and (2, 2) and, in the case of the Game of Life, retrieves the information as to whether these three neighbors are dead or alive. If we consider cell (i, j) located inside the grid, that is i and j satisfy 1 < i, j < N, the coordinates of its neighbors are (i ± 1, j ± 1) and the number of neighbors is eight (Figure 4.6).

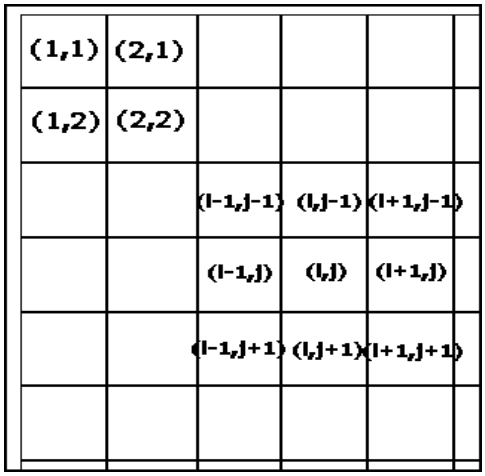


Figure 4.6: Illustration of the neighborhood and neighbors' retrieving in the cellular automata

OBEUS works with neighbors differently. When you define the Cell entity, not only is the layer of cells constructed and stored, but, in addition, a *table of neighborhood relationships* is also created and stored. In this table, each pair <cell, neighbor> defines a row. For example, in the case of a Moore 3x3 neighborhood, for left upper corner cell (1, 1) four rows are created: <(1, 1), (1, 1)> (it is convenient to consider each entity as a neighbor of itself), <(1, 1), (1, 2)>, <(1, 1), (2, 1)>, and <(1, 1), (2, 2)>. In case of internal position of a cell, the number of rows created is always nine. The first pair is <(i, j), (i, j)>, and then, if we begin from the left bottom neighbor and go counterclockwise, the pairs are: <(i, j), (i – 1, j - 1)>, <(i, j), (i, j - 1)>, <(i, j), (i + 1, j - 1)>, and so on until the last <(i, j), (i – 1, j)> is built (Figure 4.7).

ID	NID
(1,1)	(1,1)
(1,1)	(1,2)
(1,1)	(2,1)
(1,1)	(2,2)
(i,j)	(i-1,j-1)
(i,j)	(i,j-1)
(i,j)	(i+1,j)
(i,j)	(i-1,j)
(i,j)	(i,j)
(i,j)	(i+1,j)
(i,j)	(i-1,j+1)
(i,j)	(i,j+1)

Figure 4.7 – example of the relationship table rows constructed for the corner cell and a cell in an internal position

We will explain the advantages of the tables for representing neighborhood relationships in the next chapter. Just to note– the form of the relationship table

does not change when we substitute the regular grid of the Game of Life by the irregular coverage of land parcels, where the number of neighbors varies from parcel to parcel!

When the **Thorus** option is marked, a grid is built on a thorus, and the cells on the boundary of the grid become neighbors of the cells on the opposite boundary (and the number of the neighboring cells becomes eight for each cell). For example, the cell with coordinates (1, 1) becomes the (Moore) neighbor of a cell (1, n), (1, n-1), (n, 1), (n-1, 1) and (n, n) (Figure 4.8).

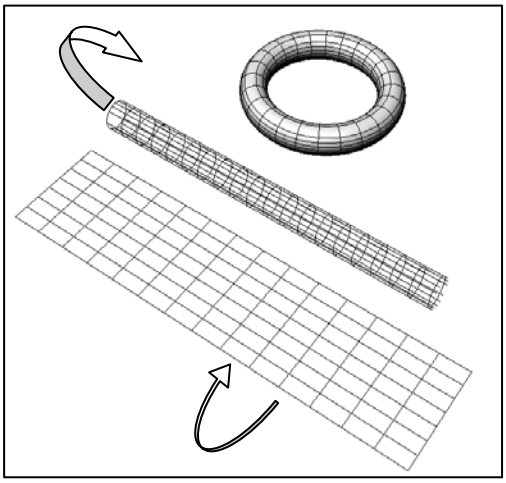


Figure 4.8: Two steps of building **Thorus** from a planar 7x14 grid

By applying the **Thorus** option, we eliminate the problems resulting from different structure of the neighborhood of the cells on the boundary of a grid and inside it; the Thorus view of the cell grid is often implemented in the CA research to avoid possible 'boundary effects'.

In OBEUS, once the dimensions of the cell grid and its neighborhood are defined, two tables of the information are constructed. The first is the cell layer, which is stored in the GIS format, and the second is the relationships table. Creation of the table of relationships is indicated in the "Relationships" part of the right branch of the tree, just below the "Entities" (Figure 4.9). We will discuss the meaning of the "FollowerID" and "LeaderID" later in this chapter.

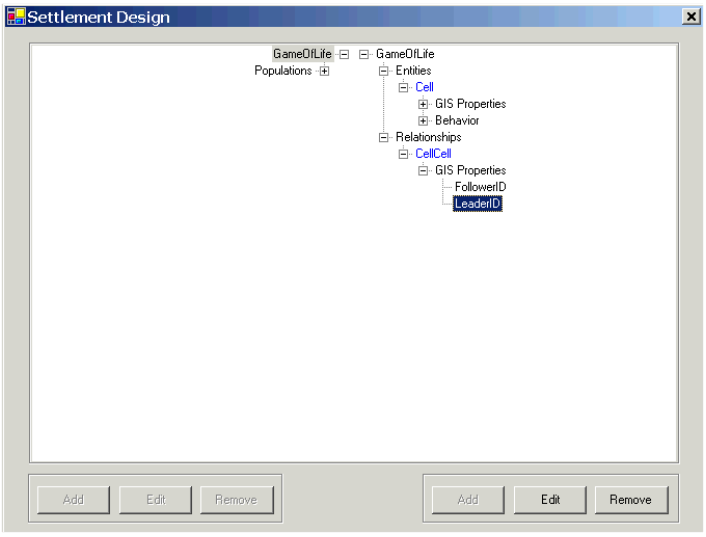


Figure 4.9: **Model Tree** after the grid is defined.

Now it is the time to push the **Save** option of the OBEUS menu. Some actions of OBEUS, including construction of the model database, are fulfilled completely only after **Save** is done. Save your work from time to time to be on the safe side, this step never causes any damage to the project.

4.6. View Map and View Table options

When the GIS layer of cells has been defined, we can view it by choosing the **Map view** option, activated by right clicking on the 'cell' entity in the model tree (Figure 4.10).

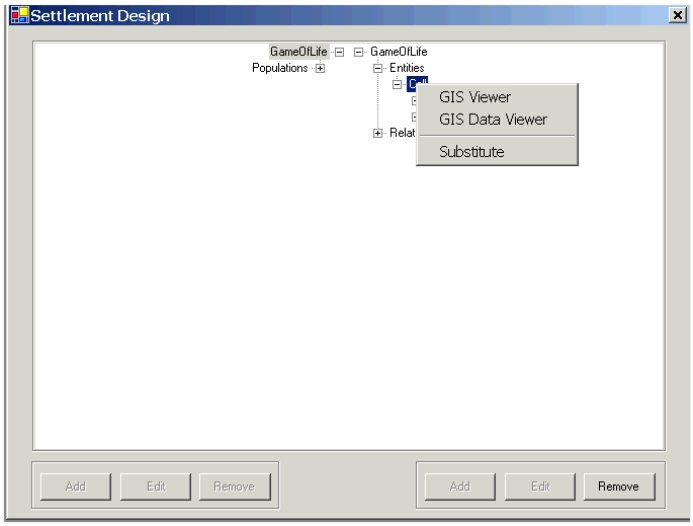


Figure 4.10: View options of the entity data

The **View map** option presents the cell grid constructed during the previous steps (Figure 4.11).

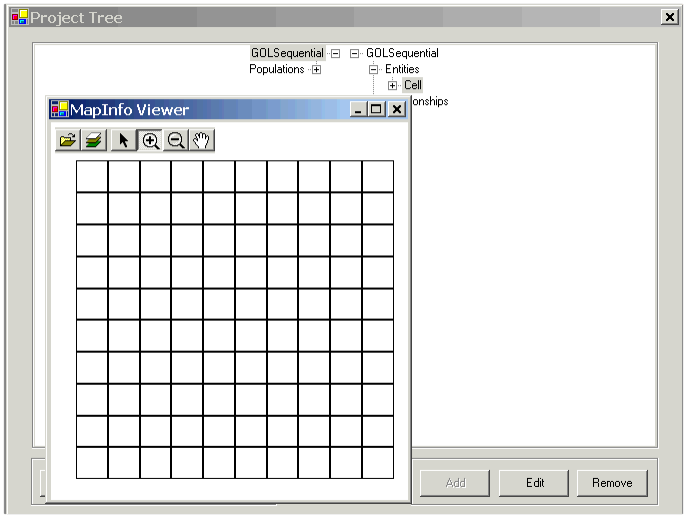


Figure 4.11: The map of a 10x10 cell grid constructed with the OBEUS

The **Table View** option opens the table of cell properties. The cells have no properties at this stage, and their only attribute, as you see in Figure 4.12 is cell identifier. The value of the identifier is unique and it is used for internal OBEUS operations (to remind - OBEUS is a database management system). You can use this identifier for your own purposes, but OBEUS will not allow you to change it.

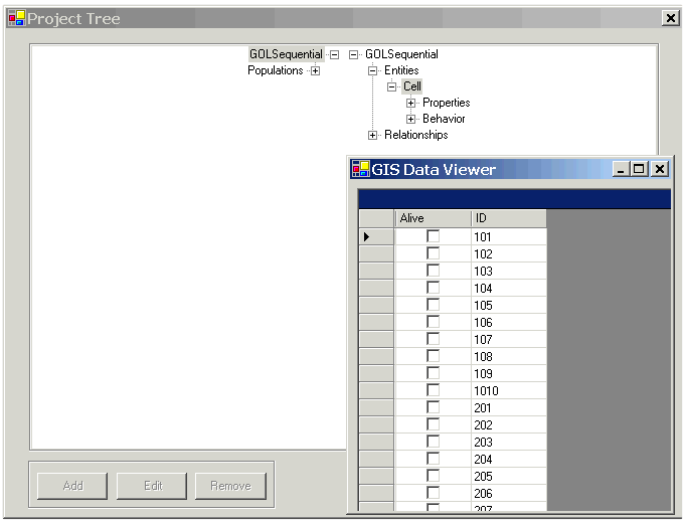


Figure 4.12: attributes of the cell entities.

As we talked above, one of the consequences of the OBEUS way of working with neighborhood relationships is that these data are stored as a table. This table can also be viewed with the right-click option when we place the mouse on the **CellCell** relationship, which, as we discussed above, was created automatically (Figure 4.13).

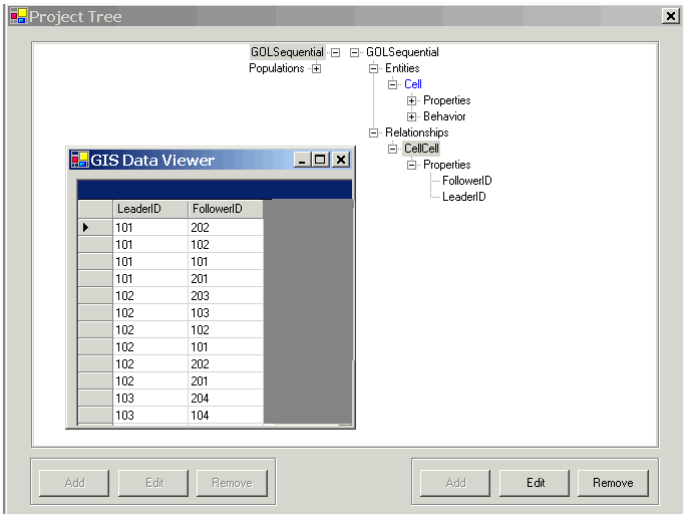


Figure 4.13: The '**Table view'** of the relationships table.

4.7. Defining entity properties

Up to here, we have been occupied in establishing the structure of the **Game of Life** *space*. Now let us define the *cell properties*. The cell we are going to work with actually has only one property with two values – it can be either 'Dead', or 'Alive'. You surely know that a property having two possible states is a Boolean property. To define the property, which in programming tradition, we will call '**Alive,**' and which will keep information on whether the cell is Dead or Alive, you have to focus on the

Cell entity at the right-hand part of the model tree and then push the **Add** button below it. The dialog box in Figure 4.14 will pop up. We then define the new property **Alive** of type **Boolean** and think of that property as being **TRUE** when the cell is 'Alive' and **FALSE** when the cell is 'Dead'.

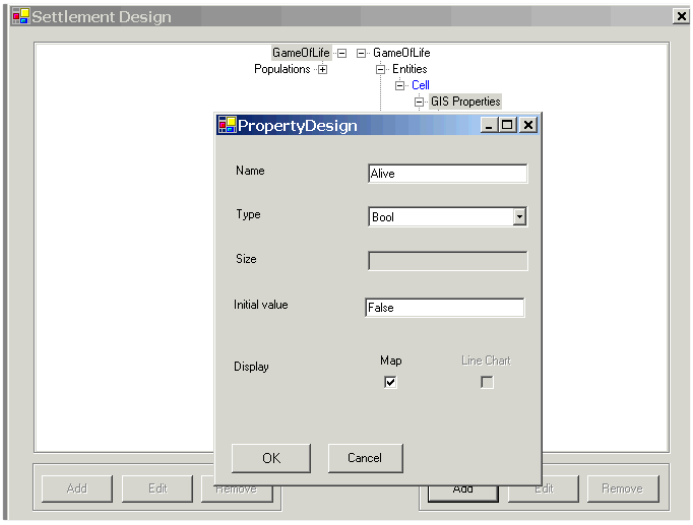


Figure 4.14: Definition of the '**Alive**' Boolean property

Let us note another important part of the property dialog box. As you can see in Figure 4.14, the bottom line of the box indicates whether the property will be displayed on the map during the model run. As it is seen in the figure, we mark '**Alive**' property for display checking "Map" checkbox.

4.8. Encoding behavioral rules

We don't need any more definitions for the cell objects of the **Game of Life** – the cells, their spatial arrangement and relations are ready to use. The only thing we still lack is the set of three rules presented in Figure 4.1, that actually define the specific Cellular Automata model we're going to investigate - the **Game of Life**. How do we formulate these rules with OBEUS? Here we come to the other basic feature characterizing use of OBEUS – you, the modeler, have **to encode these rules**. To do so, you have to use one of the **.NET languages** - C#, C++, or Visual Basic. As we will show later, OBEUS helps you do that by providing several basic methods that are **automatically** generated after you have defined your entities and relationships.

By now, you have probably enjoyed working in the interactive mode by using GUI. Putting it generally, you were able to define the structure of a model, its entities, relationships between them and the properties of the entities and relationships. The three (very algorithmically simple) rules of the **Game of Life** have to be formulated *freely*, by writing them explicitly.

The free style of rule formulation is chosen in OBEUS on purpose – the essence of the modeling style we follow lies in the rules of the entities' behavior; the modeler's achievement rests in formulating the rules in a way that no one else has done before. By aiming at universality, i.e., the ability to formulate *any* object-based model, OBEUS assumes that the modeler wants complete freedom when formulating the rules of objects' behavior. Therefore, OBEUS does not have 'predefined' sets of rules; you, yourself, create the set of rules that reflects behavior of the objects in your specific model.

OBEUS is a .NET system; that is why the user can use any .NET language for formulating rules of objects behavior – C++, VB.NET or C# currently being the most popular. We also prefer C#, which means that all our examples are programmed with this language. OBEUS itself is written in C#. Another advantage of using C# is the Borland C# compiler, which can be downloaded at no cost (see Installation section); we also utilize the Borland C# compiler in this text. If you have Microsoft .NET studio installed (see Installation section) and plan to use VB.NET of C++.NET, consider the code given below as the algorithm's description.

4.9. Behavior = Assessment rules + Automation rules

We are now ready to begin coding the (very simple) **Game of Life** rules in C#. But before doing so we should consider these rules with respect to one aspect that is important for more than OBEUS: which of the three rules we presented in section 4.1 aim at defining actual acts of **automation** and which are tools for **assessment** of 'what's going on' in order to apply the automation rule. This distinction is not completely formal, while important for establishing clear model structure.

