UFLibrary_v1.0.1 Tutorial
2005-05-28

Mattias Wahde and Jimmy Pettersson
mattias.wahde@chalmers.se, jimmy.pettersson@chalmers.se

1. Introduction to the Utility function (UF) method

The **utility function method** (hereafter: the UF method) [see refs. 1-6] is a method for behavioral organization (behavioral selection) in behavior-based autonomous robots, based on evolutionary optimization. The method provides a general solution to the difficult problem of activating behaviors at appropriate times.

1.1 Behavioral selection in the UF method

The UF method is an **arbitration method**, i.e. a method in which one and only one behavior is active (i.e. controls the robot) at each instant of time. The selection of the active behavior is based on a quantity called **utility**: The active behavior is simply the one with the highest current utility value. In the UF method, each behavior is associated with a **utility function** that determines, for any given state (situation) the utility of the behavior. Note that the UF method does *not* generate the actual behaviors: it merely attempts to solve the problem of *selecting* between the available behaviors.

Clearly, the behavioral selection will thus depend on the exact shape of the utility functions. In very simple cases involving, say, the selection between two behaviors in a robot moving in a simple arena, it is sometimes possible to specify the shape of the utility functions by hand. However, in more complex cases, involving robots equipped with many behaviors, it is all but impossible to specify the utility functions by hand. Thus, in the UF method, the utility functions are determined using an **evolutionary algorithm** (hereafter: EA) which maintains a population of robots, each equipped with a given set of behaviors, which in turn are associated with utility functions. Each robot is evaluated for a certain period of time, in one or several arenas.

As with all EAs, a fitness measure is needed in order to compare the performance of the individuals in the population. Here, another problem presents itself: In general, it is a daunting task to specify a fitness measure for a robot equipped with many different behaviors, since this requires that one be able to weigh different behaviors against each other, for all situations that might occur. By contrast, in the UF method, the specification of the fitness measure is simpler, at least in situations where the robot is given one specific task. Consider, for example, the case of a floor-sweeping robot equipped with two behaviors: *floor-sweeping* (B1) and *battery charging* (B2). Of course, as a user of the robot, one would want the robot to execute B1 (the **task behavior**, using UF nomenclature) continuously. However, a robot carrying its own energy source must, of course, charge its batteries from time to time, i.e. it must also execute the **auxiliary behavior** B2 so that, after charging its batteries, it may again proceed with B1 and thus gain fitness. In the UF method, the user need only specify a fitness measure for the task behavior. By contrast, none of the auxiliary behaviors (of course there is often more than *one* such behavior) have any impact at all on the fitness of the robot. How, then, can the robot sometimes choose B2 over B1, even though the former provides no fitness increase whereas the latter does? Clearly, what is needed is some way of determining at what situations it is beneficial to execute B2. This requires a **common currency** for comparing behaviors, and that is exactly what the utility values (obtained from the utility functions) provide: The utility functions *are* the common currency used for finding, in any given situation, the most useful behavior. Thus, to summarize, it is the user's task to specify the fitness measure (e.g. one point of fitness gained for each square meter of floor sweeped, in the example above), whereas it is the task of the UF method to determine the utility functions (using an EA) that will maximize the fitness of the robot.

1.2 State variables and utility functions

In the UF method, several types of state variables are defined, namely **external variables** (such as e.g. sensor readings), **internal physical variables** (such as e.g. the reading of a sensor measuring the battery level), and **internal abstract variables**. The internal abstract variables are readings of (artificial) **hormone variables**. Variables of this type are, just like hormones in biological systems, used for internal signalling. As will be shown below, the internal abstract variables are often used to prevent dithering, i.e. rapid and counterproductive switching between behaviors. When the UF method is to be applied to a given behavioral organization task, the user must specify an ansatz for each utility function (one for each behavior). While any utility function may, in principle, depend on all of the available state variables, it is common that such functions only depend on a small subset of the state variables. In the present version of the method, the utility functions are p^{th} degree polynomials. Thus, a typical ansatz (with p=2) for, say, a utility function depending on the two variables E and x would be

$$U(E,x) = a_{00} + a_{10}E + a_{01}x + a_{20}E^2 + a_{11}EX + a_{02}\,x^2\,, \qquad (1)$$

where the a_{ij} are the coefficients that are to be determined by the EA.

1.3 A simple example

For most of the remainder of this tutorial, a simple example involving the organization of two behaviors will be considered. The example is indeed very simple, since the aim has been to provide a clear and accessible illustration of how the UF method works, rather than trying to describe a realistic application. Thus, consider the example of a very simple, two-wheeled, differentially steered robot (e.g. a guard robot), whose task it is to move as far as possible along a circular path. The robot, which is illustrated in Fig. 1, is equipped with two behaviors, namely *Circular navigation* (B1) and *Battery charging* (B2).

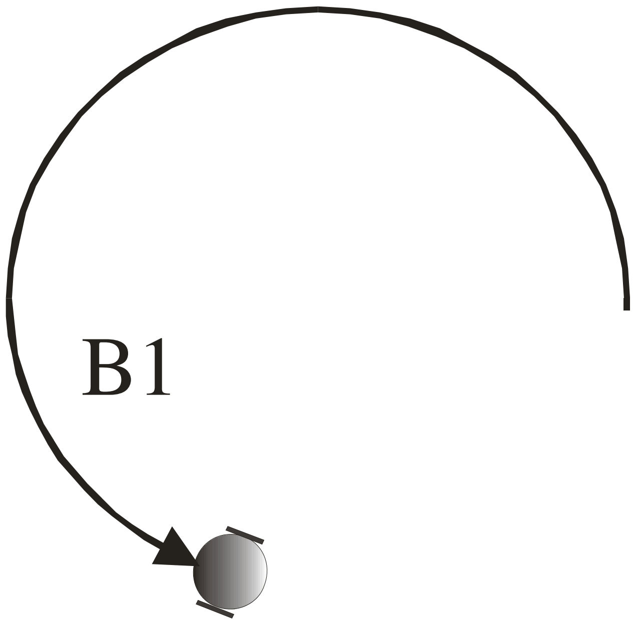


Fig. 1 The motion of the robot (seen from above) while executing B1.

In the first behavior, the robot sets the torque of its two motor to slightly different values, thus executing a circular motion, as illustrated in Fig. 1. In B2, the robot simply sets the torque of the two motors to zero, eventually reaching a standstill. It is assumed that the charging of the batteries (taking place e.g. via a conducting floor) starts after a time t_c after the motor torques have been set to zero, and then continues until B1 is activated. If the batteries become full, no further charging occurs, but the robot does not start to move until B1 is again activated.

The evaluation of the robot continues during a time interval of length T, *unless* the robot runs out of battery energy, in which case the evaluation is terminated immediately. For this robot, B1 is the task behavior, for which a fitness measure should be specified. In this simple case, the distance moved while executing B1 is a suitable fitness measure. However, since the robot consumes energy while executing B1 it must, from time to time also activate B2. In order to do so, it must be able to determine the utility of B1 and of B2, at all times. Clearly, in this simple case, the battery energy E is a very important variable. Thus, for B1, the following ansatz can be made for the utility function: (making the somewhat arbitrary choice p=2 for the polynomial degree)

$$U_1(E) = a_{00} + a_{10}E + a_{20}E^2 \qquad (2)$$

Now, since the selection of a behavior (for activation) depends only on the relative utility values of the available behaviors (i.e. the behavior with the highest utility value is activated), one might be tempted to set $U_2 \equiv 0$ in this case. However, this would lead to a more subtle problem: Assume that the coefficients determining U_1 has been set such that U_1 decreases as the battery energy falls. If U_2 were identically zero, B2 would become active as soon as U_1 dropped below zero, provided that the coefficients were such that it does.

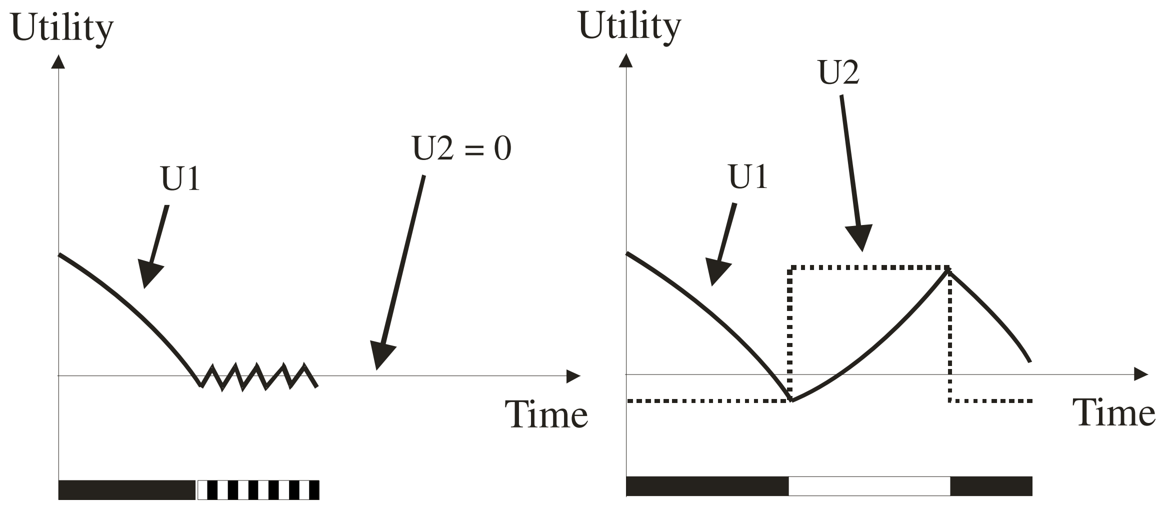


Fig. 2 Utility variation and behavior activation. In the left panel U_2 is identically zero, resulting in behavioral dithering (as indicated by the rectangles at the bottom of the figure: A black rectangle indicates that B1 is active and a white one that B2 is active). By contrast, in the right panel, U_2 depends on an internal abstract variable x.

To be specific, consider the case in which the battery energy is normalized to the range [0,1], and where U_1 is given as $U_1 = -0.5+E^2$, so that B2 would become active at $E = 0.707\ldots$. As soon as B2 becomes active, the battery energy would rise, resulting in U_1 becoming positive, and thus activating B1. However, at this point, E would again fall, thus directly making $U_2 > U_1$, again activating B2 etc. Put differently, the robot would find itself dithering endlessly between B1 and B2. This situation is illustrated in the left panel of Fig. 2. Clearly, a different ansatz is needed for U_2, and this is where the concept of abstract internal variables becomes useful. Consider a variable x that takes the value 1 whenever B2 is active and the value 0 whenever B1 is active[1]. Now, ansatz of the form

$$U_2\,(x) = \mathrm{b}_{00} + \mathrm{b}_{10}x + \mathrm{b}_{20}x^2 \qquad (3)$$

can be made for U_2[2]. Now, as U_2 becomes larger than U_1, as a result of a drop in battery energy, x (and, therefore, U_2) would make an instantaneous jump. Provided that the coefficients in Eq. (3) have been set appropriately, this would make U_2 much larger than U_1, thus keeping B2 active for some time, as illustrated in the right panel of Fig. 2.

[1] The variation of x could be more complex. However, this simple specification will suffice for the present example.

[2] Clearly, the ansatz $U_2 = \mathrm{b}_{10}x$ would be sufficient in this case.. However, the form (2) was chosen since it is more general, i.e. it would be applicable even in cases where x varied in a more complex way.

2. Introduction to UFLibrary_v1.0.1

The UFLibrary, which provides a general implementation of the UF method, has been written in Delphi object-oriented Pascal, and is provided, as the name implies, in the form of a program library rather than a stand-alone application. The rationale behind this approach is that, since each new scientific investigation generally differs significantly from previous ones, a stand-alone, non-modifiable application will in general be less useful than a library. The drawback, of course, is that the use of a library will require some programming on the part of the user.

The aim with UFLibrary has been to place all common structures (such as the mechanisms of behavioral selection) in the library, and thus to minimize the effort needed from the user. Nevertheless, the problem of behavioral organization is a complex one, and the UFLibrary is therefore far from trivial to use. Thus, in this tutorial, a simple step-by step example will be given, involving the organization of the two simple behaviors (B1 and B2) described above.

2.1 General usage issues

Even though the UFLibrary simplifies the problem of behavioral selection, there are several steps that must be carried out in any given behavioral organization task. At a first glance, the procedure described below may seem rather complex. However, in several cases (namely steps (1), (3), (5), and (7)), pre-defined templates can be used, thus minimizing the effort required by the user.

Specifically, the user must

(1) Provide, in the form of source files written in Pascal, the constituent behaviors that are to be included in the **behavioral repetoire** of the robot. The specification of a behavior can usually be based on a predefined template, thus limiting the work to writing a few procedures. Of course, it is also possible to use the baseic pre-specified behaviors provided in the UFLibrary.

(2) Specify a fitness measure. As described above, often the execution of a single behavior (the task behavior) will be associated with a fitness increase, the fitness variation for all other behaviors being zero.

(3) Provide a **robot definition file** (specifying the physical parameters of the robot, as well as its sensors and motors) and a **brain definition file** (specifying which behaviors are to be included in the behavioral repertoire, as well as any user-defined parameters included in the behaviors). Examples of such definition files will be given below.

(4) Provide an ansatz for each utility function. Again, the UFLibrary does most of the work: The user need only specify which variables are to be included in each polynomial, as well as the polynomial degree p.

(5) Provide an **arena definition file**, specifying the location of walls, obstacles, etc. in the arena in which the robot will operate. An example of an arena definition file will be given below.

(6) Specify a termination criterion, e.g. terminating an evaluation in case of collisions (collision handling is taken care of by UFLibrary, limiting the specification of this particular termination criterion to a few lines of code).

(7) Write and compile the actual application (executable file). Also in this case, the user may employ pre-defined source files, as illustrated in the example below. The user has full flexibility to specify e.g. what data should be extracted from the evaluation of each individual (e.g. the variation of utility with time, the fitness of the robot, the motion of the robot etc.)

3 A simple example: Circular navigation

3.1 Problem description

Consider again the example introduced above, namely that of an autonomous robot moving in a large, obstacle-free arena. Assume that the task of the robot is simply to guard the area around the center of the arena, by moving in circles around it, and assume further that the robot carries its own energy supply (e.g. a battery) that must, from time to time, be charged. In this simple example, it will be assumed that the robot is able to charge it batteries simply by stopping (e.g. by making use of a conducting floor) As mentioned above, it will be assumed that charging starts only after t_c seconds of execution of B2 (the battery charging behavior).

The robot will be equipped with two behaviors: *CircularNavigation* (B1) and *BatteryCharging* (B2). Since the UF method is an **arbitration method**, only one behavior can be active at any given time. Thus, the robot must find a way to select between the two available behaviors. Its goal is to move a far as possible in a given time. Since there is a delay in the activation of the charging, and since the robot only needs to stop, in any, location, to begin charging behavioral selection is quite simple in this case: A rational approach, if the robot starts with a full battery, is simply to move until the battery is almost empty, and then to stop and charge the batteries until they are full (the possible exception being the final part of the evaluation, where it might be beneficial for the robot only to charge the batteries partially).

In the particular case considered here, a hormone variable measuring the equivalent of hunger (i.e. lack of energy) will be introduced in such a way that it forces the robot to keep B2 active for some time before B1 can again become active. Thus, the ansatz given in Eq. (3) will be used for U_2, with x representing the reading of the hormone variable H. For U_1, the ansatz in Eq. (2) will be used.

Thus, the task of the EA will be to find appropriate values for the coefficients a_{ij} and b_{ij} in order to activate behaviors in a timely fashion. Since x is an artificial variable, i.e. a variable that is unrelated to physically measurable quantities, its variation with time

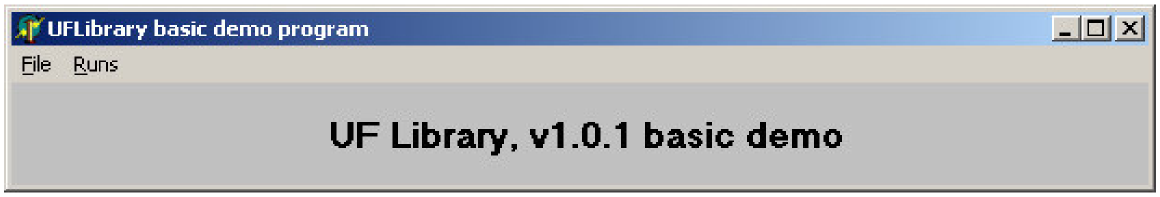


Fig. 3 The main window of the UFLibBasicDemo application.

(and other variables) must be specified as well. In UFLibrary_v1.0.1, it is the user's task to specify this variation, and it should therefore be kept as simple as possible (In future versions of the UF method and thus of UFLibrary, the variation of hormone variables will be evolved together with the utility functions). Here, the variation of *H* (and thus of *x*) will be as specified in Sect. 1 above: *H* is exactly 0 whenever B1 is active, and exactly 1 whenever B2 is active. The fitness measure will be taken simply as *the distance moved while executing B1*.

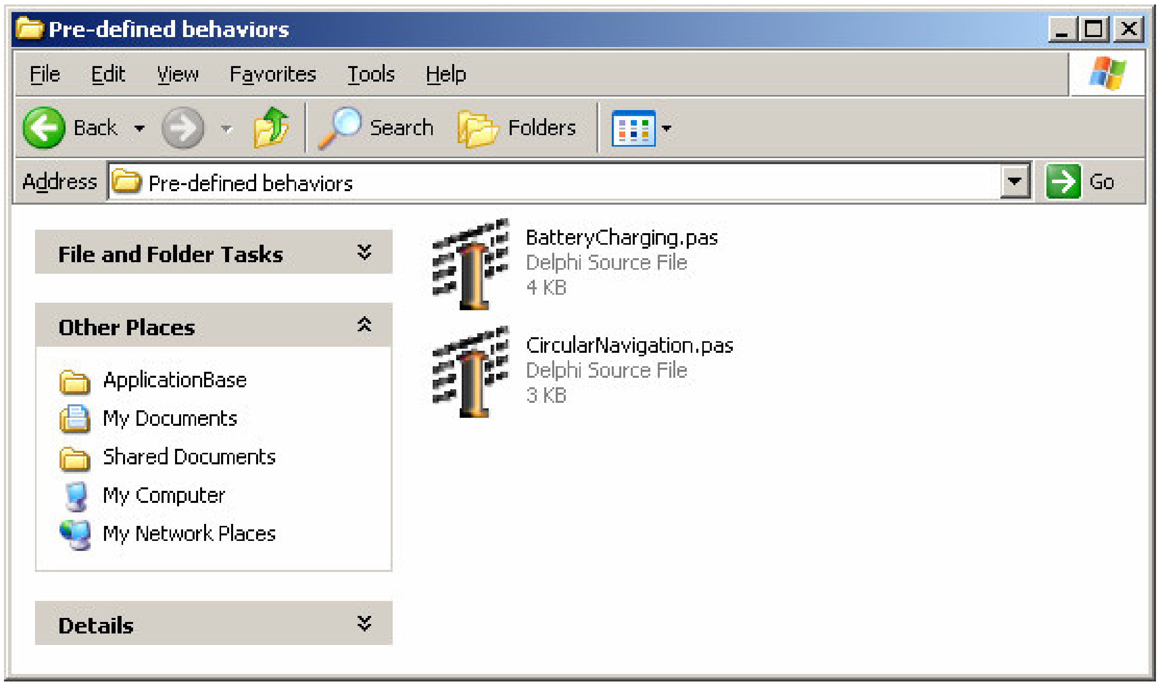
In Sect. 3.3 below, source code for the example described above will be analyzed in detail. However, before the writing of source code is considered, a simple illustration will be given of the compilation and execution of a program based on the UFLibrary.