In the **Game of Life**, this separation of automation rules into 'assessment' and 'automation' is evident: the automation - change of cell state - is determined by the number of 'Alive' neighbors, and calculation of this number is definitely an assessment rule. A cell's automation act involves either continuing with its previous Dead/Alive state or changing it. That is, in a **Game of Life** you have one assessment rule: calculation the number of 'Alive' neighbors. **Game of Life** automation rule describes the change of cell state after the assessment has been completed.

To define the rules you have to focus on the **Behavior** entry of the Cell entity, and press the **Add** button at the bottom-right (Figure 4.15). When you press this button, the Borland C# compiler is opened.

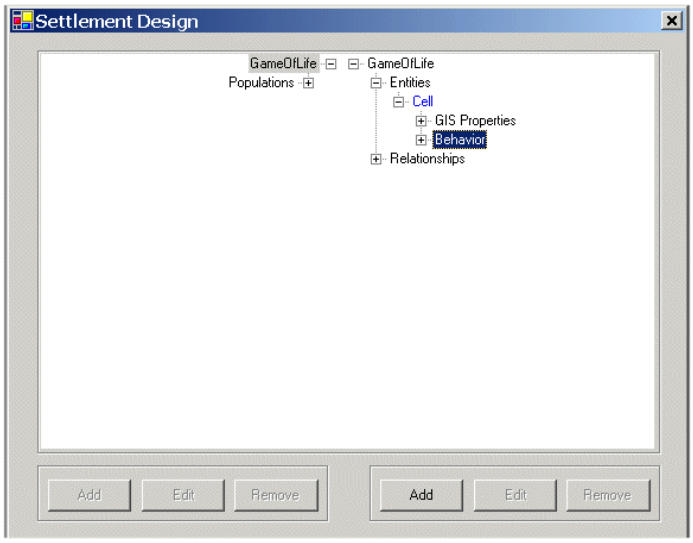


Figure 4.15: Initial situation for coding behavioral rules

We must repeat that the use of OBEUS does demand programming skills; if you have never programmed, you have to spend some time (two weeks on average) obtaining a minimal level. So, if you haven't reached this preliminary programming level by the time you read these lines, I advise put OBEUS aside and spend a couple of weeks combining the Borland C# tutorial with one of the books on C# programming listed in the Introduction section of this Help. Whatever, these two weeks will not be worn out, even if you would not be back to OBEUS any more.

4.10. Setting up a Borland C# compiler environment

Borland C#, like any other compiler, is a complicated software system containing numerous options aimed to cover the needs of very different users, ranging from novices to experts. Luckily, nowadays all programming environments are very similar. Whatever is the one you used to before, you will immediately recognize quite a lot of features you have already become used to, in the Borland C#.

To continue with this quick start, let us set up some of the C# compiler's features, just so we can proceed with the shared start situation. It will be preferable for you to have a computer with a C# environment open when reviewing the next few pages.

There are many ways you can customize the C# compiler. Below we use the 'default' desktop. To organize your desktop in this way, follow the menu options as shown in Figure 4.16.

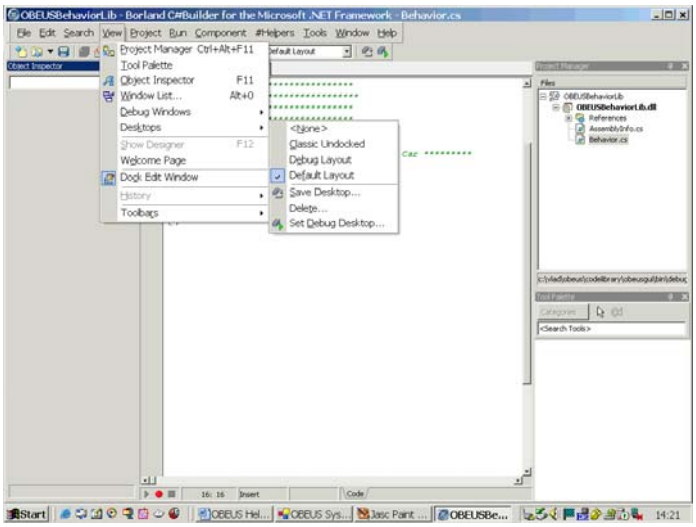


Figure 4.16: Customization of the Borland C# environment.

After you customize Borland C# compiler, click **Behavior.cs** in the bottom line of the rightmost window in Figure 4.16. Remember, '.cs' is an extension of the C# modules, and the two other .cs modules in the list of the rightmost window are internal OBEUS modules. You can see them, but don't change there anything!

4.11. Formulating assessment rules

Let us now concentrate on the main compiler window - the one in which you write the code of the Game of Life rules. As you can see in Figure 4.17, this window's content is divided into two *regions*: ASSESSMENT RULES and AUTOMATION RULES; you open either of them by clicking '+' at the right of their titles. These regions are for your convenience only, if you write assessment rule in AUTOMATION RULE region, they will work all the same. I do recommend using the regions; it makes your program more 'structured', just as OOP paradigm demands. Push **Save** in OBEUS menu before you continue, to be sure all automatically generated components are ready for use

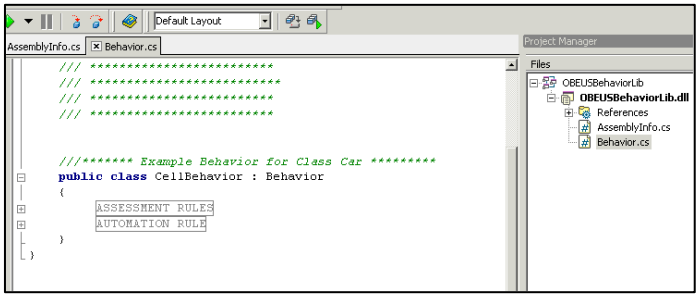


Figure 4.17: The initial state of the Behavior.cs module

Let us now formulate assessment and automation rules of the **Game of Life**.

In the case of the **Game of Life**, there is one assessment rule only, and this is its (very short and very simple) code, which you have to type in C# window of the **Behavior.cs** module:

```
private int NumberOfAliveNeighbours(Cell cell)
{
        int numberOfAliveCells = 0;
        Cell[] cells = cell.GetRelatedCells();
        if(cells != null)
    {
       foreach(Cell currentCell in cells)
       {
          if(currentCell.Alive)
          {
             numberOfAliveCells++;
          }
       }
    }
    return numberOfAliveCells;
 }
```

This is how this code looks in the **Behavior.cs** C# module (Figure 4.18). As easily seen, this code does nothing but calculate the number of cells in the 'Alive' state. Note cell.GetRelatedCells() method, which we will discuss immediately below

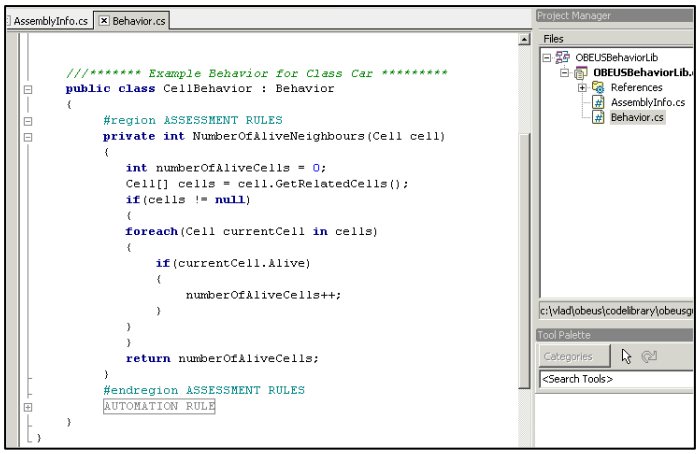


Figure 4.18: The **Game of Life** assessment rule shown in an **ASSESSMENT RULES** region of the **Behavior.cs** C# module

4.12. Methods generated by OBEUS

It is very important to note here the fundamental property of OBEUS – its ability to generate for you several basic *methods* depending on the classes of entities defined. One of these methods - **GetRelatedCells()** - is employed in the above piece of code. Let us explain this ability in more details.

As in any object-oriented programming environment, when you type '**cell.**' (cell and a point after it) in a **Behavior.cs** window, after the dot is typed the parameters and methods of the **Cell** class can be seen and selected from the pop-up window that

appears on the right click. If you've already done that for the **Game of Life** we're developing in this section, you will see, for example, the 'Alive' property as one of the properties of the **Cell** entity.

As you know, each entity and relationship defined in OBEUS with GUI turns into software class. Some of these classes, as **Cell** are defined by the modeler, while some, as **CellCell** class of relationships, which is necessary to complete the definitions made by the modeler, are defined automatically. In Object-Oriented modeling environment the properties and methods of a class are automatically available when you employ its object. Specifically, for the Game of Life, entities of the Cell class have property **Alive** we have defined via GUI, and, thus, **Alive** becomes available after dot is typed.

But this is not all! Now comes the crucial feature of OBEUS – in the box that pops up when you type '**cell.**', you will see a *method* we did not define before – **GetRelatedCells()**. It is used in the third line of an assessment rule code above, and its meaning is intuitively evident – to obtain all the cells related to a given cell for further analysis, calculation of the fraction of the 'Alive' cells among them, for example. This is what OBEUS gives to you – the set of built-in methods which are 'self-evident' and which you usually code yourself in every model you develop! As you will see when working with OBEUS, automatically constructed methods of this kind enormously reduce the amount of code you have to write when formulating your model. In addition, the methods generated by OBEUS are optimized, and so the time **cell.GetRelatedCells()** takes for execution is much less compared to the double loop you might use for retrieving the Moore neighbors of a given cell. The list of the automatically generated methods is given in Chapter 9.

To catch the bugs of the code immediately after you do them, I recommend compiling the assessment rule just after it is formulated in C#. Push **F5** (Build) of the **Borland C#** environment for that, and fix errors if found. After successful compilation the assessment method **NumberOfAliveNeighbors()** becomes available for use for further development, as one more method of the **Cell** class.

4.13. Formulating automation rule

After we have coded (the only one) assessment rule of the Game of Life, we have to code its automation rule, which simply reflects the three substitution options between the cell's Alive and Dead states. Now pay attention as to how the assessment rule we formulated above - **NumberOfAliveNeighbors()** - is employed

when formulating the automation rule:

```
public void A_MainStrategy(Entity entity)
{
    Cell  cell = (Cell)entity;
    int numberOfAliveCells = NumberOfAliveNeighbours(cell);
    if (numberOfAliveCells <= 1)
    {
        cell.Alive = false;
    }
    else if (numberOfAliveCells == 2)
    {
    }
    else if (numberOfAliveCells == 3)
    {
        cell.Alive = true;
    }
    else
    {
        cell.Alive = false;
    }
}
```

Figure 4.19 presents this piece of code written in the **Behavior.cs** module, within the **AUTOMATION RULE** region:

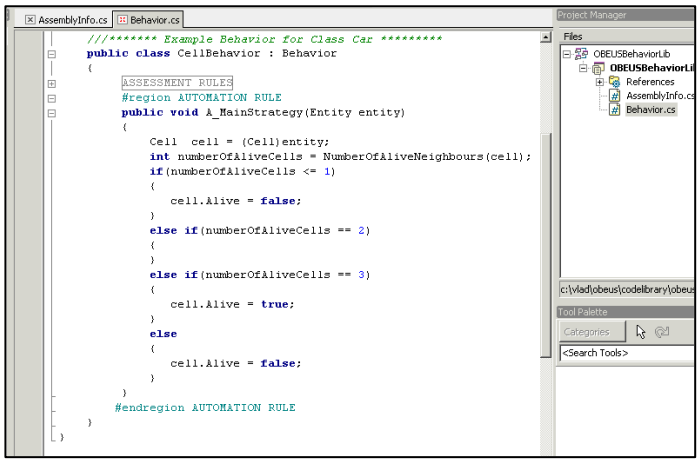


Figure 4.19: The **Game of Life** automation rule shown in an **AUTOMATION RULES** region of the **Behavior.cs** C# module

Notice that unlike the arbitrarily number of assessment rules you can formulated in OBEUS model, the *automation rule in OBEUS is unique*. It also always has a unique name - **A_MainStrategy()**. If you want to employ another automation rule, you have to rename it into **A_MainStrategy()**.

Don't forget Build (push **F5**) the **Behavior.cs** after you type the behavioral rule.

Figure 4.20 presents the final state of the model tree of the **Game of Life**, with contains all the elements defined above.

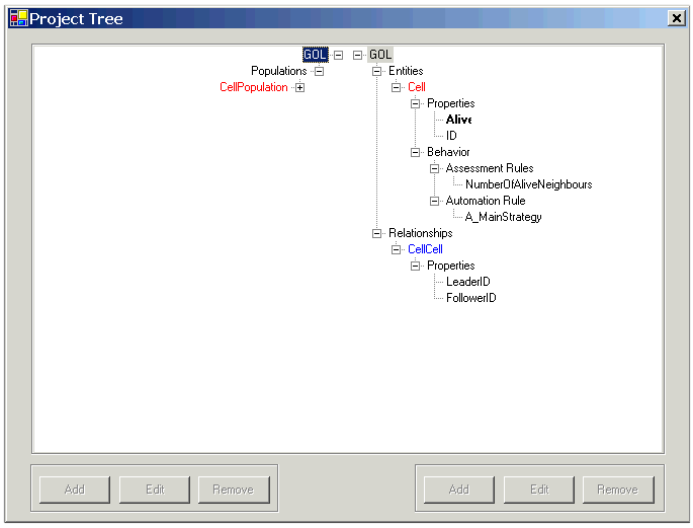


Figure 4.20: The **model tree** of the **Game of Life**, after model entities, relationships, and assessment and automation rules are defined

Let us note that the way we coded assessment and automation rules of the **Game of Life** above is only one of several possible; the same rules can be formulated according to your programming style, while for the **Game of Life** it's hard to imagine essential variations. But note that in order to compare your formalization of the **Game of Life** with someone else's formalization, you have to compare only these two pieces of code – assessment rule and automation rule. Every modeler knows that externally identical models might apply similar but not identical rules, to say nothing about the situation when the model works, but does something different from what **you**, the creator, wanted of it! One of the goals of OBEUS is to remedy just that – to make the models comparable and transferable at the level of objects and behavioral rules only.

4.14. Debugging behavioral rules

'There is no software without bugs' – if you've written code even once, you will readily subscribe to this claim. As we already mentioned twice, at any stage of writing the code of the assessment or behavioral rules of the **Game of Life** or any other model in OBEUS, you may try to 'build' this module by clicking the Build – **F5** - option in the **Borland C#** environment and fix the bugs according to the error information. OBEUS accepts all syntax and runtime debugging possibilities – break points, variables watch, etc. As we've repeatedly said, to use all them you have to have some programming experience. If you feel uncomfortable writing code, I remind my suggestion to leave OBEUS for a while, spend a couple of days with the **Borland C#** tutorial and help, and build some programs that are not related to the

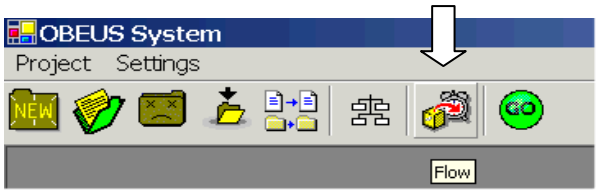
model you're going to develop with OBEUS, just to get used to **C#.NET** programming.

After the assessment and automation rules are coded and the compilation bugs fixed, make a final save of your **Game of Life** code (**Borland C#** will remind if you forgot) and close the **Borland C#** windows. I also suggest pushing the **Save** option of the OBEUS environment just to confirm that the final version of the project is indeed saved.