3.1.1 A note concerning Delphi versions

In order for it to be possible to use the UFLibrary, the user must, of course, have Delphi object-oriented Pascal installed on the computer. Furthermore, the user must be careful to use the correct version (depending on the Delphi version used) of the UFLibrary. The examples associated with this tutorial are provided for Delphi v.5 and Delphi v.6. Names of folders files will be given in ***bold italics*** below, whereas names of files and will be given in **bold face**. Menu selections in Delphi and in the example program will be given using <u>underline</u>.

3.2 Compiling and running the program

The source code for the application and the two behaviors *CircularNavigation* and *BatteryCharging* is provided with this tutorial. As a first step, the addition of behaviors to an application, as well as the compilation of the program, will be illustrated. First, open the folder ***BasicDemo***, and double-click on the blue Delphi project icon, named **UFLibBasicDemoProject.dpr** in order to start the project. The window shown in Fig. 3 should appear. By default, three **Delphi units** (source files) have been added to this project, namely the files **Main.pas**, **UFEvolution.pas**, and **BasicRobotSimulation.pas**. By selecting <u>View</u> – <u>Project Manager</u>, and clicking on the + symbol the user may view the files available in the project.



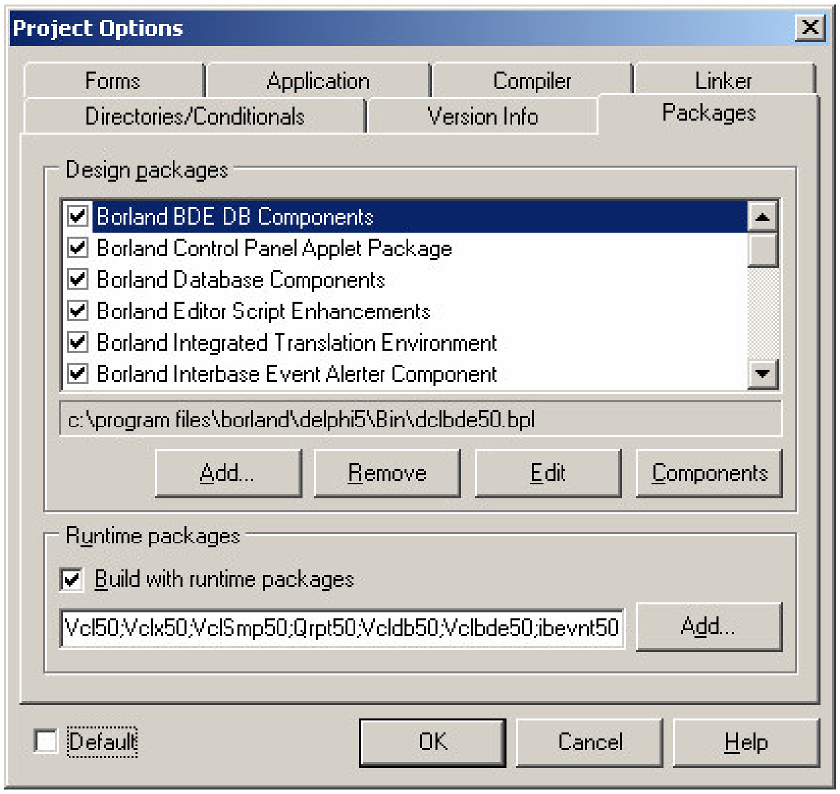
BatteryCharging.pas
Delphi Source File
4 KB

CircularNavigation.pas
Delphi Source File
3 KB

Fig. 4 The *Pre-defined behaviors* folder

Now, select Project – Add to Project, and browse to the folder ***Pre-defined behaviors***, as shown in Fig. 4 below. Next, add the two behavior units **CircularNavigation.pas** and **BatteryCharging.pas** to the project.

The next step is to compile the project. For this to be possible, it is necessary to tell Delphi where it can find the **UFLibrary**. The UFLibrary is contained in two files called **UFLibrary*xx*.dcp** (this is the Delphi compiled package, needed for the compilation) and **UFLibrary*xx*.bpl** (this is the file needed when running the application, corresponding to a *dll*, using standard Windows terminology). ***xx*** indicates the version number. Thus, in Delphi 5, ***xx*** = 50 etc. Now, select Project – Options… in the main Delphi window, and then select Packages. The window shown in Fig. 5 appears (Note: the figure shows the appearance of the Project – Options… window for Delphi 5. For other versions of Delphi, it will look slightly different). Check the Build with runtime packages check box. Select Add, and browse to the location of the UFLibrary (i.e the folder immediately above ***Pre-Defined behaviors***, see Fig. 6). Click Open and then OK (twice). Now the application should be complete. In order to compile it, press F9 or select Run – Run in the Delphi main window. The application starts by showing the main window. In the Runs menu, select Evolution of behavioral organizer. Now, the window shown in Fig. 7 appears. Press Initialize and then Run, without modifying any of the parameter values. The program will now run the evolutionary algorithm, evaluating successive generations, each of which consists of 50 individuals evaluated for 50 seconds (or until a collision occurs).


Project Options

Design packages

- [✓] Borland BDE DB Components
- [✓] Borland Control Panel Applet Package
- [✓] Borland Database Components
- [✓] Borland Editor Script Enhancements
- [✓] Borland Integrated Translation Environment
- [✓] Borland Interbase Event Alerter Component

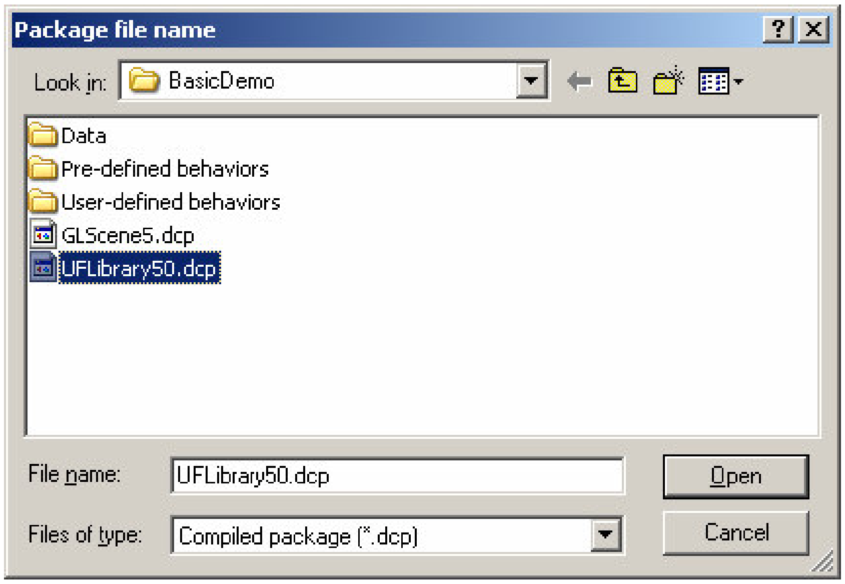
c:\program files\borland\delphi5\Bin\dclbde50.bpl

Runtime packages

[✓] Build with runtime packages

Vcl50;Vclx50;VclSmp50;Qrpt50;Vcldb50;Vclbde50;ibevnt50

Fig. 5. The <u>Project – Options…</u> window, shown for Delphi 5.


Package file name

Look in: BasicDemo

- Data
- Pre-defined behaviors
- User-defined behaviors
- GLScene5.dcp
- UFLibrary50.dcp

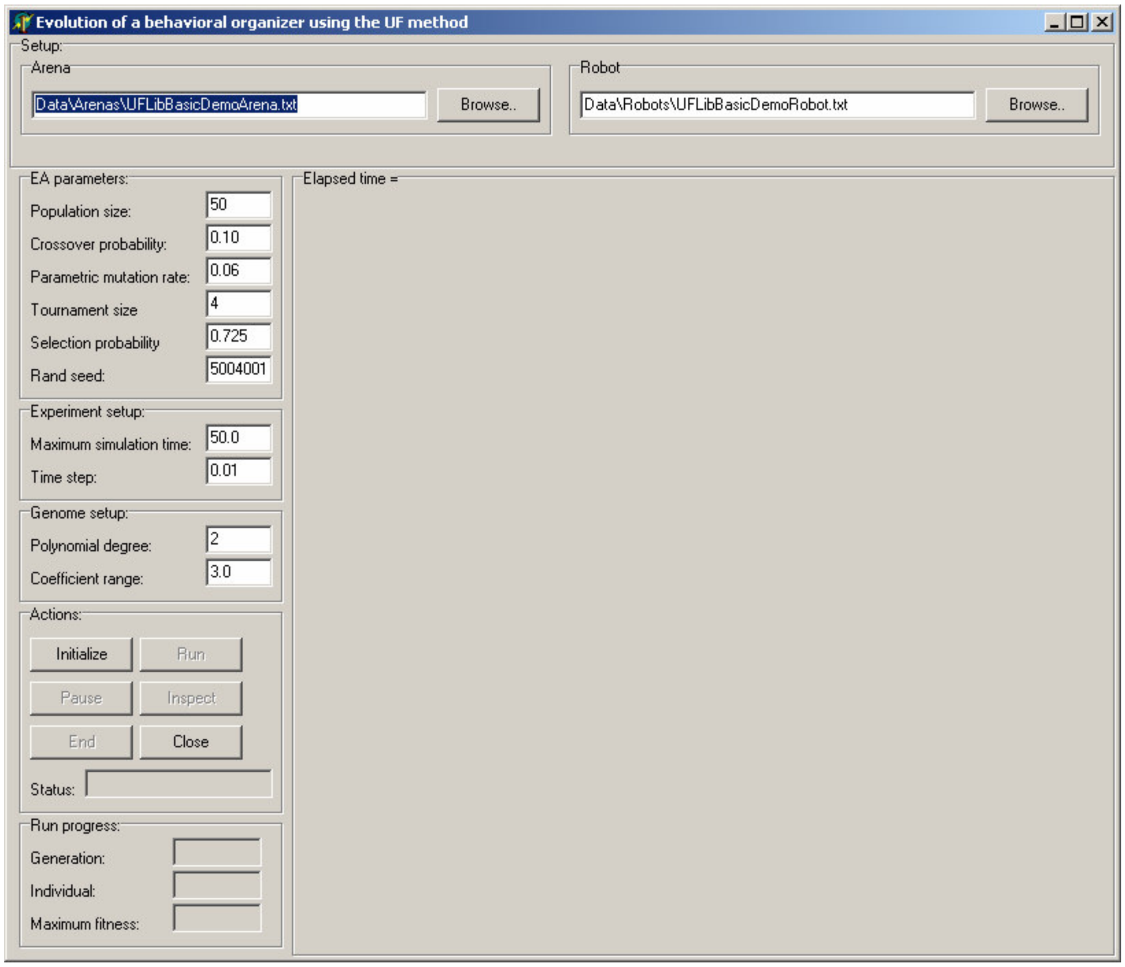
File name: UFLibrary50.dcp

Files of type: Compiled package (*.dcp)

Fig. 6. Adding the UFLibrary dcp file (in the case Delphi5).