4.15. Time flow of events in Game of Life

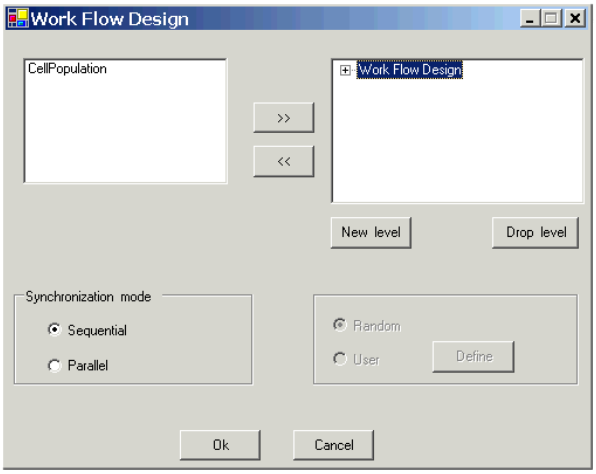
Can we run the **Game of Life** now? Not yet, although you've done almost everything you need to build a model of the game. What OBEUS still lacks is a definition of how the events in your model are carried out in time. In other words, you have to establish how time will be treated in the model. To initiate this component, click the 'Flow' option on the OBEUS menu (Figure 4.21).

Figure 4.21: Time flow tool of the OBEUS



The **Flow Design** dialog box is then opened as shown in Figure 4.22.

Figure 4.22: **Flow Design** dialog in the **Game of Life** case



Putting it briefly, the time **flow** defines two aspects of a model, both related to the modeler's understanding of how time is processed in the model. The first aspect – the order of updating entities of different classes is irrelevant for the Game of Life,

because there is only one class of entities. Only the second aspect is thus relevant, namely, the synchronous or asynchronous view of updating of entities.

The two components of time management are established in different parts of the dialog, and the order between entities of different kinds must be always defined first.

To set the (only one in case of **Game of Life**) levels of updating and to mark for OBEUS that these are **Cells** that should be updated you click the **New Level** option, and get the result as shown in Figure 4.23:

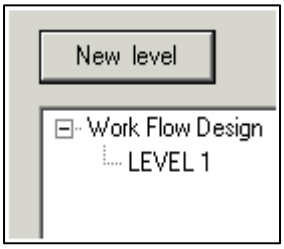


Figure 4.23: Definition of a **new level** in the **time flow** design

Then you have to focus on the only element of the left box in the Figure 4.22 – **CellPopulation** and push the button with '**>>**' on it, to get the result shown in Figure 4.24. To note, we have not defined what **Population** means yet, but the intuitive understanding is sufficient at this stage. We will talk about population later, when discussing the Schelling model of residential dynamics.

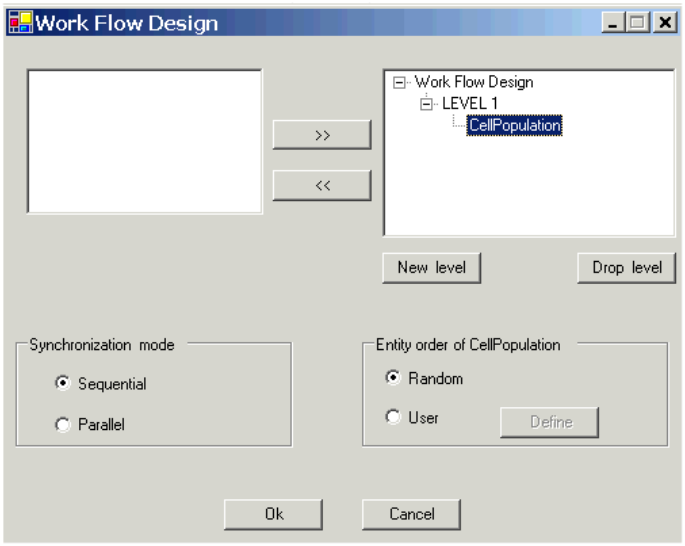


Figure 4.24: The time flow of the Game of Life

We are ready now to establish the synchronization mode of the Game of Life.

4.16. Synchronization mode

Generally, there are two basic modes of updating in discrete models – *parallel* and *sequential*. In the parallel mode, the system is 'frozen' at the beginning of the time step, and when certain behavior rule accounts for the entities' states, these states are taken from the frozen situation. The changes that occur during the current time

step are stored until its end, when all the cells are simultaneously updated.

In the sequential mode, everything that happens with the certain entity immediately becomes available to the other entities in the system. The modeler thus has to define the order in which the entities are updated. Usually, the order is random or determined by some characteristic of the entities.

As you can see in Figure 24, the sequential mode with random order of updating is now selected for the Game of Life. That means that at the beginning of the iteration, the cells will be randomly ordered, and once started, the first cell in this order will be taken, the rules of the game will be applied to it, then the second, and so forth. Beginning with the second cell, the information this cell retrieves from the system will be different from that existing at the beginning of the iteration because in the sequential mode all the changes will be immediately available. As we will see soon, the mode of cell updating is very important for the Game of Life.

Close the flow dialog; the only action left is to run the model, while you can still change initial conditions before that.

4.17. Before running the model – Change initial conditions

You have defined the cell entities, relationships between neighbors and the rules of the Game of Life. The next stage is to run the simulation. Before running **Game of Life** model you still might change model's *initial conditions*.

Establishing initial conditions is an important and necessary step of every simulation. The basic way to do that in OBEUS is to enter section called **Initialization** in the left branch of the model tree and to write in C# the rules that initialize model entities and relationships in the beginning of every run. For example, we might want to begin with the chessboard assignment of 'Alive' and 'Dead' states to the cells. We will return to the **Initialization** section in full in the next chapter, when talking about Schelling model. Right now we only want to make a few cells 'Alive' and then watch the system's dynamics. Therefore, we begin with a grid where all the cells are in a default - "Dead" - state set up by the OBEUS when the 10x10 grid of cells was defined, just because default value of the Boolean property is **FALSE**. Yet, we cannot begin with all 'Dead' cells – according to the rules of the **Game of Life** they will

remain so forever.

To change the state of some cells from ‘Dead’ to ‘Alive’ open the map of cells via the model tree (Figure 4.11) and push an **arrow** tool on the top of the map. Click cells which you want to change with the **Control** key pressed and choose **Change selected data** option from the **Right-Click** mouse menu (Figure 27a). The table of the attributes of the selected cells is shown up, and you can change the ‘Alive’ attribute from **False** to **True** (Figure 4.25b).

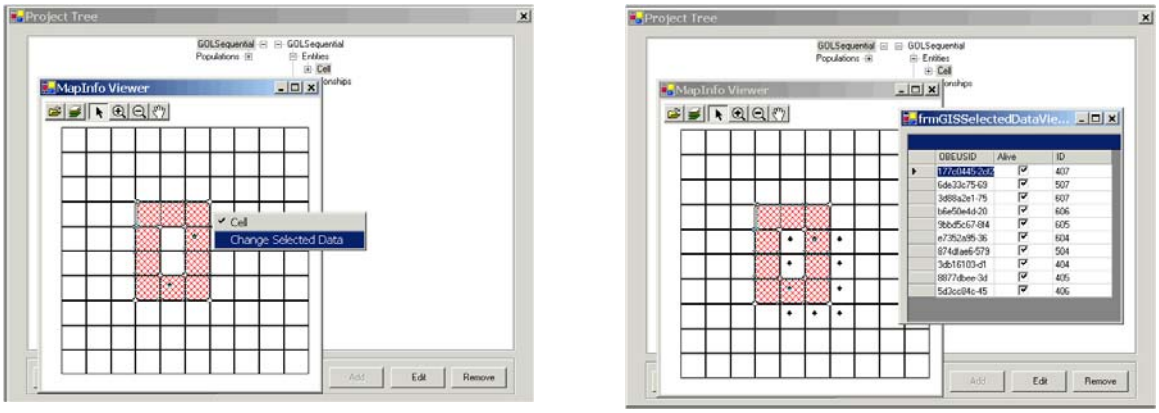


Figure 4.25: Attributes of the selected cells (a) can be changed (b) and by that new initial conditions are established. The result is shown in Figure 4.26.

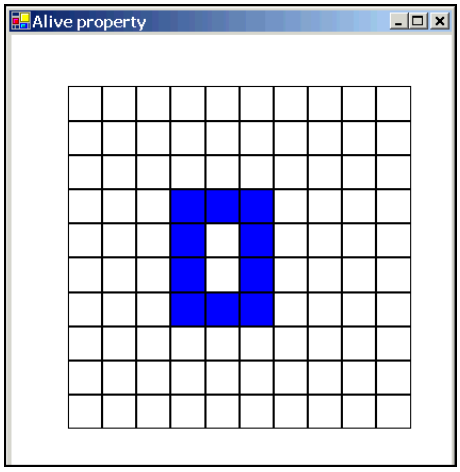


Figure 4.28: The result of changing Alive attribute of ten selected cells from **FALSE** to **TRUE**

To finalize the project we have to define model’s output – how frequently we want to store the wealth of information the model produces. At this stage we simply use OBEUS defaults for we have already worked a great deal in the quick start with so simple a model.

4.18. Run the simulation

Glick **GO** icon (Figure 4.26) the windows of the map and graph windows that you demanded when defining the model (Figure 4.27)

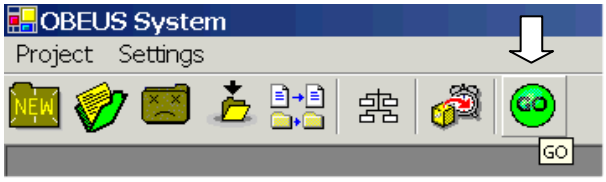


Figure 4.25: **GO** option

Initial state of OBEUS before the model is run includes two new elements: Runtime toolbar and the message window (Black window below the map window) (Figure 28)

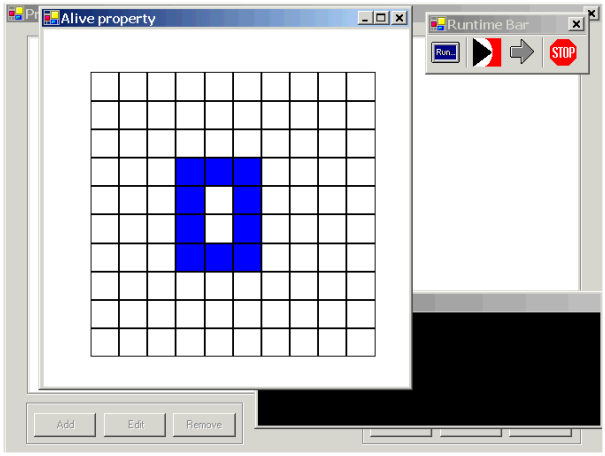
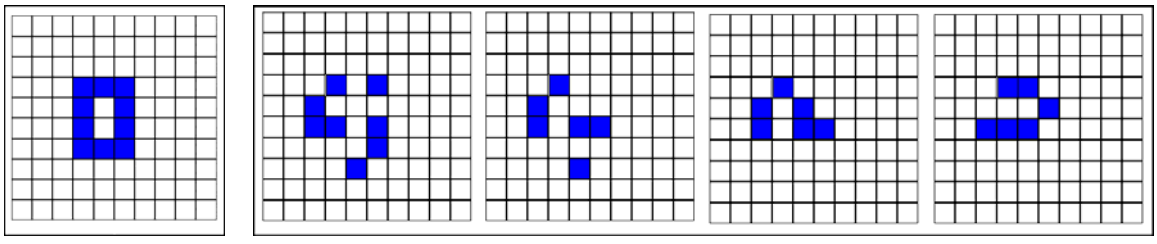


Figure 4.26: Initialization of the Game of Life

We will not talk about the latter in this chapter. Regarding the former, **Run** option (first from the left) runs the model, **Pause** (second from left) makes possible to pause execution, **Step** (third from left, looks as an arrow), and **Stop** (the last one) cancels the run and closes all model window. To run the model again after pressing **Stop** you have to begin with **GO**. Figure 4.29 presents several first map screenshots of the Game of Life model:



t = 0 t = 1 t = 2 t = 3 t = 4

Figure 4.29: The **Game of Life** model with **asynchronous** time flow. Screenshots of the Initial conditions (t = 0) and at t = 1, 2, 3, and 4

Hmm… where are the gliders and the other nice emerging patterns we're familiar with? Did we choose the wrong initial conditions or wrongly formulate the model rules? Not at all! The lack of the patterns is not an error. It is the consequence of the sequential mode of updating we chose when setting up the model flow.

4.19. From asynchronous to synchronous mode of updating

Our goal at this stage is to demonstrate the use of OBEUS. We have therefore delayed discussion of the essential differences between the **Game of Life** with synchronous as opposed to asynchronous application of the behavioral rules. At this stage we will simply compare the outcomes.

Important note: you might want to keep both versions of the model – with the parallel and sequential time flow, and compare their outputs. To do that, store the current (sequential) version of the Game of Life under different name in the model database before you change the time-flow mode (Figure 30). See more information on versioning in the next chapter.

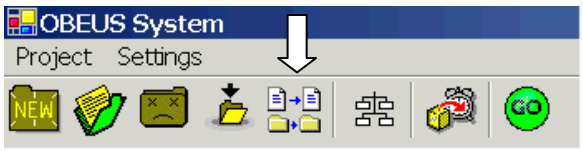


Figure 30: Save as button

Until now, we dealt with the model with the default - sequential (asynchronous) – time flow mode. The order of cell updating in the model was also default - random. To change this order to a parallel (synchronous) one, we have to call on the model's time-flow dialog again. To do that, we have to stop the model, press **Flow** button, and change the **Sequential** option to **Parallel** (Figure 4.31). Note – there are ways of establishing the order of cell updating in sequential mode (Figure 31), but will delay this discussion to the next chapter.

The change of the time flow demands minor changes in the C# text of the model and then you have to recompile it. In the case of the Game of Life there is one line one to change – go to **Cell Assessment Rules**, where you have the C# implementation of the only assessment rule of the Game of Life (Figure 4.18) press **Edit** and change the line

Cell[] cells = cell.GetRelatedCells(); into

Cell[] cells = cell.GetRelatedCellsRO();

(Simply add "RO" at the end of the name of the method **GetRelatedCells**)

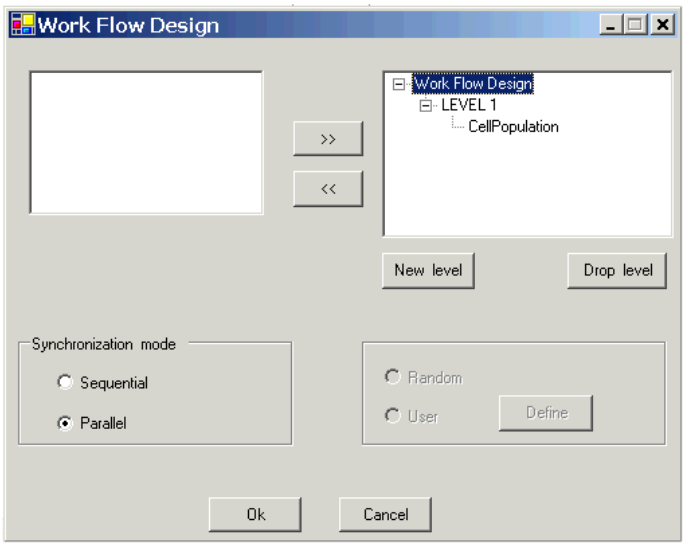
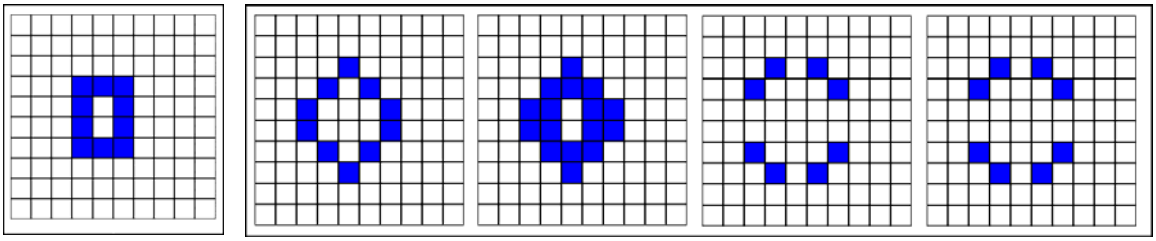


Figure 4.32: Time flow dialog box of OBEUS with **Parallel** option chosen

Repeat **Go** and **Run** actions, and the well-known pattern does appear (Figure 4.31)!



t = 0 t = 1 t = 2 t = 3 t = 4

Figure 4.32: The **Game of Life** model with **synchronous** time flow. Screenshots of the Initial conditions (t = 0) and at t = 1, 2, 3, and 4

You will not forget now that the famous patterns of the Game of Life are the result of the parallel updating. Difference in results of the same model when parallel and sequential updating is employed is always a serious issue you should think about. OBEUS makes it possible to compare the outcomes for these two time-flow modes.

4.20. Substitute layer of cells by another layer

Let's exploit the **Game of Life** to demonstrate one more basic and useful feature of OBEUS. As you will remember, when constructing the Game of Life we built a 10x10 field of cells, chosen the Moore forms of the neighborhood, and automatically obtained the table of relationships, where each row represented a pair – <ID of a cell, ID of a neighboring cell>. But why should we use a 10x10 grid of cells? What would happen if we wanted to work with a larger grid of, say 50x50 cells?

One of the basic features of OBEUS is its ability to make this transfer easily. Just recall that the 10x10 grid of cell objects we worked with up to now was stored as a GIS layer immediately after we defined it. As you know very well from your work with GIS, layers can substitute for each other! This is exactly the way OBEUS proceeds.

Important note: you might want to keep both versions of the model – with the initial and new GIS layers, and compare their outputs. To do that, store the current version of the Game of Life under different name in the model database before you substitute the layers, as it is shown in (Figure 30).

If you have a 50x50 grid of cells stored as a layer you can substitute the 10x10 gird by this stored one. The substitute option is located in the right-click menu of the model tree (Figure 4.33). Focus on the entity you want to substitute before activating this option:

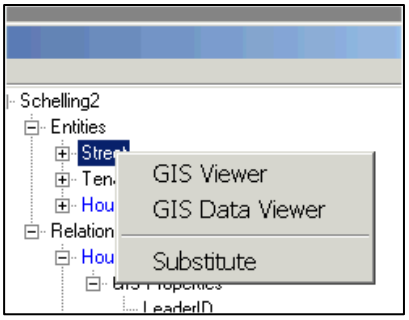


Figure 4.33: Substitute option of OBEUS

To substitute one presentation of the class objects by another, all the entities should have the same attributes. In the case of the **Game of Life**, the only attribute of cells is Boolean **Alive**; the new layer should have the same attribute then. Click the **Cell** entity and check the attributes of the Cell class. You can **Add**, **Edit** and **Delete** each of them; use these option to guarantee the **Alive** attribute exists. While don't think OBEUS relies on you at this stage, if necessary attributes do not exist in a new layer, OBEUS will give a message that lists lacking attributes and cancel the process (Figure 34).

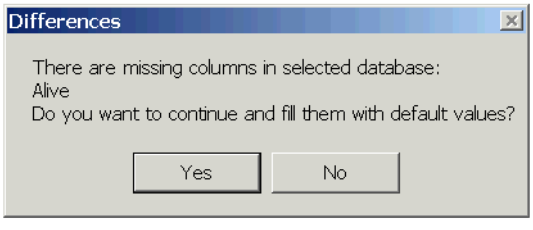


Figure 34: OBEUS message in case of difference in the attributes between the current and the new one GIS layers

There is no need to delete the unnecessary attributes if existing in a substituted layer. The only negative consequence will be the storage of the garbage data these attributes contain.

Less evident, but not less important condition that should be also tested before substitution – does the relationships table, expressing neighborhood relationships between cells of the 50x50 layer, already exists?

Remember that a relationships table was automatically created when we chosen the 3x3 Moore neighborhood in a **Fixed (New)** dialog. The substitution action demands that we provide a new table of neighborhood relationships together with the layer itself (Figure 4.34). If it is missing, the substitute action is cancelled by OBEUS, just as when you don't want to proceed in case of different attributes in Figure 34.

It is worth noting that OBEUS does not take responsibility for the correspondence between the layer of cells and the table of neighborhood relationships, just because the relationships between cells can be defined in numerous ways. If, for example, you click the relationships table that was built on the base of Moore definition of the neighborhood, for the initial 10x10 grid and not for the new 50x50 layer, OBEUS will accept that action. However, in this case you will inevitably get a runtime error and an error message later, at the **GO** and **Run** stages.

The simplest way to ensure proper substitution is to create the 50x50 layer with the **Alive** attribute within OBEUS itself. Remind that in this case, the relationships table will be created automatically by OBEUS.

To do that, open a new project, say, **GOL50**, and follow the steps of the Game of Life definition until you reach the dialog presented in Figure 5; then type 50 instead of 10 as the numbers of rows and columns. The layers and tables will be stored in the directory /Settlements/GOL50/. Save the new project the project and close it. Use the layers of the GOL50 project for substitution and then, if you don't want to proceed with it, just open it again and delete. Now you can use **Substitute** option with the layers in which correctness you are sure.

Use **GIS viewer** option to confirm the substitution, the map should look as it is shown in Figure 4.35

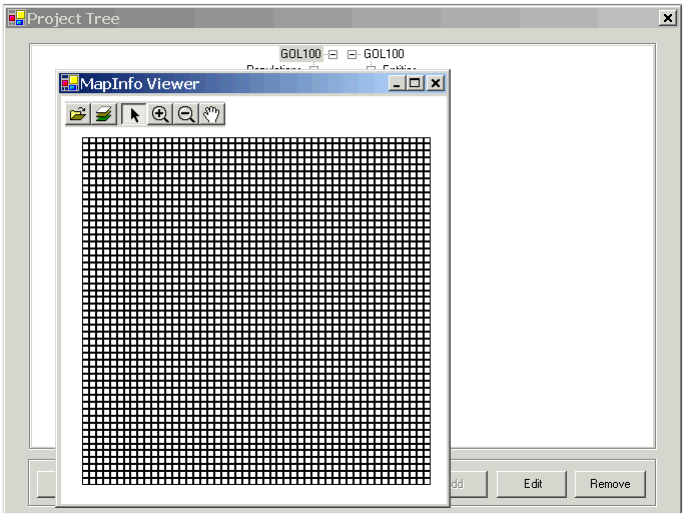


Figure 4.35: **Game of Life** map in case of 50x50 grid.

The advantage of the substitute option is evident – if you continue with the GOL50 project you will have to code all the rules again; in case of the substitution you use the old ones.

4.21. What we have learned with the Game of Life

We can now list the main OBEUS actions we studied with the help of the Game of Life:

1. How to open a new project
2. How to define a new class of fixed entities
3. How to define entity's properties
4. How to build a presentation of entities as a rectangular grid
5. What is neighborhood relationship table and how it is created automatically
6. How to view the map and the attribute table of the class of entities
7. What are assessment and automation rules and how to formulate and debug them with the Borland C# compiler
8. How to define the synchronization mode – parallel or sequential
9. How to Run/Pause/Stop the model
10. How to update initial conditions before running the model
11. How to substitute the current presentation of the class objects by another one

There are several more basic actions in OBEUS, but the Game of Life is too simple a model to demonstrate them. We now turn to another, more complex but also simple example – Schelling's model of residential dynamics in the city — to complete our demonstration.

5. Quick start with Schelling model of residential segregation

Despite the variety of the patterns produced, the rules of **Game of Life** are very simple, and its entities are all of one and the same type – immobile cells. The **Game of Life** does not employ migrating objects, which, potentially, might have more complex behavior than fixed cells. Our second example of the OBEUS modeling does include migrating entities; it implements for this purpose the model of residential dynamics proposed by Thomas Schelling at the end of 1960s.

5.1. The Schelling model – an introduction

Schelling model deals with (very abstract) householders, who can migrate in reaction to their neighbors. Neighbors can be of two types, and householders prefer to settle within the friendly neighborhood; they migrate when the neighborhood is not friendly enough. Formally, **Schelling model** works with *two types of entities*. Entities of the first type represent *houses*, which host entities of the second type – *tenant agents*, and the only reason for tenant's migration is dissatisfaction from the neighbors – other tenants of the same house or tenants of the neighboring houses.

Schelling model has many versions, while we focus on the basic one, proposed in 1971 (Schelling 1971). In basic version of the model, each house can host one tenant only. Tenants in the basic Schelling model are of two kinds: Black (**B**) and White (**W**). An agent reacts (by migration) to neighbors of their own type – "friends". Namely, if the fraction of friends in the neighborhood is below the predefined threshold $\mathbf{F_{th}}$, then the tenant tries to migrate to a house, where the fraction of friends in the neighborhood is above $\mathbf{F_{th}}$. Conceptually similar model, in which tenants react not only to friends, but also to strangers within the neighborhood, was considered by Sakoda (Sakoda 1971). Different from the **Game of Life**, we do not need such graphic presentation of the **Schelling model**, just because instead of four-fold rule in the **Game of Life**, the dynamics of the **Schelling model** is defined by one parameter only - fraction of friends $\mathbf{F_{th}}$

Schelling model has been investigated in many papers and its typical outcome is a segregated residential pattern, where **B**- and **W**- tenants aggregate into Black and White areas [REFS].

Below, demonstrating development of the **Schelling model** within OBEUS, we reduce the sections that we've already passed with the **Game of Life**, while figure out the details that are important for the models where entities of several classes are considered and some of them are able to relocate.

Let us begin just as in the **Game of Life** – to open new project and define 10x10-grid with Moore neighborhood for spatial arrangement of the house. In the **Schelling model** we will call them **houses**, and not **cells**, and we don't need any property of the **house** to be defined, for the **houses** are neutral containers of the **tenants**.

5.2. Using existing GIS layers to define new classes of entities

Building new grid and the corresponding table of the neighborhood relationships anew it not necessarily – we could choose **Fixed (Open)** option when defining the 'cell' entity, and then use the layer of cells and the corresponding table of relationships we've built for the Game of Life (Figure 5.1).

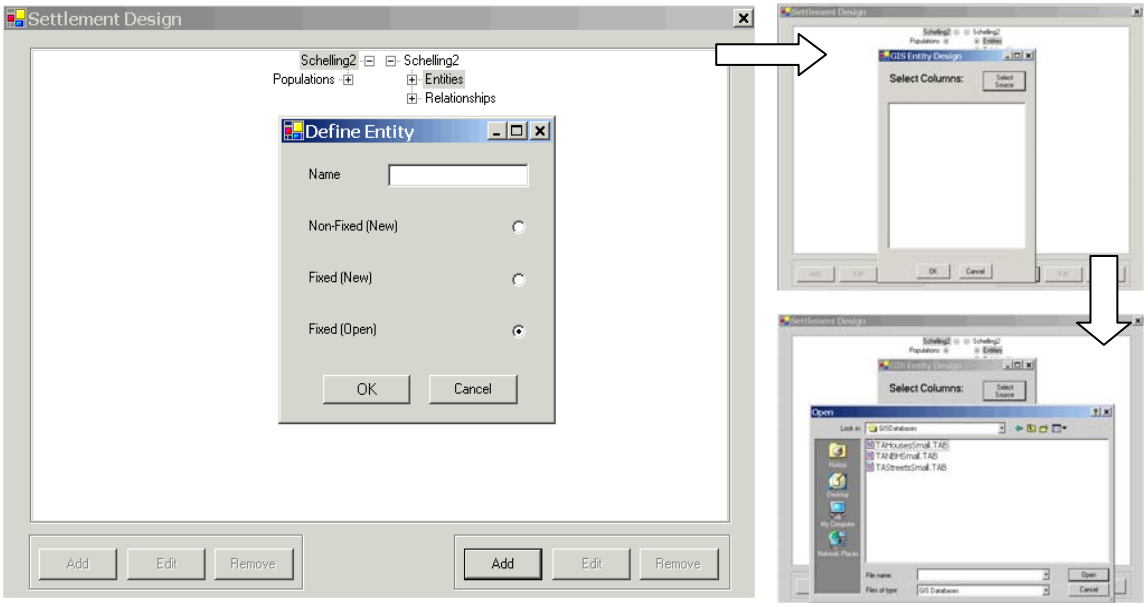


Figure 5.1: Stages of the opening existing grid as a layer of houses of the **Schelling model**

The **cell** entities of the grid were defined for the **Game of Life** and they have **Alive** property, which we don't need for the **houses**. Nothing will happen if we keep this property, but it is better to remove it, just to get rid of the garbage. To do that, focus cursor on **Alive** property and push **Remove** - the rightmost button at the bottom of the **Model Tree** dialog. You will be warned before remove will be executed (Figure 5.2).

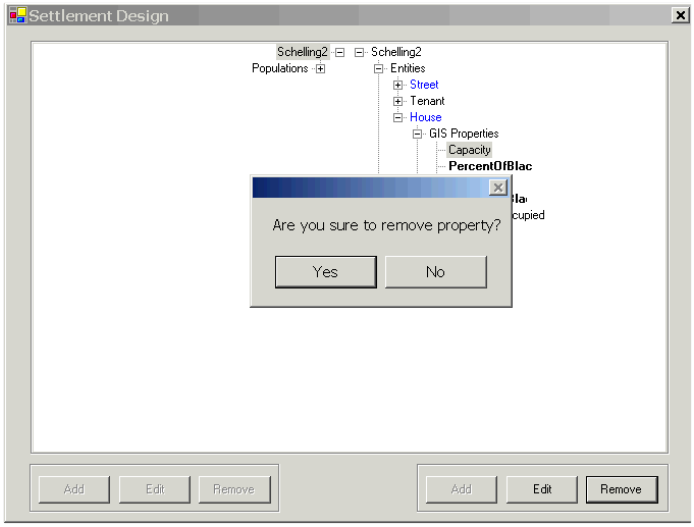


Figure 5.2: Warning when executing **Remove** operation

We reach real differences between **Game of Life** and **Schelling model** when defining mobile **tenants**.

5.3. Defining and locating mobile entities

We begin definition of **Tenant** class of entities in the same way as we did with **Houses** - focus on **Entities**, and push **Add** at the bottom of the **model tree** dialog. To define **Tenants** of the **Schelling model** we have to choose the **Non-Fixed** option (Figure 5.3).

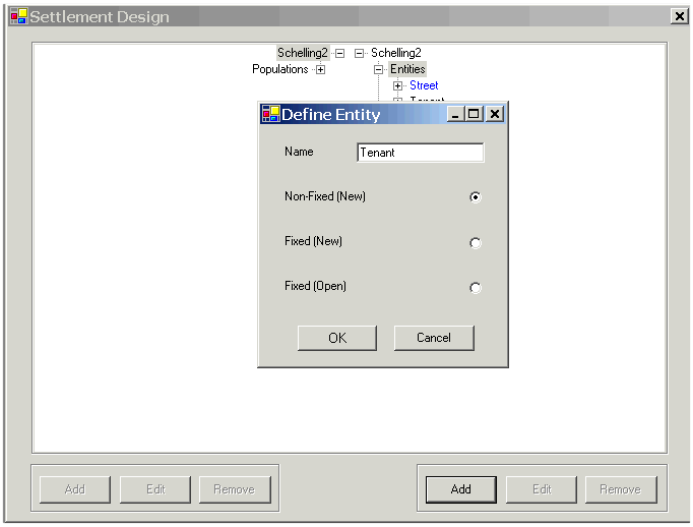


Figure 5.3: Choice of **Non-Fixed** option for defining **Tenants** of the **Schelling model**

Different for the fixed objects, OBEUS does not build GIS layer for the objects of the non-fixed class. Indeed, **tenants** do not have 'their own' location at some (x, y) point of geographic space. The essence of being **tenant** is to be located in a **house**

and that is why tenant's location cannot be absolute – it should be defined via houses. The layer of houses already exists, and, thus, for each tenant we have to mark the house it lives in – in other words, to locate tenants we have to *define relationship between them and the houses*.

We already used relationships in the **Game of Life** - neighborhood relationships between **cells**, which were automatically built by the OBEUS. In **Schelling model** we have two classes of objects, and besides neighborhood relationships between houses we have to define the relationship which determines *the house the tenant lives in*. Let us call these relationships **House-House** and **Tenant-House**.

5.4. Indirect geo-referencing

Tenant-House relationship presents basic feature of OBEUS – indirect geo-referencing. As we mentioned above, the tenant in Schelling model does not exist in an 'open space'. We will define the rules of the tenants 'migration behavior' later on, but whatever they will be, tenants in Schelling model always either move to the other house or remains in one they occupy now. That is why a tenant can be unambiguously located by pointing to the house, that is, by means of the occurrence of the **Tenant-House** relationship.