Let the program run until the maximum fitness value (shown in the lower left corner of the window) reaches 10 or more. This should take less than 30 seconds on a computer equipped with a P4 processor with a clock frequency of 2.5GHz or above. Then press Pause. Once the current generation has been completed, the program pauses. When this has occurred, press the Inspect button. Now, the robot can be seen in action, switching between a circular motion (in B1) and standing still to charge the batteries (in B2). A typical screenshot from such a run is shown in Fig. 8.

Now, press End, and then Close. Exit the program by selection File – Exit in the main window, and then inspect the ***BasicDemo*** folder. As can be seen, a new folder, named after the exact time at which the program was started, has been added. Open this folder. It should contain two files, namely an evaluation file called **Robot_Gen_*n*.txt** (where *n* is the generation after which the Inspect button was pressed), and a run summary called **Run_Results_Summary_YYYY_MMMDD_HH_NN_SS.txt**, where the string YYYY_MMMDD_HH_NN_SS indicate the date and time at which the run was started.


Evolution of a behavioral organizer using the UF method

Setup:
Arena: Data\Arenas\UFLibBasicDemoArena.txt
Robot: Data\Robots\UFLibBasicDemoRobot.txt

EA parameters:
Population size: 50
Crossover probability: 0.10
Parametric mutation rate: 0.06
Tournament size: 4
Selection probability: 0.725
Rand seed: 5004001

Experiment setup:
Maximum simulation time: 50.0
Time step: 0.01

Genome setup:
Polynomial degree: 2
Coefficient range: 3.0

Actions:
Status:

Run progress:
Generation:
Individual:
Maximum fitness:

Elapsed time =

Fig. 7. The window used when evolving the behavioral organizer.

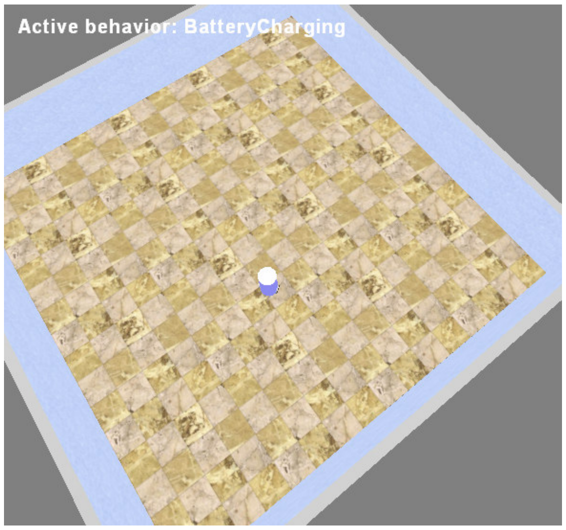


Fig. 8. A screenshot showing the robot in the arena, at the end of an evaluation.

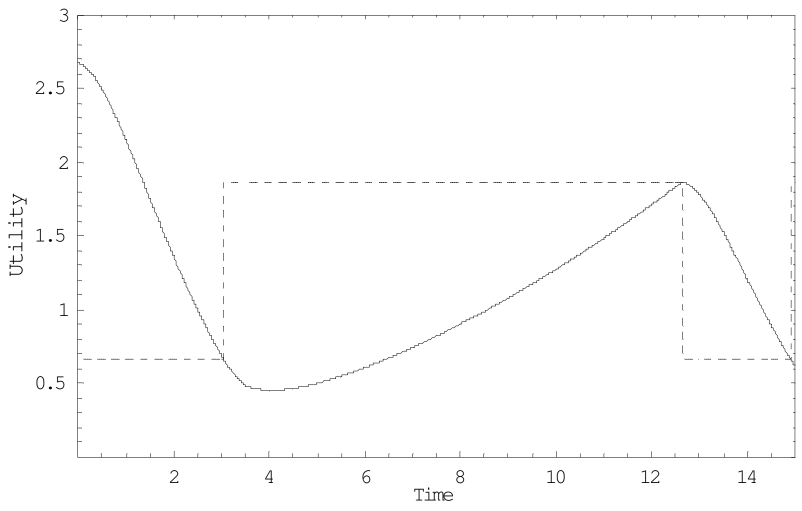


Fig. 9. The actual utility functions obtained using the UFBaseApplication executable. The solid line shows U_1, and the dashed line represents U_2.

The latter file contains a list of the run parameters, as well as the maximum and average fitness values measured during the run. The file **Robot_Gen_*n*.txt** contains the evolved utility functions, as well as their variation with time, the position and angle of heading of the robot (as functions of time), and the battery energy (also as a function of time). In Fig. 9, the actual utility functions obtained for an individual with fitness > 10. In the figure, U_1 and U_2 are shown as solid and dashed lines, respectively. Initially, B1 is active, since $U_1 > U_2$. At around t = 3.0, U_1 dips below U_2, so that B2 becomes active. The internal abstract variable x is then set to 1, resulting in an upward jump in U_2. The charging of the battery begins after a short delay t_c, and U_1 then begins to rise until, at around t=12.5, it again exceeds U_2, thus activating B1.

3.3 Writing behavior source files and definition files

The program will now be studied in greater detail, by going through the source units for the behaviors, and then analyzing the definition files. Thus, as the next step, select Project – Remove from Project…, and remove the two source files **CircularNavigation.pas** and **BatteryCharging.pas** from the project. Next, open the folder ***BasicDemo***. As discussed above, in this folder, there are three source files (all of which carry the suffix .pas), namely **Main.pas**, **BasicRobotSimulation.pas**, and **UFEvolution.pas**. **Main.pas** and **UFEvolution.pas** are each associated with a Delphi form (window), contained in the files **Main.dfm** and **UFEvolution.dfm**, respectively. In addition, the folder also contains the UFLibrary (as two files, **UFLibrary*xx*.bpl** and **UFLibrary*xx*.dcp**), GLScene (for 3D graphics, contained in the files **GLScene*n*.bpl** and **GLScene*n*.dcp**), where *xx* and *n* indicate the Delphi version used (for e.g. Delphi 5, ***xx*** = 50, and ***n*** = 5). The folder also contains the two files **vcl50.bpl** and **vcljpg50.bpl**, which are needed by the UFLibrary (but which are normally already present on the computer, provided that Delphi is installed).

Note: None of the three .pas files mentioned above are *necessary* for using the UFLibrary. They are provided as templates for rapid application development, but can easily be replaced by files generated by the user. For example, there is, in principle, no need to generate a separate form for running the evolutionary algorithm: This could be done directly under the main.dfm window, should the user wish to do so. Next, take some time to study the three .pas files:

(1) **Main.pas**

This is the main window of the application. All it does is to open a form of type `TUFEvolutionForm`, in which the actual evolution takes place.

(2) **UFEvolution.pas**

This file contains several procedures. Note, however, that there is no need to look at these procedures in detail at this point, and neither is it necessary to modify any of them. In fact, the **UFEvolution.pas** unit can be used with any UF experiment. However, if the user should generate a custom class for robot simulation with a different name than `TBasicRobotSimulation` (see below), the corresponding changes must of course be made also in **UFEvolution.pas** (i.e. all occurences of `TBasicRobotSimulation` must be modified to `T<name>RobotSimulation`, where `<name>` is the custom name provided by the user).

Anyway, consider briefly the most important procedure in this source file, namely the **RunButtonClick** procedure. This procedure contains the main loop used by the evolutionary algorithm (EA). Basically, the EA loops over all individuals in the population, generates a robot simulation and sets the corresponding arena (`SetArena(fArena);` For a description of arena files, see below), decodes the chromosome of the current individual (`fEA.Population[i].DecodeGenome`), sends the individual to the robot simulation (`SetAgent(fEA.Population[i]);`), sets up a collision manager that handles collisions between the robot and objects in the arena, and then evaluates the individual, time step by time step, until the simulation is complete, i.e. until either the alloted time has been used up, or the robot collides with an obstacle. When all individuals have been evaluated, a new generation is formed (`fEA.MakeNewGeneration`).