Let us note that the study of different formulations of migration behavior is the main aspect of the **Schelling model** investigation. For example, assuming several free locations with a fraction of friendly neighbors above $\mathbf{F_{Th}}$ are available, which one is occupied by a tenant? There are several ways to formulate the choice rule – one can investigate what will be the residential pattern if tenant chooses the house with maximal fraction of friends, or when the house is randomly selected from the available ones, or when the house which is closest to the current location is selected, and so on. Whatever option is chosen, the result of the migration will be occupation of the house and, thus, the old occurrence of the **Tenant-House** relationship should be destroyed and the new one constructed.

Different from **House-House** relationship, which was built automatically, the **Tenant-House** one should be explicitly defined in OBEUS. To do that we have to focus on **Relationship** entry in the left side of the **Model tree** and click **Add** button in the bottom row. The dialog-box that pops up allows defining the entities to relate,

and which of them will be a *leader* and the *follower* (Figure 5.4).

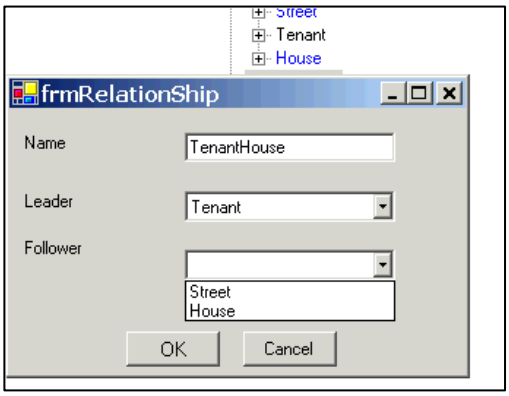


Figure 5.4: Definition of the **Tenant-House** relationship via OBEUS GUI, with **Tenants** selected as a leader

The *Leader* entity is an active participant of the relationship. Leader can destroy its relationship with the Follower and create relationship with the other entity of the follower class. In case of **Tenants** and **Houses**, **Tenants** are evident leaders – they decide weather to leave the **house** when the fraction of friends in the neighborhood is insufficient, and which of the free **houses** will be chosen for next occupation. Entities of the **House** class are evidently followers.

The case of **Tenants** and **Houses** is typical – non-fixed (mobile, migrating) entities are usually leaders, while fixed entities are usually followers in relationships. OBEUS imposes this view as a general limitation, and, as you may see in Figure 5.4, **Tenant** entity does not appear in the 'follower' control at all. We will talk more about the Leader-Follower pattern of relationship in the next chapter, which presents general scheme of OBEUS.

The **Tenant-House** relationship has the same presentation as the **House-House** one (Figure 5.5), but you cannot see it in a current version of OBEUS.

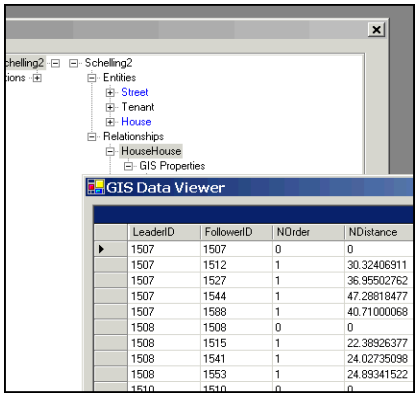


Figure 5.5: **House-House** relationship table for non-zero number of tenants in the **Schelling model**

When defined, the **Tenant-House** relationship table is empty; new occurrences of

the relationship are constructed when **Tenants** enter the city (say, immigrate during the model run).

5.5. General view of relationships between entities of the Schelling model

If you tend to formal completeness, you will immediately ask about two more relationships that can be considered in the **Schelling model**: **House-Tenant** and **Tenant-Tenant**. You are right - this is more than a formal question. **House-Tenant** relationship might help in determining tenants living in a given house, while **Tenant-Tenant** relationship is important in determining tenant's neighbors. Schelling model is, actually, based on the Tenant-Tenant relationships: a tenant agent reacts to the fraction of friends, i.e. tenants, of the same type. Let us consider all four possible relationships between **Tenants** and **Houses** - two we defined above and two remaining ones in order to understand the general logics of the OBEUS regarding relationships.

House-House is a neighboring relationship. It is built by OBEUS when a grid of the house is built and then stored, just as in a **Game of Life**. Let us note that in case of relationship between the houses, just as in the general case of two fixed entities, the relationship does not change it time. Thus, the leader and the follower of the relationship between the fixed entities can be set arbitrarily.

Tenant-House is used for tenants' location and Schelling model cannot be implemented in OBEUS until is this relationship is defined. As we discussed above, **Tenants** are leaders and **Houses** are followers in this relationship.

Let us turn now to two relationships that we didn't define yet - **House-Tenant** and **Tenant-Tenant**.

House-Tenant relationship is evidently convenient for retrieving tenants who live in a certain house; it is clear however that there is no need to define it as a separate component of a model. Indeed, the occurrences <TenantID, HouseID> are stored in a **Tenant-House** relationship table, and the tenants living in a certain house, identified by the HouseID, can be retrieved from this table by the 'backward' query, which finds all **Tenants** related to a given HouseID. This is just the general approach

of OBEUS– when the relationship **A-B** is defined, the occurrences of the **B-A** relationship are retrieved on the base of the former. Let us note, however, that following the Leader-Follower view of the relationship we can retrieve the occurrences of the **House-Tenant** relationship based on the **Tenant-House** one, but not change it. **Tenant** is non-fixed entity and **House** is fixed one; Tenant only can alter the relationship with the House.

To continue with the Leader-Follower view, we must prohibit **Tenant-Tenant** relationship in OBEUS at all, just because it is a relationship between non-fixed entities and it is impossible to define leader and follower. Yes, it is, and if you have OBEUS GUI in front of you, you could note that **non-fixed** entities simply do not appear in the *Follower* control of the 'Define Relationship' dialog.

Isn't it a serious limitation? We do need **Tenant-Tenant** relationship for retrieving the fraction of friendly agents! – Right, and resolve this problem OBEUS applies the same solution as in **House-Tenant** case above. Namely, the pairs of neighbors are retrieved in OBEUS on the base of two properly defined relationships - **House-House** and **Tenant-House**.

Pairs of neighbors that is, occurrences of the **Tenant-Tenants** relationship, are retrieved in OBEUS in a transitive way:

Given tenant $\boldsymbol{T_1}$ → *retrieve the house* $\boldsymbol{H_1}$ *the tenant* $\boldsymbol{T_1}$ *lives in based on* ***Tenant-House*** *relationship*→ *retrieve all houses* $\boldsymbol{H_i}$ *neighboring to the house* $\boldsymbol{H_1}$ *based on* ***House-House*** *relationship* → *retrieve all tenants* $\boldsymbol{T_i}$ *living in all* $\boldsymbol{H_i}$ *retrieved.*

How this transitive chain can be implemented? One of the options is to leave it to the user who will code it in C#; the OBEUS' goal, however, is to help to the modeler whenever possible; that is why the transitive chain above is constructed in OBEUS automatically. This construction has the same limitation as the **House-Tenant** relationship above – the **Tenant-Tenant** relationship is also **Read-Only**, and none of two related tenants can destroy it, create, or change its characteristics. It is easy to accept this limitation, however. Indeed, if it is discarded, you will have to define, for instance, which of two neighboring tenants should leave the house first in case when each of them has insufficient number of friends in their (partially intersecting) neighborhood. Usually, the modeler simply does not have any definite assumption

about this fine mechanism of the dynamics of relationships between non-fixed entities. For myself, I prove this statement empirically – in all the publications I'm aware of, the relationships between non-fixed entities are expressed in a transitive way, explicitly or implicitly. Yes, this limitation OBEUS might be inconvenient if you develop psychological model, but, luckily, we model not love affairs, but the collective urban spatial phenomena.

Important is that the **Tenant-Tenant** relationship can be transitively defined in several ways, depending on the fixed objects (houses in case of Schelling model) taken as the middle link of the chain. This variety is an additional reason to make this relationship **Read Only**. To illustrate the ambiguousness let us assume for a moment that the capacity of a house can be more than one **Tenant**. In this case we can define two tenants as neighbors in two ways: first, when they live in the same house, second, when they occupy neighboring houses. If houses have floors, than we can introduce one more definition – tenants are neighbors if they occupy apartments on the same floor. OBEUS recognizes this ambiguity, and makes possible retrieving relationship between the non-fixed objects in *any* of logically possible way. The necessary variety of methods is *constructed automatically*, when entities and relationships of the <fixed, fixed> and <non-fixed, fixed> types are just defined by the modeler. For the basic Schelling model, where a house's capacity is always one tenant, one way only to define **Tenant-Tenant** relationship exists. The automatically constructed method is of the Tenant class of entities, and it's name is automatically generated by the system as **Tenant.GetRelatedTenantsViaHouses**. We will apply this method below, coding **Tenant**'s behavioral rules for the **Schelling model**.

5.6. General view of population properties and population methods

There are several other components of OBEUS, we ignored in the **Game of Life**. All of them are important for the **Schelling model**. These components are all located on the left, **Population** branch of the model tree (Figure 5.6).

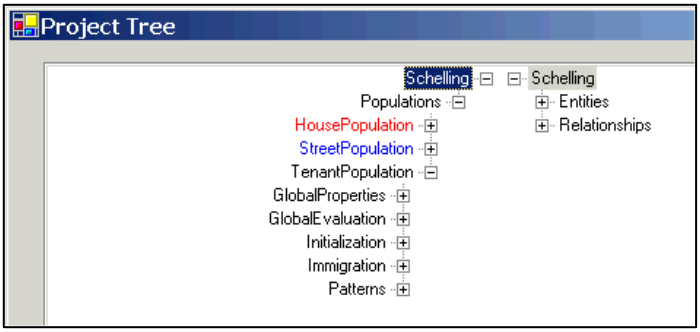


Figure 5.6: General view of the **Population** branch of the **Schelling model** tree

Population of the objects of a given class is created and named automatically when you define the class of entities. In Schelling model these are **HousePopulation** and **TenantPopulation**. Let us note that yet not all possible components of the **Population** branch are necessary in the **Schelling model**, and in this chapter we present only those of its compartments that are relevant for **Schelling model**.

Left branch of the model tree serves in OBEUS for defining **Population properties and rules** – all them are applied to *all* entities of a given class. These properties are updated once during iteration, after all entities of a given type have been considered. Population properties are divided between several sections of the left branch of the model tree.

5.7. Population properties of classes of entities of the Schelling model

Shelling model has at least one intuitive global parameter, which is common for all of the tenants – this is the threshold fraction of the friendly neighbors $\mathbf{F_{Th}}$ ($\mathbf{F_{Th}}$ remains population property until we decide that tenants can have different thresholds).

To define properties common for all the entities of a given class we have to focus on the **Global properties** and **Add** the new one using the left-hand side of the buttons at the bottom of the **Model tree**. Just as in case of the entity's properties, you have to name the property, define its type and check whether you want to see it displayed on the chart (Figure 42). The threshold fraction of the friendly neighbors $\mathbf{F_{Th}}$, which is the global property of the **TenantPopulation**, doesn't need to be displayed of the graph, for it doesn't change in time. OBEUS GUI makes possible to define the initial value of the global parameter (Figure 5.7)

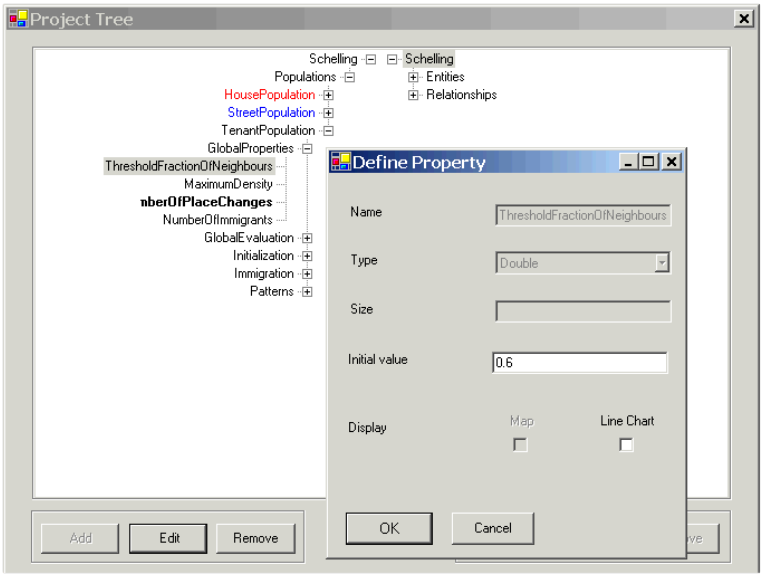


Figure 5.7: Dialog box for defining global properties and the definition of the threshold fraction of friendly neighbors $\mathbf{F_{Th}}$. Initial value of $\mathbf{F_{Th}}$ is set equal to 0.6

Threshold fraction of friends $\mathbf{F_{Th}}$ belongs to the group of population parameters which are necessary for running a model. Another, and not less important, group of global parameters consists of the aggregate population characteristics you, the modeler, want to store and display in order to follow and understand the model results. For example, in the **Schelling model**, I would define **numberOfMigrants** parameter of the **TenantPopulation**, which represents the number of tenants who changed their location - internal migrants - during the iteration. Just to link to the previous chapter - in **Game of Life** a number of **cells** in 'Alive' state can be the global parameter of the **CellPopulation**. There is evidently no need to define initial value of the global output characteristics of the model, they will be calculated on the base of the state of the model entities anyway.

5.8. Updating global properties

As mentioned above, the global parameters are re-estimated once per iteration, after the behavioral procedure has been executed for all entities of all classes. The rules of updating global parameters are also defined in the left-hand part of the model tree, in the **GlobalEvaluationMethods** section (Figure 5.8).

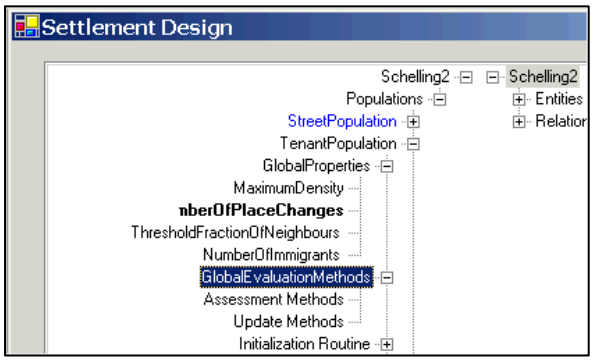


Figure 5.8: OBEUS GUI, **GlobalEvaluationMethods** section

The rules of global parameters updating are formulated in the same manner as entities' behavioral rules. Just as in the right branch of the **Model tree**, OBEUS distinguishes between **Assessment Methods** aimed at preparing information for evaluation and **Update Methods**, which aim at changing the global parameters and correspond to **Automation Method** of the entities.

As above, you formulate global parameters **Assessment Methods** and **Update Methods** with **C#**. The **C#** code below presents an example of updating of the **FractionOfBlack** global parameters – the population fraction of **B-Tenants**, which is important for the **Schelling model** when in- and out-migration from the city are included:

```
public void PerformUpdateMethod1()
{
   House[] houses = HousePopulation.GetHouses();
   foreach(House house in houses)
   {
      int blackCount = 0;
      Tenant[] tenants = house.GetTenantsForTenantHouse();
      if(tenants != null)
      {
         foreach(Tenant man in tenants)
         {
            if(man.Color)
            {
               blackCount++;
            }
         }
       house.PercentOfBlack = (double)blackCount / (double)house.NumberOfOccupied;
      }
      else
      {
         house.PercentOfBlack = 0;
      }
      house.PercentOfBlack1 = house.PercentOfBlack;
   }
}
```

Figure 5.9 presents the same code in C# window

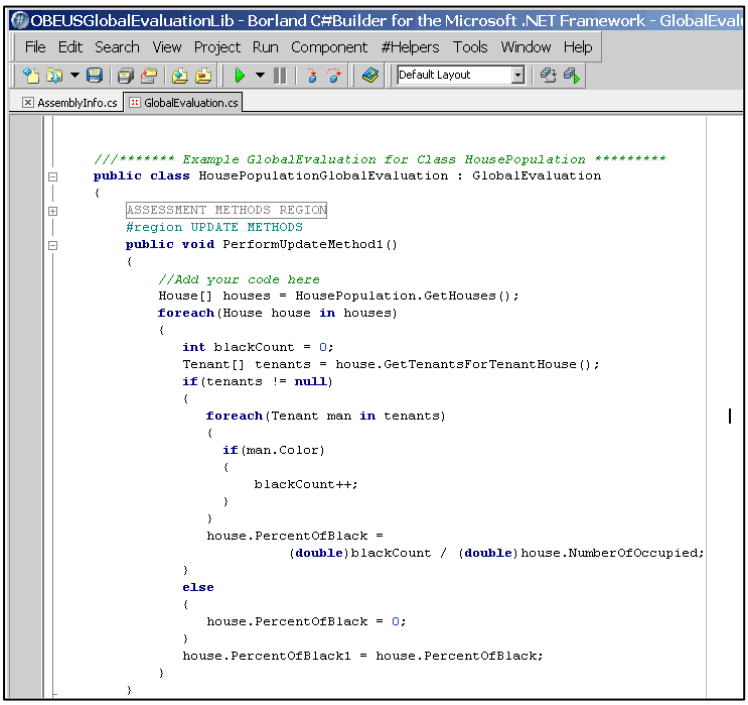


Figure 5.9: The **Update Method** for the **FractionOfBlack** global parameter of the **TenantPopulation**

5.9. Initialization Routines

The **Initialization routines** (Figure 5.10) aim at setting the values of objects' properties at the beginning of the simulation. They are also **C#** routines that should be coded by the modeler, and the component is organized just as the other C#-based components of OBEUS.