(3) **BasicRobotSimulation.pas**

This unit defines a class `TBasicRobotSimulation`, which is derived from the class `TRobotSimulation`, defined in the UFLibrary package. The `TRobotSimulation` class contains three virtual procedures, i.e. procedures that can be modified in a descendant class derived from it. In the base class `TRobotSimulation`, there are two termination criteria: the evaluation of the robot (i.e. the simulation is terminated if the maximum time is reached or if the robot collides with an obstacle. However, in the example considered here, the evaluation should also be terminated if the battery energy level drops to 0. Thus, a new class, i.e. `TBasicRobotSimulation`, is needed. For clarity, all three termination criteria have been included in the **CheckTerminationCriteria** procedure in this class. An alternative way would have been to replace the first two criteria with a call to the procedure inherited from the `TRobotSimulation` class, i.e. by writing the procedure as

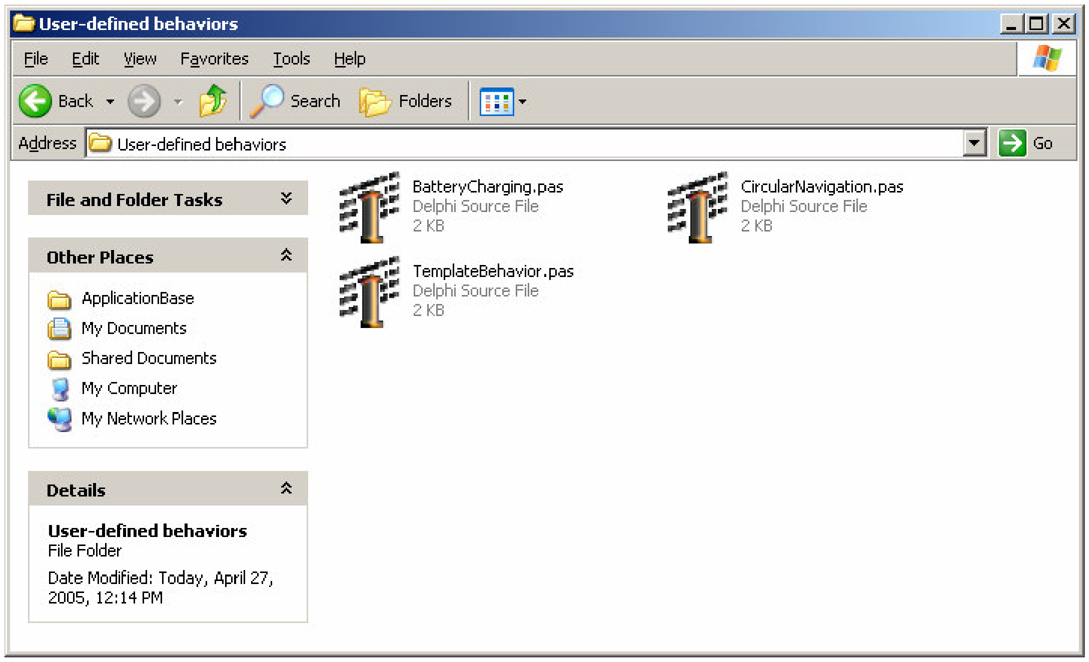
```
procedure TBasicRobotSimulation.CheckTerminationCriteria;
begin
  inherited;
  if (TRobot(fAgent).Body.Battery.Level <= 0.0) then
    begin
     fIsComplete := True;
     fTerminationType := ttBatteryDepleted;
    end;
end;
```

The procedure `UpdateFitness`, which does nothing in the base class, can be used in cases where one wishes to modify the fitness of the individual continuously during an evaluation. This is not the case here, so the procedure is empty also in the derived class. Finally, in the base class, the procedure `FinalizeFitness` sets the fitness simply to the distance travelled by the robot. This is a suitable fitness measure also for the example considered here (since the robot moves actively only when executing B1), and therefore the procedure in the descendant class simply calls the corresponding procedure in the base class.

The next steps are (1) to generate the source code for the actual behaviors, and then (2) to specify the general structure of the robotic body and brain, as well as the arena, in the form of text files.

3.3.1 Writing behaviors

The base class from which all specific behavior classes are derived is the TBehavior class. In order to write a specific behavior, the user must, as a minimum, provide source code for the `Step` procedure, which determines the actions take by the robot while executing a single time step using the behavior in question. In addition, source code for the constructors `Create` and `CreateAndSet`, the function `Copy`, the procedures `LoadFromDefinition`, and (in certain cases) the destructor `Destroy`, must be provided. Furthermore, for some behaviors (e.g. the battery charging behavior, see below), the procedures `Enter`, `Initialize`, and `Exit` must be provided as well However, as we shall see, only a few lines of code are needed for each routine, at least in this simple example.



User-defined behaviors
File Folder
Date Modified: Today, April 27, 2005, 12:14 PM

BatteryCharging.pas
Delphi Source File
2 KB

CircularNavigation.pas
Delphi Source File
2 KB

TemplateBehavior.pas
Delphi Source File
2 KB

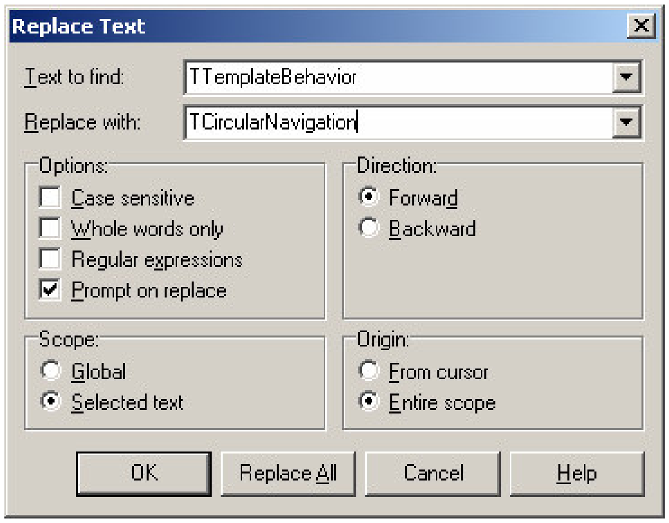
Fig. 10. The folder *User-defined behaviors*, after making copies of TemplateBehavior.pas.

Now, Use the explorer to open the folder ***User-defined behaviors***. Here, a template for behaviors is provided in the source file **TemplateBehavior.pas**. Make two copies of this template file, and rename them to **CircularNavigation.pas** and **BatteryCharging.pas**. Thus, after this step, the folder ***User-defined behaviors*** should look as in Fig. 10 above. Next, open one of the files, e.g. **CircularNavigation.pas**, using the File – Open command in Delphi. Since the file is a direct copy of **TemplateBehavior.pas**, the name of the unit is still `TemplateBehavior`. Thus, as a first step, change the name of the unit to `CircularNavigation`, so that the first line reads

```
unit CircularNavigation;
```

The class implementing the circular navigation behavior should preferably be called `TCircularNavigation`. Thus, change all occurrences of the word `TTemplateBehavior` to `TCircularNavigation`. The easiest way of doing so it to select Search – Replace in Delphi, specifying the change as shown in Fig. 11 below, and pressing Replace All.

In the template file, the defined class is of type TBehavior. In the present case we are about to write a specific type of behavior, namely a motor behavior, for which a class (derived from TBehavior) is available, namely TMotorBehavior. This class differs from TBehavior only in one respect: it contains a vector for storing the motor outputs generated by the behavior. Our behavior for circular navigation will be derived from TMotorBehavior. Thus, change the class header to read


Replace Text

Text to find: TTemplateBehavior

Replace with: TCircularNavigation

Options: [ ] Case sensitive, [ ] Whole words only, [ ] Regular expressions, [✓] Prompt on replace

Direction: [● Forward] ○ Backward

Scope: ○ Global [● Selected text]

Origin: ○ From cursor [● Entire scope]

Fig. 11. Replacing all occurences of `TTemplateBehavior` by `TCircularNavigation`.

```
type TCircularNavigation = class(TMotorBehavior)
```

Inspecting the interface part of the class TCircularNavigation, we find (as indicated above) that six routines must be written:

```
type TCircularNavigation = class(TMotorBehavior)
 private
 public
  constructor Create; override;
  constructor CreateAndSet(B: TBehavior); override;
  function Copy: TBehavior; override;
  procedure LoadFromDefinition(ObjDef: TObjectDefinition); override;
  procedure Step(TimeStep: real); override;
  destructor Destroy; override;
 end;
```

Let us now write these six routines. The `Copy` function is already complete as it is: the only change needed is to set the right class name, which was just done (see above). The constructor `Create` must set the number of output variables used by the motors of the robot, since the base class `TBehavior` (and thus `TMotorBehavior`) has no way of knowing exactly how many motors are present in the robot under study. In this case we shall use a differentially steered, two-wheeled robot with two motors. Thus, add a line to the constructor `Create` so that it reads

```
constructor TCircularNavigation.Create;
begin
  inherited;
  fOutputVariables.SetSize(2); // added line!
end;
```

The part "// added line" is a comment that can, of course, be omitted. `fOutputVariables` is a field in the base class `TMotorBehavior`, of type `TVector`, for which the `SetSize` procedure is defined. Thus, the line just added sets the number of output variables to two.

Next, the `Step` procedure should be written. This is the procedure that determines the actual behavior of the robot, and it will, of course, vary from behavior to behavior. In connection with the writing of the `Step` procedure, it is common that one must define local variables needed for the implementation of the behavior. These can range from simple constants to complex objects. For the particular case of the circular navigation behavior, only two motor outputs are needed: one for the left motor (M_L) and one for the right motor (M_R). If the motor outputs are set to constant values, the robot will execute a clockwise motion if $M_L > M_R$ and a counterclockwise motion otherwise (shown in Fig. 1). Clearly, most behaviors will be made more advanced than this, but for the present example, two constant motor outputs will be sufficient.

Note: the motor outputs will be fed to DC motors (implemented in the class `TDCMotor` in UFLibrary), which expect signals in the range [-1, 1]. The procedure for actually setting the numerical values, through the text files defining the robot, will be described below).

Thus, as the next step, add two fields of type `real`, `fLeftMotorOutput` and `fRightMotorOutput`, to the interface of the `TCircularNavigation` class, so that it reads:

```
type TCircularNavigation = class(TMotorBehavior)
 private
  fLeftMotorOutput: real;
  fRightMotorOutput: real;
 public
```

```
  constructor Create; override;
  constructor CreateAndSet(B: TBehavior); override;
  function Copy: TBehavior; override;
  procedure LoadFromDefinition(ObjDef: TObjectDefinition); override;
  procedure Step(TimeStep: real); override;
  procedure Exit; override;
  destructor Destroy; override;
  property LeftMotorOutput: real read fLeftMotorOutput;
  property RightMotorOutput: real read fRightMotorOutput;
 end;
```

Note the addition of the properties at the end of the class interface: The fields for the motor outputs, `fLeftMotorOutput` and `fRightMotorOutput` will be invisible outside the unit `CircularNavigation` (since they are defined as `private`). However, the two properties `LeftMotorOutput` and `RightMotorOutput` can be read (but not written to) by any unit containing `CircularNavigation` in its **uses** clause. Next, proceed to the implementation part of the `Step` procedure, and modify it to read:

```
procedure TCircularNavigation.Step(TimeStep: real);
begin
  inherited;
  fOutputVariables[1] := fLeftMotorOutput;
  fOutputVariables[2] := fRightMotorOutput;
end;
```

This `Step` procedure will set the left and right motor outputs to constant values. In this simple case, it would have been possible to set the constant motor outputs in the `Enter` procedure, which is defined in the base class `TBehavior`. However, the preferred way of setting torque values (even constant ones), is to use the `Step` procedure.

The `CreateAndSet` constructor can now be written. Its purpose is to generate an exact copy of the behavior, which is needed e.g. when creating new individuals in the evolutionary algorithm used for optimizing the utility functions. In this case, there are two fields defined in the behavior, namely the motor outpus `fLeftMotorOutput` and `fRightMotorOutput`. Thus, `CreateAndSet` takes the form:

```
constructor TCircularNavigation.CreateAndSet(B: TBehavior);

var
  CB: TCircularNavigation;

begin
  inherited;
  CB := TCircularNavigation(B);
  fLeftMotorOutput := CB.LeftMotorOutput;
  fRightMotorOutput := CB.RightMotorOutput;
end;
```

Note that an explicit typecast must be used, since the input variable (`B`) is of type `TBehavior`. The next step is to write the procedure `LoadFromDefinition`, which is used when reading the parameters of a behavior (e.g. the parameters `LeftMotorOutput` and `RightMotorOutput` in this case) from the text file defining the brain of the robot (see below). The general nomenclature for reading a field in a `LoadFromDefinition` procedure is

```
<fieldname> := ObjDef.PropertyAsFloat('Name of field in text file');
```

Similar procedures are defined for reading variables in the form of integers (`PropertyAsInteger`), strings (`PropertyAsStrings`), and vectors (`PropertyToVector`). When the file is read, the field in question in identified by its name as written in the text file (Example are shown below). Thus, in principle, any name or descriptive string (without spaces) can be used for identifying a field. The preferred way, however, is to use the name of the corresponding property, as defined in the class interface. In this case, the `LoadFromDefinition` procedure will take the form:

```
procedure TCircularNavigation.LoadFromDefinition(ObjDef: TObjectDefinition);
begin
 inherited;
 fLeftMotorOutput := ObjDef.PropertyAsFloat('LeftMotorOutput');
 fRightMotorOutput := ObjDef.PropertyAsFloat('RightMotorOutput');
end;
```

Finally, the destructor `Destroy` should be written. In this case, the only fields that need to be destroyed are `fLeftMotorOutput` and `fRightMotorOutput`, which are of type `real` and are thus handled by the default destructor associated with all Delphi objects. Thus, the destructor need not be changed. The behavior *CircularNavigation* is now complete. Save the file, and take a moment to check that your code looks *exactly* as the source code for *CircularNavigation* provided in the folder ***Pre-defined behaviors***.

Next, the source code for the battery charging behavior should be written. Thus, open the file **BatteryCharging.pas** located in the **user-defined behaviors** folder (see Fig. 10 above). If this file does not yet exist, simply make a copy of **TemplateBehavior.pas**, and rename it to **BatteryCharging.pas**. The class implementing the battery charging behavior should be called `TBatteryCharging`. Thus, change all occurrences of the word `TTemplateBehavior` to `TBatteryCharging`, using the method shown in Fig. 11 above. Also, change the first line of the unit to read

```
unit BatteryCharging;
```

The battery charging behavior will also make use of the motors, and should therefore be a motor behavior. Thus, change the first line of the class interface to read

```
type TBatteryCharging = class(TMotorBehavior)
```

Furthermore, the battery charging behavior will make use of some Delphi units (contained in the UFLibrary) which are not listed in the **`uses`** clause of the template behavior. Thus, modify the **`uses`** clause to read

```
uses
  Definitions
  , Behaviors
  , ObjectDefinition
  , Variables
  , Motors
  , Matrices
  , EnergySource
  , Sensors
  , Hormones
  , BatterySensor
  ;
```

(The four last units have been added).

The six routines `Create`, `CreateAndSet`, `Copy`, `LoadFromDefinition`, `Step`, and `Destroy` should now be written. Just as in the case of the *CircularNavigation* behavior, the `Copy` procedure needs no further change. The constructor `Create` is identical to the one used in *CircularNavigation*:

```
constructor TBatteryCharging.Create;
begin
  inherited;
  fOutputVariables.SetSize(2);
end;
```

In the battery charging behavior, the robot need only stop in order to begin charging the battery, and charging will begin a certain time after activation of the behavior. However, the charging behavior must also be given the information needed to identify the battery and its corresponding sensor (measuring the energy level in the battery). Thus, modify the interface part of the class definition to read

```
type TBatteryCharging = class(TMotorBehavior)
 private
  fBatteryPointer: TEnergySource;
  fCorrespondingBatteryName: string;
  fChargingStartTime: real;
 public
  constructor Create; override;
  constructor CreateAndSet(B: TBehavior); override;
  function Copy: TBehavior; override;
  procedure LoadFromDefinition(ObjDef: TObjectDefinition); override;
  procedure Initialize(Sensors: TSensors; Hormones: THormones); override;
  procedure Enter; override;
  procedure Exit; override;
  procedure Step(TimeStep: real); override;
  destructor Destroy; override;
  property BatteryPointer: TEnergySource read fBatteryPointer;
  property CorrespondingBatteryName: string read fCorrespondingBatteryName;
  property ChargingStartTime: real read fChargingStartTime;
 end;
```

Note that the battery charging behavior is slightly more complex than the navigation behavior, in that it contains three additional procedures: `Initialize`, `Enter`, and `Exit`. The `Initialize` procedure is called (from `TBrain`) once and for all when the robotic brain is created. Its task is to map sensor and hormone variables onto state variables (i.e. the scalar variables used in the utility functions) and input variables (i.e. variables used e.g. in the `Step` procedure of each behavior). The `Enter` procedure, by contrast, is called *every* time the corresponding behavior becomes active, and the `Exit` procedure is called *every* time an active behavior is de-activated. Here, the purpose of `Initialize` procedure is to map the `fBatteryPointer` field to the battery sensor. Thus:

```
procedure TBatteryCharging.Initialize(Sensors: TSensors; Hormones: THormones);
override;

var
  i: integer;

begin
  inherited;
  fBatteryPointer := nil;
  for i := 1 to Sensors.NumberOfSensors do
  begin
    if (Sensors[i] is TBatterySensor) then
    begin
      if (TBatterySensor(Sensors[i]).CorrespondingBatteryName =
        fCorrespondingBatteryName) then
      begin
        fBatteryPointer := TBatterySensor(Sensors[i]).CorrespondingBattery;
        Break;
      end;
    end;
  end;
end;
```

As a precaution, the `Initialize` procedure begins by setting the `fBatteryPointer` to **`nil`**. Next, it loops through all sensors (even though, in the present case, there is only one sensor available, namely the battery sensor), identifies the battery sensors via its name, and sets the corresponding pointer.

In many cases, e.g. the circular navigation behavior, there is no need to add explicitly any of the three procedures `Initialize`, `Enter`, and `Exit`. However, the battery charging behavior is associated with a hormone variable ("hunger"), whose value must be set. In future versions of UFLibrary, the variation of hormone variables will be evolved together with the utility functions. However, in the current version, hormone variables must be set by hand. In the simple application considered here, the hunger hormone will be set to 1 upon activation of *BatteryCharging* and to 0 upon de-activation of the same behavior. In the navigation behavior, the hunger hormone will remain at 0 at all times. The value of the hormone will be made available to the utility function of B2 in the form of an internal abstract state variable (see Sect. 1.2 above), which must thus be present in the definition file of the robotic brain. Here, it will be assumed that the first state variable in B2 corresponds to the hunger hormone. Thus, the `Enter` and `Exit` procedures take the form:

```
procedure TBatteryCharging.Enter;

begin
  inherited;
  TInternalAbstractVariable(fStateVariables[1]).HormonePointer.Level := 1.0;
end;

procedure TBatteryCharging.Exit;

begin
  inherited;
  TInternalAbstractVariable(fStateVariables[1]).HormonePointer.Level := 0.0;
end;
```

Note the typecast, which identifies state variable 1 as an internal abstract variable. Note also that the definition file for the robotic brain cannot be written completely independently of the battery charging behavior, since the latter explicitly identifies the first state variable as an internal abstract variable measuring the level of a hormone (In furture versions of UFLibrary, state variables will be identified by name rather than their index). Now, the `Step` procedure can be written. Put simply, this procedure will stop the motors of the robot, so that charging can begin

```
procedure TBatteryCharging.Step(TimeStep: real);
begin
  inherited;
  fOutputVariables[1] := 0.0;
  fOutputVariables[2] := 0.0;
  if (fBehaviorTime > fChargingStartTime) then
   begin
    if fBatteryPointer <> nil then
     begin
      fBatteryPointer.Charge(TimeStep);
     end;
   end;
end;
```