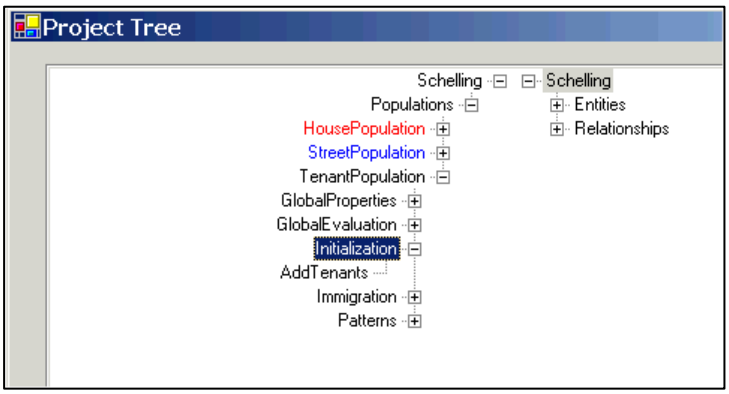


Figure 5.10: OBEUS GUI, **Initialization routines**

We have not employed **Initialization routines** in the **Game of Life**. In **Schelling model** we will employ them in a simplest way, which, nonetheless fits to the idea of the Schelling himself – we fill the houses randomly with **B** and **W** tenants. Let us assume that the probability that the cell is populated is $\mathbf{P_{initial\_populated\_fraction}}$ (one more global parameter of the model) and that the chance that a tenant of the population house is **B** equals $\mathbf{P_B}$ (one more global parameter), and, thus the chance to be **W** equals $\mathbf{1 - P_B}$. The idea of **Schelling model** is that whatever is the initial

residential distribution, if the value of $\mathbf{F_{Th}}$ is sufficiently high (tenants agree to stay in sufficiently friendly neighborhood only), then the residential pattern in the city eventually evolves to the segregated one. That is why random initial distribution is typical first stage of the **Schelling model** investigation. The C#-code of the random initialization of $\mathbf{P_{initial\_populated\_fraction}}$ houses with $100*\mathbf{P_B}$ percent of B-Tenants and $100*(1 - \mathbf{P_B})$ of the W-ones is as follows

```
public void ImmigrateTenants()
      {
         Random rnd = new Random();
         //Get Free Cells
         ArrayList freeHouses = new ArrayList();
         House[] houses = HousePopulation.GetHouses();
         foreach(House house in houses)
         {
            if(house.Capacity > house.NumberOfOccupied)
            {
               freeHouses.Add(house);
            }
         }
         bool IsBlack = false;
         for(int i=0;i<TenantPopulation.NumberOfImmigrants;i++)
         {
            if(freeHouses.Count > 0)
            {
               int seededNumber = rnd.Next(0, freeHouses.Count - 1);
               House freeHouse = (House)freeHouses[seededNumber];
               if(freeHouse.NumberOfOccupied < freeHouse.Capacity)
               {
                  Tenant man = new Tenant();
                  TenantPopulation.AddTenant(man);
                  man.SetRelationship(freeHouse);
                  freeHouse.NumberOfOccupied = freeHouse.NumberOfOccupied + 1;
                  if(IsBlack)
                  {
                     man.Color = true;
                  }
                  else
                  {
                     man.Color = false;
                  }
                  IsBlack = !IsBlack;
               }
            }
         }
      }
```

And Figure 5.11 presents this code in a C# window

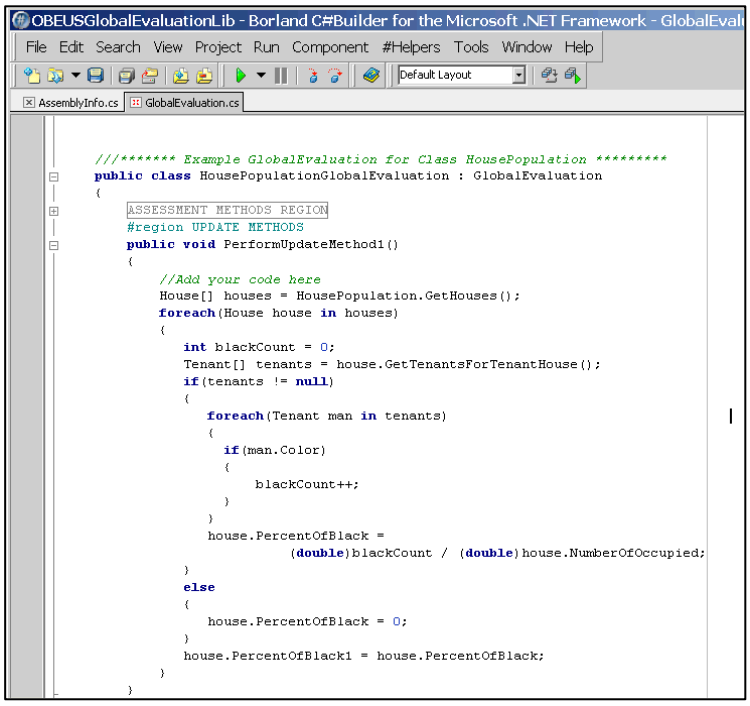


Figure 5.11: The **Initialization Method** for the random allocation of the **B-** and **W-Tenants** in the houses of the **Schelling model**

5.10. Immigration routines

Immigration routines are close to the **Initialization routines** by their meaning. They define the numbers and characteristics of the new (non-fixed) entities entering the system from outside, and initialize necessary objects and relationships when this happens. The difference between **Initialization routines** and **Immigration routines** is evident – **Initialization routines** are applied once, at the beginning of the first iteration, while **Immigration routines** are applied at each iteration. **Immigration routines** should be, as usual, coded by the modeler in **C#**, and Figure 5.12 presents some version of the Immigration routine as implemented in example of the Schelling model with immigration we supply

```
foreach(House house in houses)
        {
            if(house.Capacity > house.NumberOfOccupied)
            {
                freeHouses.Add(house);
            }
        }
        bool IsBlack = false;
        for(int i=0;i<TenantPopulation.NumberOfImmigrants;i++)
        {
            if(freeHouses.Count > 0)
            {
                int seededNumber = rnd.Next(0, freeHouses.Count - 1);
                House freeHouse = (House)freeHouses[seededNumber];
                if(freeHouse.NumberOfOccupied < freeHouse.Capacity)
```

```
            {
                Tenant man = new Tenant();
                TenantPopulation.AddTenant(man);
                man.SetRelationship(freeHouse);
                freeHouse.NumberOfOccupied = freeHouse.NumberOfOccupied + 1;
                if(IsBlack)
                {
                    man.Color = true;
                }
                else
                {
                    man.Color = false;
                }
                IsBlack = !IsBlack;
            }
        }
    }
```

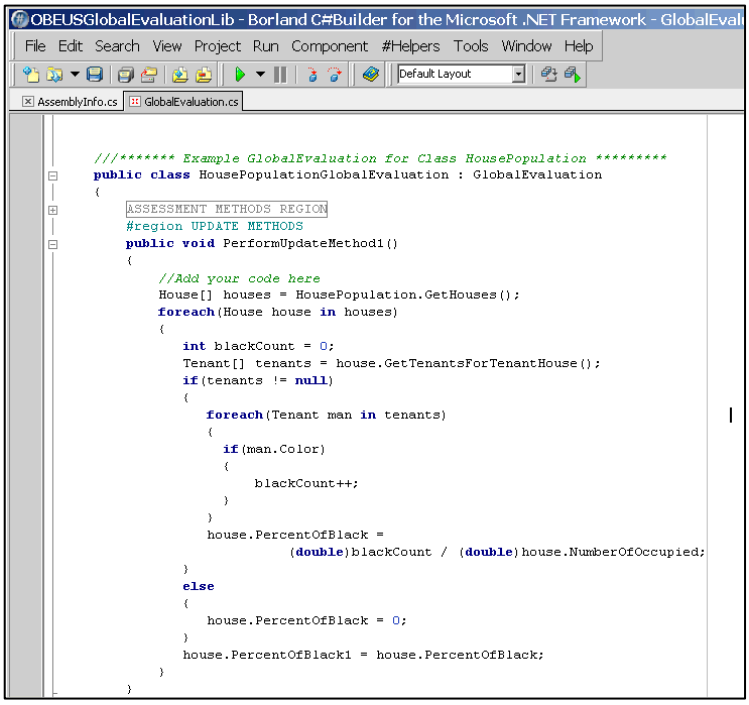


Figure 5.12: Possible code of the immigration routine

Figure 5.13 presents the complete Model Tree of the Schelling model with all non-empty branches expanded

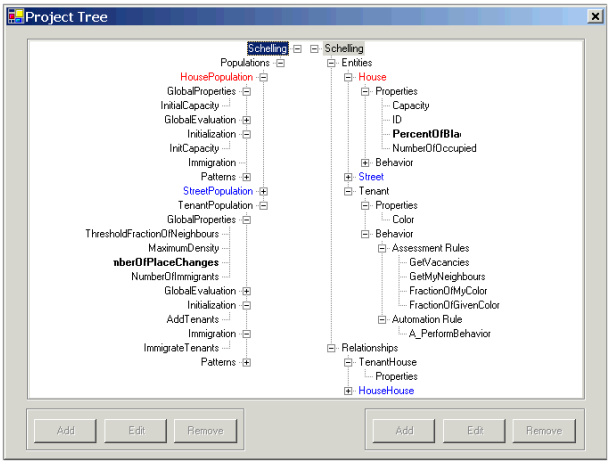


Figure 5.13: Full Model Tree for the generalized Schelling model

5.11. Running the Schelling model

To run **Schelling model** you have to do the same step as in the **Game of Life**. It is quite possible you might need more debugging, just because there is more lines of code, even if you used the examples above and typed the lines carefully. When you will be able to run the model, its outcome should look as presented in Figure 5.14.

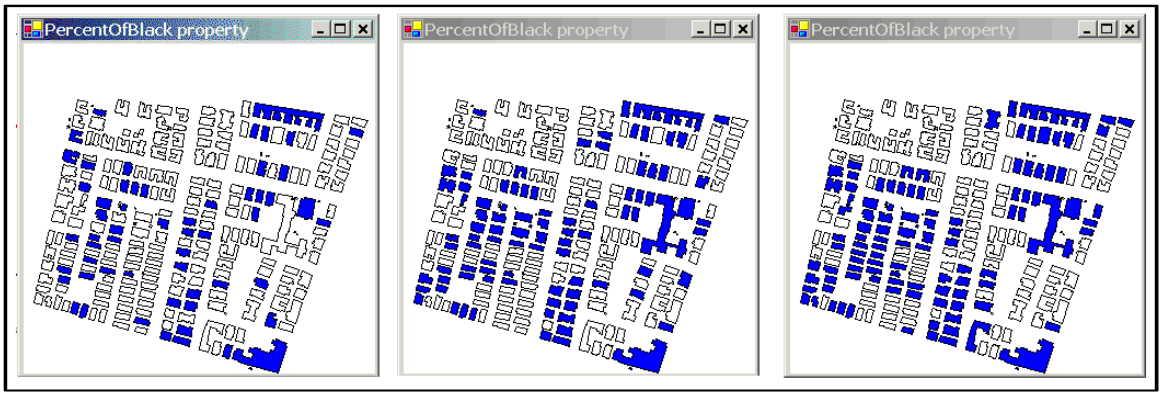


Figure 5.14: Typical time evolution of the **Schelling model with immigration** for $\mathbf{F_{Th}}$ = 0.6

5.12. Patterns - OBEUS component we have yet to employ

Before we proceed to the next chapter 6, which presents general concept of the Geographic Automata System (GAS) and OBEUS as an environment that implements GAS, let us introduce one more component of OBEUS, which aims at capturing emergence and self-organization in the models you develop. In OBEUS we call these *emerging* objects as **Patterns.**

The notion of **patterns** is beyond the level of the initial experience we want you to get from Game of Life and Schelling models, but can be employed in studying these models as well. You can skip this section until you will feel yourself familiar with OBEUS, or you proceed to the next chapter, where patterns are presented together with the general concept of the GAS and OBEUS. This section is for those who, just like me, like “working by example” when studying new software. We discuss patterns just after the **Schelling model**, because the latter is convenient for the initial presentation of the idea.

To illustrate self-organization of **patterns** with the **Schelling model** let us begin with the initial random distribution of the **B-** and **W-Tenants** (Figure 5.14).

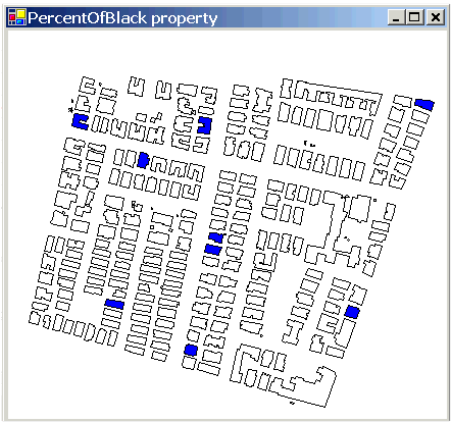


Figure 5.15: Randomly initialized **Schelling model**

As you remember, **Tenants** in **Schelling model** behave *locally*, each tries to resettle to a friendly neighborhood. Despite the local behavior of each tenant, the collective outcome of the Schelling tenants behavior is global. In a course of time **B-** and **W-Tenants** concentrate in big areas, one consisting of only **B-** and one of only **W- Tenants**, as you see you can see in Figure 5.13 (and most probably read the same in many papers). The emergence of this segregated pattern depends on the parameters of the model and the analysis shows that basically, this happens when $\mathbf{F_{Th}}$ is above 1/3 **Patterns** help the modeler to formalize the recognition of the segregation pattern of **B-** and **W-Tenants**. Comparing to the other components of OBEUS, **Patterns** section (Figure 5.15) is similar to **Behavior** and **GlobalEvaluation** sections, which aim at description of the essence of any model – the rules of entities' behavior and global parameters update.

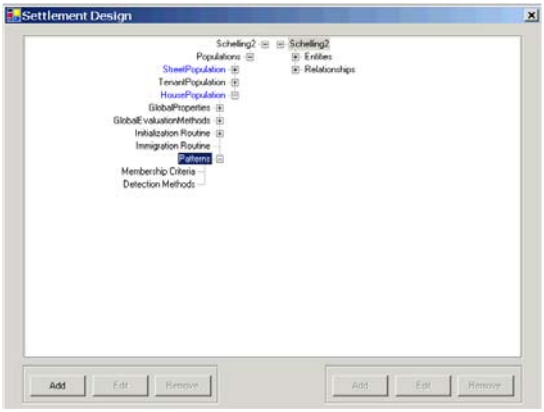


Figure 5.16: OBEUS GUI **Patterns** component

Just as in **Behavior** and **GlobalEvaluation** sections, there is no, and cannot be, general rules that recognize patterns and estimates their characteristics. As usual, OBEUS presents instead the possibility to formulate these rules, which we call **Membership Criteria** and **Detection Method**. The **Membership Criteria** recognize entities that belong to the self-organizing pattern, for it can easily happen

that not all entities belong to it. Say, B-Tenants of the Schelling model may be segregated in the northern part of the city, but not in the southern. **Detection method** aims at recognizing the pattern. Different from all the other compartments of the OBEUS, we do not insist that this organization is sufficient or even convenient for recognizing emerging spatial patterns. In what follows, we describe the approach we employ in our model studies, which does utilize the above separation between **Membership Criteria** and **Detection Method**.