As is evident from the source code for the `Step` procedure, charging only begins when the behavior time (which is updated by the `Step` procedure in the base class `TBehavior`) exceeds `fChargingStartTime`[3]. Note that the *discharging* of the battery is taken care of by the `Move` procedure in the class `TDifferentiallySteeredBody` (which, in turn, is derived from the base class `TBody`), and consists of two parts: a discharge rate at rest, and a speed-dependent discharge rate. Clearly, in order for the battery to be charged by the charging behavior, the charge rate must exceed the discharge rate at rest. All charging rates are defined in the definition file for the body of the robot, see below. Now, the `CreateAndSet` and `LoadFromDefinition` procedures can be written:

```
constructor TBatteryCharging.CreateAndSet(B: TBehavior);

var
 BC: TBatteryCharging;

begin
  inherited;
  BC := TBatteryCharging(B);
```

[3] The behavior time is set to zero each time the behavior is activated.

```
  fCorrespondingBatteryName := BC.CorrespondingBatteryName;
  fChargingStartTime := BC.ChargingStartTime;
end;
```

Note that the `fBatteryPointer` is set by the `Initialize` procedure. This is so, since, during the creation of a robot, the body must first be initialized in order to identify the sensors, implying that the `fBatteryPointer` cannot be set already at the time of creation of the battery charging behavior. The `LoadFromDefinition` procedure takes the form

```
procedure TBatteryCharging.LoadFromDefinition(ObjDef: TObjectDefinition);

begin
  inherited;
  fCorrespondingBatteryName :=
    ObjDef.PropertyAsString('CorrespondingBatteryName');
  fChargingStartTime := ObjDef.PropertyAsFloat('ChargingStartTime');
end;
```

Finally, the destructor should be written. In addition to calling the default constructor inherited from `TObject`, it needs to set the `fBatteryPointer` to **`nil`**:

```
destructor TBatteryCharging.Destroy;

begin
  fBatteryPointer := nil;
  inherited;
end;
```

The battery charging behavior is now complete. Take some time to make sure that it is identical to the battery charging behavior provided in the file **BatteryCharging.pas** in the folder ***Pre-defined behaviors***. Now, when the behaviors have been completed and saved, they can be added to the application. In order to do so, select Project **–** Add to project… in the main Delphi window. Browse to the folder ***User-defined behaviors***, and add the two files **CicularNavigation.pas** and **BatteryCharging.pas**.

3.3.2 Writing the definition files

While the actual code for the program has now be completed, there are a few components missing before the program can actually be run. These components are:

(1) A definition file for the robot, specifying its physical properties (i.e. its mass, moment of inertia, height etc.), its motors, and its sensors.
(2) A definition file for the brain of the robot. This file should define the general structure of the brain, i.e. it should specify the behavioral repertoire and the parameters for each behavior.
(3) An arena file, defining the environment in which the robot will operate.

Writing these files from scratch is rather cumbersome. Fortunately, however, they can often be based on existing templates. For the present example, we will contend ourselves with a brief analyis of the files needed for the program to run.

```
object Robot: TRobot

  object Body: TDifferentiallySteeredBody
    Position = 1.0 1.5 0.0
    Mass = 10.0
    Radius = 0.20
    MomentOfInertia = 0.005
    WheelRadius = 0.1
    WheelWidth = 0.02
    Height = 1.0
    Alpha = 1.25
    Beta = 0.39

    object Battery: TEnergySource
      MinEnergy = 0.0
      MaxEnergy = 1.0
      InitialEnergy = 1.0
      DischargeRateAtRest = 0.02
      DischargeRateInMotion = 0.30
      ChargeRate = 0.10
    end

    object Motor1: TDCMotor
      MaximumVoltage = 12.0
      TorqueConstant = 0.0333
      BackEMFConstant = 0.25
      ArmatureResistance = 0.62
      CoulombFriction = 0.008
      ViscousFriction = 0.02
      GearRatio = 4.0
      GearEfficiency = 1.0
      MaxTorque = 1.00
    end

    object Motor2: TDCMotor
      MaximumVoltage = 12.0
      TorqueConstant = 0.0333
      BackEMFConstant = 0.25
      ArmatureResistance = 0.62
      CoulombFriction = 0.008
      ViscousFriction = 0.02
      GearRatio = 4.0
      GearEfficiency = 1.0
      MaxTorque = 1.00
    end

    object BatterySensor1: TBatterySensor
      CorrespondingBatteryName = 'Battery'
    end

  end   # Body

  object Brain: TBrain
    DefinitionFile = '..\Brains\UFLibBasicDemoRoboticBrain.txt'
  end

end #Robot
```

Fig. 12. The robot definition file.

The robot definition file

To begin with, open the folder ***Data*** and then the folder ***Robots***. Next, open the file **UFLibBasicDemoRobot.txt**. The contents of the file are as shown in Fig. 12 above. In general, definition files (both for the body and for the brain of a robot) contain nested structures beginning with the phrase

```
Object <objectname>: T<objectclassname>
```

and ending with the word `End`. In this case, the object `Robot` (of type `TRobot`) contains an object `Body` (of type `TDifferentiallySteeredBody`, which is a descendant class of `TBody`), which, in turn, contains definitions of the physical parameters of the robot and of the battery sensor and the two motor. Finally, the lines

```
object Brain: TBrain
    DefinitionFile = '..\Brains\UFLibBasicDemoRoboticBrain.txt'
end
```

indicate the location of the brain definition file. For clarity, the definition of the brain is normally placed in a separate file, even though it is theoretically possible to place it directly in the robot definition file (in which case the brain definition will, of course, no longer be needed). Separating the brain definition file from the definition of the body also makes it easy to replace the brain definition file by changing a single line in the robot definition file.

Most of the parameters listed in the robot definition file are self-explanatory. The battery discharges according to

$$\mathrm{d}E/\mathrm{d}t = -c_1 - c_2\, v, \qquad (4)$$

where E is the battery energy, v is the speed of the robot, i.e. the modulus of the velocity vector $\boldsymbol{v}$. The constant c_1 corresponds to the `DischargeRateAtRest` defined in the robot definition file, and c_2 corresponds to the `DischargeRateInMotion`. The discharging of the battery is taken care of by the procedure `TEnergySource.Discharge`, When the battery is charging, the equation for its energy changes to

$$\mathrm{d}E/\mathrm{d}t = -c_1 - c_2\, v + c_3, \qquad (5)$$

where c_3 corresponds to the parameter `ChargeRate` in the robot definition file. Note the difference between the actual battery, and the battery sensor, defined close to the end of the robot definition file: the battery is needed for the robot to move, and the battery sensors (if available) allows the robot to monitor the state of the battery.

The brain definition file

Turning now to the definition file for the robotic brain, return to the folder ***Data***, open the folder ***Brains*** and then the file **UFLibBasicDemoRoboticBrain.txt**. The contents of the file are as shown in Fig. 13 below.

```
object Brain: TBrain

  object Hormones: THormones

    object Hunger: THormone
      MinimumLevel = 0.0
      MaximumLevel = 1.0
    end

  end #Hormones

  object Behaviors: TBehaviorList
    Level = 1

    object Navigation: TCircularNavigation
      LeftMotorOutput = 0.9
      RightMotorOutput = 0.1

      object StateVariables: TStateVariables
        object StateVariable1: TInternalPhysicalVariable
         StateVariableType = 'svtInternalPhysical'
         CorrespondingSensorName = 'BatterySensor1'
         ReadingProcedure = 'rpSinglePixel'
         Pixel = 1 1
        end
      end
    end #CircularNavigation

    object BatteryCharging: TBatteryCharging
      CorrespondingBatteryName = 'Battery'
      ChargingStartTime = 0.5

      object StateVariables: TStateVariables
        object StateVariable1: TInternalAbstractVariable
          StateVariableType = 'svtInternalAbstract'
          CorrespondingHormoneName = 'Hunger'
        end
      end
    end #BatteryCharging

  end #Behaviors
end
```

Fig. 13. The brain definition file, UFLibBasicDemoRoboticBrain.txt.

As can be seen, the syntax of the brain definition file is similar to that of the robot definition file. First, the object `Brain` (of type `TBrain`) is defined. Next, the hormone variables are defined. In this case, one such variable is used, as discussed in connection with the writing of the battery charging behavior above. Next, the behavioral repertoire follows. In the UF method, it is possible to define a hierarchical structure, in which any given behavior may contain a behavior list with several other behaviors, which, in turn, may contain their own behavior lists etc. The `Level` parameter indicates the hierarchical level on which the behaviors in the current behavior list are located. In this simple example, there are only two behaviors, and they are located on the same hierarchical level (=1).

The behaviors then follow. The definition of any behavior begins with a specification of the *parameters* of the behavior. Thus, for example, the circular navigation behavior has two parameters, `LeftMotorOutput` and `RightMotorOutput`, which are set to 0.8 and 0.5, respectively. Next, the *input variables* normally follow. These are variables used by the behaviors themselves. However, in this simple example, no input variables are needed (for an example of the definition of such variables, see the brain definition file associated with the example given in Sect. 4 below). Finally, the *state variables* should be defined. As mentioned above, these are scalar variables that are used in the utility functions, which, in the UF method, are optimized by the evolutionary algorithm. In this particular example, U_1 depends on the state variable E, whereas U_2 depends on the variable x. These variables are introduced towards the end of the definition of the battery charging behavior.

As described in Sect. 1.2, in the UF method, there are three types of state variables, namely external variables, internal physical variables, and internal abstract variables. Variables of the first two types may be extracted from sensors whose readings are non-scalar. For example, a state variable may be defined as the average reading of a laser range finder (containing perhaps hundreds of rays), or as the reading of a single ray. In the former case, the *reading procedure* `rpAverage` would be used whereas, in the latter case, the reading procedure `rpSinglePixel` would be used. The reading of a battery sensor is of course scalar, but since the corresponding state variable is defined as an internal physical variable a reading procedure *must* nevertheless be specified, hence the two lines

```
ReadingProcedure = 'rpSinglePixel'
Pixel = 1 1
```

in its definition. Internal abstract state variables, by contrast, correspond to the reading of hormone variables, which are always scalar, so no reading procedure needs to be defined for such variables.