You will not be surprised when we say that the approach we use for detecting patterns follows employs relationships. Shortly, it repeats human view of the spatial aggregation – a pattern is 'sufficiently large' domain, 'sufficiently densely' filled with the elements sharing common properties. There is no problem to recognize entities sharing common properties in OBEUS, and the problem reduces to recognition of the areas, where the density of these similar entities is high. Here we employ OBEUS' complete knowledge of relationships - density of the entities sharing set of properties **C** is 'sufficiently high' when each entity has 'sufficiently high number of neighbors' that share **C-**properties.

Let us define a measure of similarity $\mathbf{S}(E_1, E_2)$ between two arbitrary entities E_1 and E_2 of the same entity class. The method that calculates this measure will be the first membership rule. The second membership rule will estimate the fraction of the entities X related to a given entity E that differ from E according to the similarity measure **S**(E, X) on less than Δ.

In case of Schelling model, for example, we can define $\mathbf{S}(T_1, T_2) = 0$ if tenants T_1 and T_2 have the same color and $\mathbf{S}(T_1, T_2) = 1$ if colors of T_1 and T_2 are different (first membership rule), and to consider entity T as the member of a pattern if the faction of the neighbors of its color is 0.6 or higher (that is $\Delta = 0.6$).

The existence of the B-Tenants "surrounded" by sufficiently high number of the B-neighbors is not enough for being a "pattern." We detect the pattern when we see 'sufficiently continuous' area filled with B-tenants, each having sufficient number of B-neighbors. To recognize these continuous areas, we apply recursive Detection rule – to begin with arbitrarily B-tenant, to apply Membership Rule 2 and to determine if the B-tenant has sufficient number of B-neighbors, then proceed with the neighbors, apply Membership Rule 2 to each of them, etc. The Detection rule we apply demands

has a parameter - the number of entities included into the set. If they are few, the result is not a pattern.

It can be proved that the result of this test does not depend on the selection of the B-tenant to begin with. The result of the recursive algorithm above will be the number of sets of B-tenants, not necessarily contiguous (there will be many holes if Δ is not close to 1) in each of which each B-tenat has sufficiently high number of B-neighbors

Another version of the Membership Rule can be the test of sufficient fraction of B-neighbors irrespective of the type of tenant itself. The outcome of this rule will be the set of continuous areas. Figure 5.17 illustrates the latter idea and constructs for Δ = 0.5:

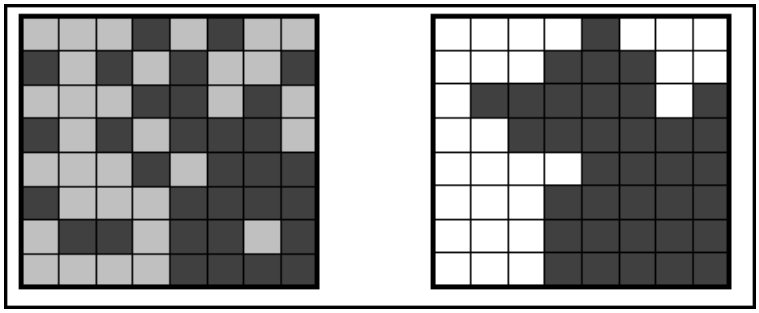


Figure 5.17: (a) Actual distribution of the B-tenants and (b) the resulting **Pattern** of the B-Tenants

Formally the method can be presented as follows:

Let some predicate - criteria C - is defined on the entities of class E.

Let us mark the entities satisfying C as C-TRUE and the rest as C-FALSE. The method that constructs pattern R_C containing sufficient number of C-True entities is as follows:

buildPattern(float $F_{CThreshold}$, int $N_{EThreshold}$)
{
Construct empty temporary pattern R;
Insert Ent into R;
While there are new entities in R {
Loop by entities currEnt recently included into R
Get list $NBRH_R$ of neighbors of currEnt
Calculate fraction F_C of C-TRUE Entities in $NBRH_R$
If F_C > $F_{CThreshold}$ then {Include ALL entities from $NBRH_R$ in E}
Else remove currEnt from R}
Calculate N_R - number of buildings in R
If N_R > $N_{RThreshold}$ then {Mark all buildings in R as belonging to D_C}

Else {drop R}
}

Based on buildPattern(), we can easily construct all regions R_C satisfying criterion C:

buildAllPatterns (float $F_{CThreshold}$, int $N_{EThreshold}$):
Loop by all entities Ent in a settlement {
If Ent is marked as C-TRUE then {
If Not (Ent$\in D_C$) then {buildPattern($F_{CThreshold}$, $N_{EThreshold}$)}
}}

Remarks:

- Pattern may contain holes.
- Entities can belong to domain D_C despite being C-FALSE.
- Threshold value $F_{CThreshold}$ should be sufficiently high to reflect intuitive understanding of a domain as an area where *most* entities satisfy criteria C.
- The value of $N_{RThreshold}$ determines the minimal size of domain

OBEUS examples contain SchellingP project, which includes pattern detection in a way described above. We still didn't decide finally how to manage patterns in OBEUS and will appreciate your suggestions.

5.13. What did we learn with Schelling model?

We can now list the main OBEUS actions we studied with the Schelling model in addition to those we investigated with the Game of Life:

1. How to define non-fixed objects
2. How to locate non-fixed objects on the base of their relationships with the fixed ones
3. How OBEUS works with the relationships that are not defined directly be the user, as the relationship between non-fixed objects of the same or two different classes
4. How global parameters of the population of the objects of the given class are introduced and their initial values are defined and visualized
5. How global parameters that describe aggregate characteristics of the population of a given class are introduced and visualized
6. How the rules of assessment and evaluation of the global parameters can be coded in C# and debugged with the Borland C# compiler
7. How initial conditions of the model are coded in C# and debugged with the Borland C# compiler
8. How immigration processes are coded in C# and debugged with the Borland C# compiler
9. How self-organization processes can be interpreted with the help of OBEUS patterns

6. Build-in methods of OBEUS

The internal methods of OBEUS are few. Yes, the definition of entities, and relationships, their properties, settings of the output demand tens of classes and hundreds of methods, but all that is done by the environment and the user employs them via GUI. All of us like modeling, and not at all programming, and one of our main goals with OBEUS was to deliberate you and ourselves of the dirty part of coding and to leave us with the essence of the model, which cannot be unified – entities behavior.

6.1. An idea of automatic construction of methods

The methods that OBEUS presents to developers aim at more of the same - to reduce the code the user has to write when programming the behavioral rules. Basically, each model is unique and not so much generic methods could be formalized. Those that could are **constructed by OBEUS on the base of the entities and relationships names as defined by the user**, just after the new entities and relationships are defined. In this way we can hold these methods encapsulated in the necessary classes. To enable these methods the user must save the model; that is why we repeat the reminder to push Save from time to time when working with OBEUS.

6.2. Conventions

In what follows we assume that some entities, as XXX or YYY, relationships between them, as XXX_XXX or YYY_XXX, and populations, as XXXPopulation or YYYPopulation are defined. As we said above, the names of the methods are built from the names of the entities and relationships.

Generally, OBEUS builds internal methods with each new class of entities and relationships defined (actually, when you push **Save**). When constructed, the built-in methods appear in the prompt, after the ‘dot’ is typed in the Borland C# window. We tried our best to make the names of the build methods self-explainable, and the rule of their construction is as follows:

OBEUS Built-in method name := BasePart1 + [VarPart1] + BasePart2 + [VarPart2]

+ … + {Optional “RO” part} + (*ClassOfObject ObjectVariable)*

BasePart of the name reflects the methods itself, as “Add”, or “Get”;

VarPart is used to specify the names of the OBEUS entities defined via GUI

OptionalPart “**RO**” (ReadOnly) **part** is used in case of parallel synchronization mode

For example, in the name of the method **AddXXX (***XXX e***)**: **BasePart1 ~ Add**; VarPart1 **~** XXX; Optional part **~** “”, *ClassOfObject* **~** *XXX*; *ObjectVariable* **~ *e***.

Presenting the examples of the methods, we employ the default style of the Borland C# and mark red the lines with the method employed. The text of the comment is always given before the commented line of code.

6.3. Remark on referential integrity

As we repeat several times in this manual, in OBEUS we manage entities and relationship according to the database fashion. An important feature of this approach is preserving **referential integrity**. Shortly, it means that if you have an entity A, which is related to the entity B, and you want to delete an entity A, the database should recognize A’s participation in relationship and do not permit deleting of A until relationship between A and B is not destroyed.

We do not check referential integrity in this version of OBEUS (while we will incorporate it into the next one). That is why you have to take care of destructing relationship between A and B before deleting A by yourself. Basically, nothing terrible happens if you forget to delete the relationship – the model can yet function, while you can get an entity A when you retrive all the entities related to B, and this can cause runtime errors.

Don’t forget about your responsibility for referential integrity of your entities!

6.4. Population methods

There are two population methods, aimed at including entities of the given class into population of the entities of this class or remove (delete) them from the population. OBEUS uses populations in order to distinguish between the entities that are

supposed to really 'exist' and temporarily objects of the classes of entities used in the model code. For example, if we want to know the number of Alive cells in the Game of Life, we have to loop over all cells of the grid, and not over all objects of the Cell class that are currently introduced. By the same reason, for the Game of Life model we cannot add Cell entities to the CellPopulation, or delet them form there, but we can easily extend Schelling model and include into it the processes of immigration and emigration. In the latter case we have explicitly say to OBEUS which ones of the objects of the Tenant type are immigrants and which are created just for storing some temporarily information. In the same way we have to remove a Tenant entity from TenantPopulation, when a tenant emigrates from the model city.

6.4.1. **GetXXXs** – Retrieves all entities of the XXXPopulation class

Syntax: **GetXXXs ()**

Purpose: Retrieves all objects of the XXX-class

Returns: Array of XXX-class objects

Exceptions: None

Example:

XXX := House (Class House, fixed entities, defined via OBEUS GUI)

XXXPopulation := HousePopulation (Class HousePopulation, created automatically when the class House is defined)

Method := **GetHouses**() (Method GetHouses, created automatically when HousePopulation class is created)

Use: Typically is used to retrieve all objects of a given class for further loop processing, for example to find out those satisfying some condition. The code below estimates the overall capacity of the houses of HousePopulation:

Code:

```
//Retrieve all object of the HousePopulation
House[] cityHousesList = GetHouses();
//cityCapacity will store the overall capacity of a city
int cityCapacity = 0;
foreach (House currentHouse in cityHouses)
{
   //Update overall number of occupied apartments
   cityCapacity += currentHouse.Capacity;
```

}

6.4.2. **AddXXX** - Include an object of the XXX-class into a population of XXX-entities

Syntax: **AddXXX (***XXX e***)**

Purpose: Includes object e of the XXX-class into population of the XXX-entities

Returns: None

Exceptions: None

Example:

XXX := Tenant (Class Tenant, non-fixed entities, defined via OBEUS GUI)

XXXPopulation := TenantPopulation (Class TenantPopulation, created automatically when the class Tenant is defined)

Method := **AddTenant**(Tenant e) (Method AddTenant, created automatically when TenantPopulation class is created)

Use: Typically is used to simulate the processes of immigration and birth. The code below creates new immigrant tenant and includes it into Tenant population:

Code:

```
//create new object of the class Tenant
Tenant foreigner = new Tenant();
//Set object's properties
foreigner.Age = 20;
// Include new object into population of entities
TenantPopulation.AddTenant(foreigner);
```

6.4.3. **RemoveXXX** (XXX e) - Remove an XXX-entity from population of XXX-entities

Syntax: **RemoveXXX** (XXX e)

Purpose: To delete an entity e of class XXX from the population of XXX-entities

Returns: None

Exceptions: None

Example:

XXX := Tenant - Class Tenant of non-fixed entities is defined via OBEUS GUI

XXXPopulation := TenantPopulation - Class TenantPopulation is created automatically when the class of entities Tenants is defined

Method := **RemoveTenant**(Tenant e) (Method RemoveTenant is created automatically when TenantPopulation class is created)

Use: Typically used to simulate the processes of emigration and death. The piece of code below aims at removing Tenant resident from the population of tenants:

Code:

```
//Retrieve the houses the resident occupies (see description of GetRelatedHouses()
//below) – it is actually one house only, but we should account for the possibility of
//Many:Many relationship between non-fixed and fixed entities
House[] myHouses = resident.GetRelatedHouses();
//To preserve referential integrity, destroy relationship with the house before
deleting the entity (see description of RemoveRelationship() below)
foreach (House currentHouse in myHouses)
{
   //destroy relationship with the house
   resident.RemoveRelationship(currentHouse);
   //One tenant less in each of myHouse
   currentHouse.NumberOfOccupied = currentHouse.NumberOfOccupied - 1;
}
//Remove a tenant from TenantPopulation
TenantPopulation.RemoveTenant(resident);
```

This example employs two built-in functions: GetRelatedHouses(), see section 5.4 below and RemoveRelationship(), see sections 5.5.3 below

6.5. Entity methods

The goal of the entity methods is not entities, but those related to them

6.5.1. **GetRelatedXXXs** - Retrieve XXX-entities related to a given entity

Syntax: **GetRelatedXXXs**

Purpose: The method allows systematically apply from the entity of YYY class to those related to it of the XXX class. The relationship between entities of YYY and XXX classes should be defined via OBEUS interface. Generally, the number of entities related to the given one is more than one; that is why the method return array of the entities. The method can be applied to both directly defined and implied relationships. The existence of relationship between the entities does not presumes that every entity form one class must be related to some entity of the other. Some entities of one class can be yet unrelated to any entity of the other (in database theory it is called as *non-obligatory membership)*. For the entity of, say, class YYY,

which is not related to any entities of the class XXX, despite the relationship YYY_XXX being defined, the method returns NULL.

Returns: An array of entities of the related class related to a given entity

Exceptions: None

Note: This function is defined for both directly defined and implied relationships.