The arena file

As a final step before running the program, an arena file must be defined. Move to the folder ***Data - Arenas***, and open the **UFLibBasicDemoArena.txt** file. Note that only a part of the arena file is shown in Fig. 14 below. The structure of arena files is similar to that of the other two files described above, even though the arena file does not contain nested object definitions. The arena used in this example consists of five objects: a floor and four walls of type `TcornerWall`. Note that all objects defining an arena *must* have unique names (e.g. `object_1`, `object_2` etc.).

```
# Generated by ArenaBuilder 20050525 10:38:29 (GMT+1)
object Object_0: TFloor
  Position = 0.000 0.000 0.000
  Velocity = 0.000 0.000 0.000
  Angle = 0.000
  Height = 2.500
  Mass = -1.000
  RGBColor = 0 0 0
  Texture = Textures/marbletiles.jpg
  Length = 10.000
  Width = 10.000
  TileLength = 2.500
  TileWidth = 2.500
end

object Object_1: TCornerWall
  Position = -5.100 -5.100 0.000
  Velocity = 0.000 0.000 0.000
  Angle = 0.000
  Height = 1.500
  Mass = -1.000
  RGBColor = 0 0 0
#  Texture =
  LengthPart1 = 5.000
  LengthPart2 = 5.000
  Thickness = 0.200
  Transparent = False
  HasWallPaperFront = True
  WallPaperFront = Textures/wallpaper.jpg
  HasWallPaperBack = False
#  WallPaperBack =
  TextureTileLength = 1.000
  TextureTileHeight = 1.000
End
```

< *Truncated …* >

Fig. 14. (Part of) the arena definition file.

Lines beginning with `#` are comments, i.e. they are ignored when the file is read. Note that it is possible to add arbitrary textures to the arena, in order to enhance its appearance. For example, in this case, textures contained in the files **marbletiles.jpg** and **wallpaper.jpg** (located in the ***Textures*** folder) have been added to the floor and walls, respectively. The arena is shown in Fig. 15 below.

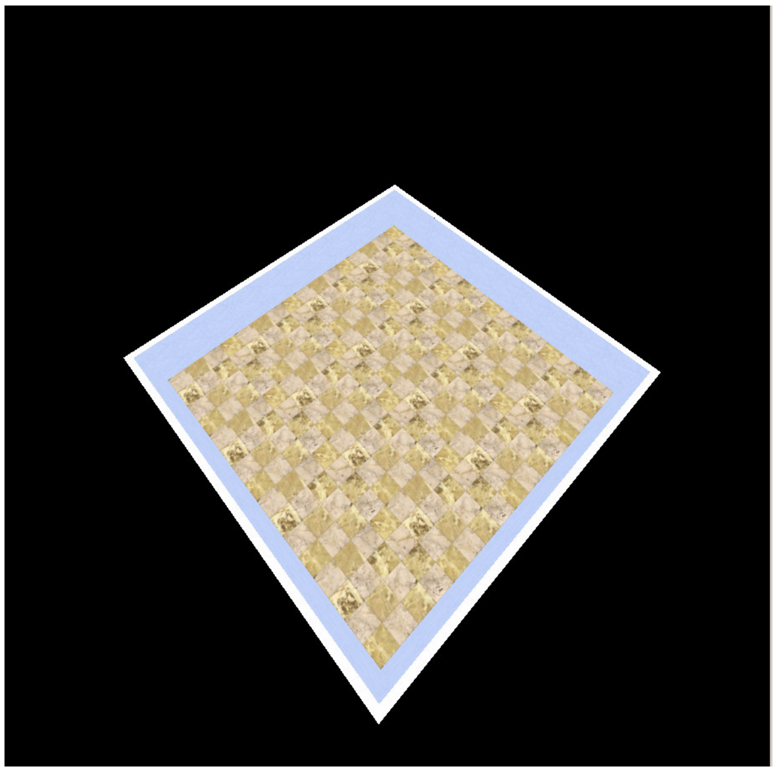


Fig. 15. The arena, with textures added to the floor and the walls.

Finally, press F9 to compile and run the program. It should, of course, work exactly as the program compiled in Sect 3.2 above.

3.4 Final notes

3.4.1 Computer requirements

When the Inspect button is pressed, the program attempts to show the motion of the robot in real time. However, on some computers (notably laptops with insufficients graphics capabilities), the motion may become very slow. In order to runs programs based on the UFLibrary, it is recommended to use a computer running Windows XP and having a clock frequency of at least 2.5 GHz, equipped with a good graphics card with at least 64 MB memory.

3.4.2 Future releases

The UFLibrary is under continuous development. New releases will be made frequently. Please make sure to check the UFLibrary web page often for new releases.

4. Additional example

In this version of the tutorial, only one additional example will be given, namely that of a robot navigating in an arena with obstacles. This robot is equipped with two behaviors, namely *StraightLineNavigation* (which, as the name implies, makes the robot navigate in a straight line, using equal torques for each motor) and *CollisionAvoidance* (in which the torques for the two motors are set to values with equal magnitude but opposite sign, making the robot turn without moving its center-of-mass). The example can be found in the folder ***SimpleNavigation*** (again, the example is provided for Delphi versions 5 and 6). The two behavior units **StraightLineNavigation.pas** and **CollisionAvoidance.pas** are located in the ***behaviors*** folder.

As in the previous example, double-click the project icon, in this case named **SimpleNavigationProject.dpr**, and add the search path to the UFLibrary .dcp-file, as described in Figs. 5-6 above. Then press F9 to compile and run the program. Let the program run for a few minutes, and then press the inspect button to analyze the run. After completing the run, inspect the definition file, particularly the brain definition file (**SimpleNavigationRoboticBrain.txt**, located in the ***Brains*** folder). Unlike the simple example described above, this example involves the use not only of state variables but input variables as well. Next, analyze the source files for the two behaviors.

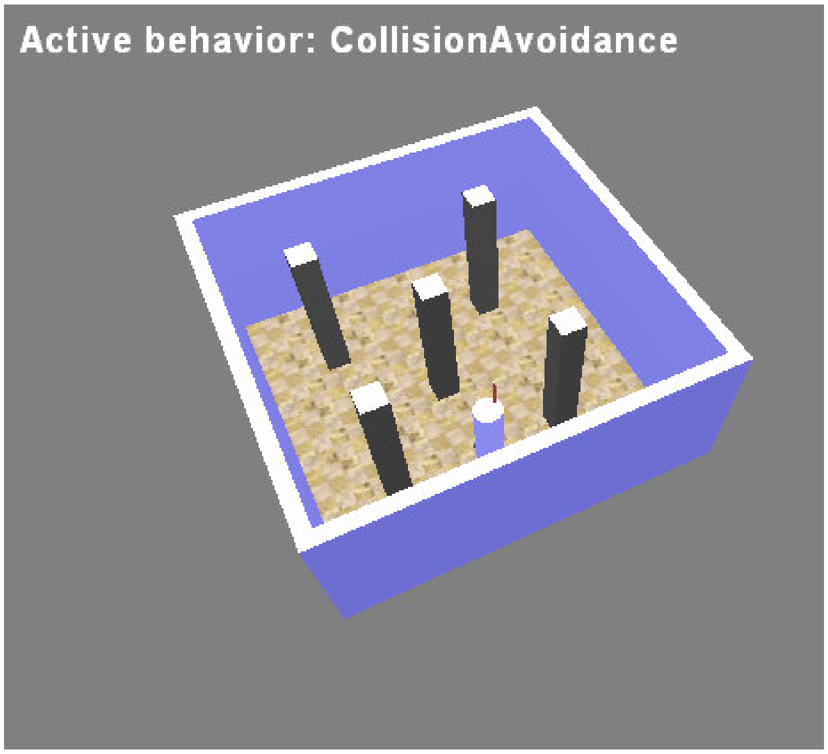


Fig. 16. The simple navigation robot in action.

Contact information

For further information concerning this example or the UFLibrary in general, please feel free to contact Mattias Wahde at mattias.wahde@chalmers.se. You may also wish to visit the web page of the Adaptive systems research group at Chalmers University of Technology, located at www.me.chalmers.se/~mwahde.

References

Reference [1] is the original paper on the UF method. Reference [2] is a general tutorial on evolutionary robotics, which contains some basic information concerning the UF method. Copies of the papers listed below can be obtained upon request to mattias.wahde@me.chalmers.se

[1] Wahde, M. *A Method for Behavioral Organization for Autonomous Robots Based on Evolutionary Optimization of Utility Functions*, Journal of Systems and Control Engineering (IMechI), **217**, 249-258, 2003

[2] Wahde, M. *Evolutionary robotics – The use of artificial evolution in robotcs*, Tutorial presented at IROS 2004 in Sendai, Japan, available at www.me.chalmers.se/~mwahde/robotics/TechReports/TR-BBR-2004-001.pdf

[3] Pettersson, J. and Wahde, M. Pettersson, J. and Wahde, M. *Application of the utility function method for behavioral organization in a locomotion task*, IEEE Trans. Ev. Comp. (In press, 2005)

[4] Sandholt, H. and Wahde, M. *A study of multiple behavior implementations in connection with the utility function method for behavioral organization*, Submitted to the Journal of Robotics and Autonomous Systems, 2005.

[5] Pettersson, J and Wahde, M. *UFLibrary: A Simulation Library Implementing the Utility Function Method for Behavioral Organization in Autonomous Robots,* Submitted to SMC2005

[6] Wahde, M., Pettersson, J., Sandholt, H., and Wolff, K. *Behavioral Selection using the Utility Function Method: A Case Study Involving a Simple Guard Robot*, Submitted to AmiRE 2005