That is in typical case of two classes XXX of the fixed entities and YYY of the non-fixed entities and two classes of relationships XXX-XXX YYY_XXX defined, three methods are constructed:

GetRelatedXXXs() in class XXX (this method is based on the direct relationship)

GetRelatedXXXs() in class YYY (this method is based on the direct relationship)

GetRelatedYYYs() in class XXX (this method is based on the implied relationship)

Use: This is the most frequently used method of OBEUS

Example 1:

XXX := House (Class House of fixed entities is defined via OBEUS GUI)

XXX_XXX := HouseHouse (Class HouseHouse of neighborhood relationships is created automatically when the class of entities House is defined)

Method := **GetRelatedHouses()** (Method **GetRelatedHouses()** is created automatically in a class House when HouseHouse class is created)

Code:

```
//Build the list of houses neighboring to the given house
private ArrayList GetVacancies()
//Create empty list of houses with vacancies
ArrayList vacancies = new ArrayList();
//Retrieve neighbors of an entity myhouse
House[] houseNeighbors = myhouse.GetRelatedHouses();
//If there exist neighboring houses analyze them
if (houseNeighbors != null)
{
   //Loop by neighboring houses
   foreach (House currentHouse in houseNeighbors)
   {
      //if there are vacancies in currentHouse
      if (currentHouse.Capacity > currentHouse.NumberOfOccupied)
      {
         //include the house into a list of houses with vacancies
         vacancies.Add(currentHouse);
```

```
      }
   }
   //return list of vacancies
   return vacancies;
}
```

Example 2:

XXX := House (Class House of fixed entities is defined via OBEUS GUI)

YYY := Tenant (Class Tenant of non-fixed entities is defined via OBEUS GUI)

XXX_XXX := HouseHouse (Class HouseHouse of neighborhood relationships is created automatically when the class of entities House is defined)

YYY_XXX := TenantHouse (Class TenantHouse of relationships is defined via OBEUS GUI)

Method := **GetRelatedHouses**() (Method **GetRelatedHouses** is created automatically in a class Tenant when TenantHouse relationship class is created)

Code:

```
//Retrieve the house resident occupies
House[] myHouses = resident.GetRelatedHouses();
House myHouse = myHouses[0]
//Create the list of houses neighboring to the given tenant
ArrayList vacancies = new ArrayList();
//Retrieve neighbors of an entity myhouse
House[] houseNeighbors = myhouse.GetRelatedHouses();
//If there exist neighboring houses analyze them
if (houseNeighbors != null)
{
   //Loop by neighboring houses
   foreach (House currentHouse in houseNeighbors)
   {
      //if there are vacancies in currentHouse
      if (currentHouse.Capacity > currentHouse.NumberOfOccupied)
      {
         //include the house into a list of houses with vacancies
         vacancies.Add(currentHouse);
      }
   }
}
```

Example 3:

XXX := House (Class House of fixed entities is defined via OBEUS GUI)

YYY := Tenant (Class Tenant of non-fixed entities is defined via OBEUS GUI)

XXX_XXX := HouseHouse (Class HouseHouse of neighborhood relationships is created automatically when the class of entities House is defined)

YYY_XXX := TenantHouse (Class TenantHouse of relationships is defined via OBEUS GUI)

Method := **GetRelatedTenants**() (Method **GetRelatedTenants** is created automatically in a class Tenant when TenantHouse relationship class is created)

Code:

```
//Retrieve the house resident occupies
House[] myHouses = resident.GetRelatedHouses();
House myHouse = myHouses[0];
//Create the empty list of houses neighboring to the given tenant
ArrayList myneighbors = new ArrayList();
//Retrieve houses neighboring to myhouse
House[] houseNeighbors = myhouse.GetRelatedHouses();
//If there exist neighboring houses analyze them
if (houseNeighbors != null)
{
   //Loop by neighboring houses
   foreach (House currentHouse in houseNeighbors)
   {
      //retrieve neighbors if there are vacancies in currentHouse
      Tenant[] houseTenants = currentHouse.GetRelatedHouses()
      //if there are tenents in the house
      if (houseTenants != null)
      {
         //loop by tenants in the house
         foreach (Tenant currentTenant in houseTenants)
         {
            //include the tenant into a list of neighbors
            myneighbors.Add(currentTenant);
         }
      }
   }
}
```

6.5.2. **SetRelationship**(XXX e) - Creates new relationship for a given entity with an entity e

Syntax: **SetRelationship**(XXX e)

Purpose: This method creates and returns relationship between given entity and XXX-class entity e. If a given entity is of YYY-class, then relationship YYY_XXX should be defined via OBEUS GUI prior to employing this method

Returns: Relationship

Note: function is constructed for each relationship between not-fixed – fixed entities defined via OBEUS dialog box

Exceptions: None

Example:

XXX := House (Class House of fixed entities is defined via OBEUS GUI)

YYY := Tenant (Class Tenant of non-fixed entities is defined via OBEUS GUI)

YYY_XXX := TenantHouse (Class TenantHouse of relationships is defined via OBEUS GUI)

Method := **SetRelationship(House house)** (Method **SetRelationship()** is created automatically in a class Tenant when TenantHouse relationship class is created)

Methods:

Use: Typically is used to indirectly locate non-fixed entity

Code:

```
//It is supposed that we have already calculated the fraction of the friendly
//individuals within the neighborhood of a given location House vacancy and
// stored this value in the double fractionOfMyColor variable
//check if fractionOfMyColor is sufficient to accupy the vacancy
if (fractionOfMyColor >= TenantPopulation.ThresholdFractionOfNeighbours)
{
   //occupy the vacancy
   householder.SetRelationship(vacancy);
   //update the vacancy state
   vacancy.NumberOfOccupied = vacancy.NumberOfOccupied + 1;
}
```

6.5.3. **RemoveRelationship**(XXX e) - Remove relationship of a given entity with an XXX-class entity e

Syntax: **RemoveRelationship**(XXX e)

Purpose: Removes relationship of a given entity with an XXX-class entity e

Returns: None

Exceptions: None

Note: Is constructed for each relationship between not-fixed – fixed entities defined via OBEUS GUI.

Example:

XXX := House (Class House of fixed entities is defined via OBEUS GUI)

YYY := Tenant (Class Tenant of non-fixed entities is defined via OBEUS GUI)

YYY_XXX := TenantHouse (Class TenantHouse of relationships is defined via OBEUS GUI)

Method:= **RemoveRelationship**(House house)

Use: Typically is used to code the process of leaving

Code:

```
//retrieve the houses resident occupies. It is always one house, while we
consider more complex situation of 1:Many TenantHouse relationships
House[] myHouses = resident.GetRelatedHouses();
//loop by myHouses
foreach (House currentHouse in myHouses)
{
   //destroy relationship with the house
   resident.RemoveRelationship(currentHouse);
   //update currentHouse parameters
   currentHouse.NumberOfOccupied = currentHouse.NumberOfOccupied - 1;
}
```

6.5.4. **GetRelationShipsYYYXXX**() - Retrieves all relationships of a given entity

Syntax: **GetRelationShipsYYYXXX**()

Purpose: Retrieves all relationships of a given entity

Returns: Array of YYYXXX relationships

Exceptions: None

Note: Is constructed for each relationship defined via OBEUS GUI.

Example:

XXX := House (Class House of fixed entities is defined via OBEUS GUI)

YYY := Tenant (Class Tenant of non-fixed entities is defined via OBEUS GUI)

YYY_XXX := TenantHouse (Class TenantHouse of relationships is defined via OBEUS GUI)

Method := **GetRelationShipsTenantHouse()**

Use: Typically is used for investigating the properties of relationship

Code:

```
// apartmentsForSale is anumber of houses a resident possesses more than 5 years
int apartmentsForSale = 0;
//retrieve all the houses a resident possesses
TenantHouse[] myHouseAffairs = resident.GetRelationshipTenantHouse();
//loop by houses
foreach (TenantHouse currentHouseAffair in myHouseAffairs)
{
   //if the property is in possession more than 5 years
   if currentHouseAffair.year > 5
   {
      //count this apartment
      apartmentsForSale++;
   }
}
```

6.6. Transitive retrieve methods

Transitive retrieve methods are all “Read-Only”. They are built in order to simplify retrieving of indirect relationships between non-fixed entities; direct definition of the latter is prohibited (and impossible) in OBEUS, which follows leader-follower pattern of relationships. As you know, to keep the system universal, OBEUS interprets non-fixed—non-fixed relationships through the chain of relationships between non-fixed and fixed entities.

The standard example is a definition of two tenants as neighbors if they live in the neighboring houses. If we want to retrieve the neighbors of a given tenant we have to apply three-step procedure:

Three-step transitive relationship

- Retrieve tenant’s house H, via TenantHouse relationship →
- Retrieve houses neighboring to H, via HouseHouse relationship →
- Retrieve the tenants of the neighboring house via TenantHouse relationship.

As can be easily noted, these procedures depend on class chosen as an intermediate link in the chain. If, for example, Tenants are related to Pubs via TenantPub relationship, then tenants who might meet in the Pub are retrieved along the chain, which intermediate link is Pub-class instead of the House-class in the example above. OBEUS takes responsibility of the chain process above and provides two series of the

read-only methods for that

6.6.1. **GetRelatedYYYsThruZZZs_Order0**() - Retrieve related entities via neighbors of the zero order

Syntax: "GetRelated" + [CurrentEntity] + "sThru" + [RelatedEntity] + "s_Order0"

Purpose: To simplify retrieving indirectly related non-fixed entities

Returns: Array of non-fixed entities

Exceptions: None

Note: None

Example:

YYY := Tenant

XXX := House

YYY_XXX := TenantHouse

Method := Tenant[] GetRelatedTenantsThruHouses_Order0()

Use: Just according to the its main goal, to retrieve non-fixed entities to a given non-fixed entity via the nearest neighbors

Code:

```
//calculate the number of black neighbors among the nearest neighbors of the resident
int numberOfBlacks = 0;
//retrieve the neighbors
Tenant[] myNeighbors = resident.GetRelatedTenantsThruHouses_Order0();
//if there are neighbors
if (myNeighbors != null)
{
   //loop by neighbors
   Foreach (Tenant currentNeighbor in myNeighbors)
   {
      //if a neighbors is black
      if (currentNeighbor.Color = Black)
      {
         //count black neighbor
         numberOfBlacks++
      }
   }
}
```

6.6.2. **GetRelatedYYYsThruZZZs_Order1**() - Retrieve related entities via

neighbors of the first order

Syntax: "GetRelated" + [CurrentEntity] + "sThru" + [RelatedEntity] + "s_Order1"

Purpose: To simplify retrieving indirectly related non-fixed entities

Returns: Array of non-fixed entities

Exceptions: None

Note: None

Example:

YYY := Tenant

XXX := House

YYY_XXX := TenantHouse

Method := Tenant[] GetRelatedTenantsThruHouses_Order1()

Use: Just according to the its main goal, to retrieve non-fixed entities to a given non-fixed entity via the nearest neighbors

Code:

```
//calculate the number of black neighbors among the nearest neighbors of the resident
int numberOfBlacks = 0;
//retrieve the neighbors
Tenant[] myNeighbors = resident.GetRelatedTenantsThruHouses_Order1();
//if there are neighbors
if (myNeighbors != null)
{
   //loop by neighbors
   Foreach (Tenant currentNeighbor in myNeighbors)
   {
      //if a neighbors is black
      if (currentNeighbor.Color = Black)
      {
         //count black neighbor
         numberOfBlacks++
      }
   }
}
```

6.7. Relationships' methods

Currently, there are no relationships' methods in OBEUS

6.8. Note on programming style

To keep the proper programming style, implement your own method – Assessment and Behavior rules, Initialization routines, etc., - according to the following template:

```
try
{
   …Your code…
}
catch(Exception ex)
{
   Trace.Write(ex.Message)
}
```

The content of the trace window is visible during the runtime.

For example, the following block of code wrongly ignores the possibility of a house which ahs no neighboring houses at all. When calculating the fraction of neighbors of the same color as house's resident, it sends the exception message of "dividing on zero".

```
try
{
   int numberOfMyColor = 0;
   ArrayList myNeighbours = GetMyNeighbours(resident);
   foreach (Tenant currentNeighbour in myNeighbours)
   {
        if (currentNeighbour.Color == resident.Color)
        {
          numberOfMyColor++;
        }
   }
   double fractionOfMyColor = (double) numberOfMyColor/myNeighbours.Count;
}
catch(Exception ex)
{
  Trace.Write(ex.Message)
}
```

If resident is located in a house which has no neighboring houses at all, then myNeighbours.Count equals zero and the following message will be obtained in a message window

THE BEST IS TO CAPTURE SCREEN OF THE MESSAGE HERE

6.9. Note on “How to begin”

Two examples of the Quick Start chapters and the description of the built-in methods in the chapter above provide the basic understanding of main components of OBEUS.

Your own model, the one you have in mind spending time for getting used to OBEUS, is, undoubtedly, more complex. Don’t run with it if it is your first experience with OBEUS. Even if you definitely feel how to present the classes of objects, relationships between them, the rules of objects’ behavior, etc., we strongly recommend formulating that in semi-theoretical form before you begin to build an OBEUS shell for it, just as if you try to write down incomplete equations, on the first steps of formulating analytical models. Pay attention on separating between Assessment rules and Behavioral rule, input parameters, initial conditions. Different from the “development from scratch” coding of the model in OBEUS takes very short time; there is no need to write most of the code you use to.

6.9.1. Time flow is important

Special attention should be put on the time flow in the model. As you already know, in parallel mode objects are “frozen” at the end of the time step. Temporary versions of all objects are created, which are all updating and behaving reacting to the frozen ones, until at the end of the time step the set of the frozen objects is substituted by the set of the temporarily ones. In sequential mode the changes in any object are immediately available to the others.

OBEUS time-flow dialog forces you to think far beyond the difference between parallel and sequential updating. There is another aspect – the order of updating you establish between entities of different classes. Say, tenants populate houses which price changes in time. What should be done first – prices update or tenant decision to leave? In this and many similar cases modeler does not case much and does “something reasonable”. Often it really does not matters, but sometimes all the aspects of the model behavior become dependent on this order, as say, when the prices can raise twice during the model time step of three months and all the tenants are students who rent the apartments and may leave easily at the end of the term, but not in the middle. OBEUS forces you to think about this part of every model straightforwardly, and we consider that as a great advantage, which makes models

comparable and fully understandable.

An advice: Usually, it makes sense keep two version of the model simultaneously with the parallel and sequential updating and compare the results obtained.

6.9.2. Be careful when changing spatial and temporal resolution of the model

Delayed

7. OBEUS versus another model styles and systems

7.1. What are the “inconvenient for OBEUS” models?

These are models where all entities are of non-fixed type, the neighbors are determined on the base of distances, and the neighborhoods are wide. If all these conditions hold OBEUS’ approach of locating by relationship and building relationships between the fixed entities in advance looses its advantages. You still can get much from OBEUS, as the following example of flocks model demonstrates.

Flock model scheme (REF): Flocks of birds (points) fly in 2D or 3D space, each adjusting its vector of velocity towards average velocity vector of the neighboring points within some constant view radius. Sometimes some of the birds change their velocity vector by internal or external reason, independently of the neighbors’ behavior and if these deviations are sufficiently strong and/or frequent the flock flow destabilizes, and its dynamics become complex and turbulent.

OBEUS approach works fine with the flock model, if at least one of the conditions holds – either the view radius is small, or we are not interested in fine tuning of the dependency of the neighbor’s influence on distance. In each of these cases we can divide 2D space into cells and 3D space into cubes, roughly define the dependency function at resolution of cell or cubes and follow the logic of Schelling model. We will have to substitute assessment function of Schelling model by that for flocks.

The above fake discretization of space does not work if we want to investigate the fine influence of the dependency function. In this case, if we preserve the discrete view of space, the fine division of 2D or 3D space into cells should be hold. Tables of relationships in this case become huge and to retrieve the related birds from the table might take just the same time as recalculating them anew.

Our view of the situation in this case is close to earth – we don’t believe you will begin with the fine tuning of the dependency function. So you could get quite a lot of performance form OBEUS’ GUI and built-in functions when formulating flocks behavior, and become tie with the standard, based on permanent recalculating of the neighbors, approach when you will want to investigate some specific and non-rough changes in neighbors’ influence. Whatever, don’t forget that numeric methods of the differential equations solution always demand discretization of space to be applied.

7.2. OBEUS versus Repast and other libraries of agent-based models' methods

To be done

8. Geosimulation, Geographic Automata Systems, and OBEUS-related publications

Besides direct objective to teach you how to use the OBEUS software, this long manual has a hidden goal – to convince you that OBEUS is not only the software, but also the theory, that proposes the general view of the agent-based systems in case we are interested in their collective behavior. OBEUS is actually an implementation of the theory of Geographic Automata Systems (GAS). You could know more about GAS theory and Geosimulation, as a general approach to investigating complex geographic systems reading next chapter of this manual. Here are the main sources, where Geosimulation, GAS, and OBEUS are discussed.

First, this is a book of Itzhak Benenson and Paul Torrens:

Benenson, I. and P. M. Torrens (2004a). Geosimulation: automata-based modeling of urban phenomena. London, Wiley, 232pp,

where you could find the theory and, especially, a bulk of applications developed during last decades in various fields, from pedestrian and vehicle movement to city growth and sprawl.

Several recent papers discuss the theoretical principles of Geosimulation, GAS and OBEUS in more details than the book. Here goes the list:

Benenson, I. and P. M. Torrens (2003). Geographic Automata Systems: a new paradigm for integrating GIS and geographic simulation. Association Geographic Information Laboratories Europe (AGILE), Lyons.

Benenson, I., S. Aronovich, et al. (2004). "Let's Talk Objects: Generic Methodology for Urban High-Resolution Simulation." Computers, Environment and Urban Systems, **Forthcoming**.

Torrens, P. M. and I. Benenson (2005). "Geographic Automata Systems." International Journal of Geographic Information Science **forthcoming**.

You can download all these papers from the www.geosimulationbook.com site.

We are very much interested in your critics and suggestions

bennya@post.tau.ac.il, vlad@eslab.tau.ac.il