A Genetic Programming Environment in Smalltalk

Tony White
Bell-Northern Research
P.O. Box 3511 Station C
Ottawa, Ontario, CANADA K1Y 4H7
arpw@bnr.ca

Abstract

Genetic Programming (GP), an extension of the Genetic Algorithm (GA), is a powerful optimization technique wherein tree objects representing programs are manipulated in order to evolve programmatic solutions to given problems. A number of GP implementations have been forthcoming -- the original being in LISP -- all of which have been deficient in one or more areas. This author's implementation of a Genetic Programming Environment in Smalltalk (GPEIST) attempts to provide an environment with a rich set of inspectors, utilities and instrumentation that is extensible enough to provide a test bed for GP problems.

1 Introduction

Genetic algorithms (GAs) were originally proposed as a search technique for use in optimization problems. Many optimization problems are hard and although in specific cases algorithms of considerable importance have been developed for them, they are quite often not able to guarantee an optimal solution. Examples are the use of Simulated Annealing for wire routing and component placement problems [Kirkpatrick et al, 1983]. General techniques for the solution of NP-hard problems have not been forthcoming.

Their considerable power arises from the characteristics of survival of the fittest and adaptation achieved through mixing of genetic information using operators such as crossover and mutation. The power of a GA has been analyzed in terms of the Schema Theorem [Holland, 1975] and the Building Block Hypothesis [Goldberg, 1989]. In the schema analysis, search is shown to proceed through identification of highly fit schema, and that such schema are preferentially propagated during the reproductive phase of a GA.

Genetic algorithms are stochastic search algorithms whose search methods model aspects of Darwinian evolution. As stated in [Davis, Steenstrup, 1987]:

"... the metaphor underlying genetic algorithms is that of natural evolution. In evolution, the problem each species faces is one of searching for beneficial adaptations to a complicated and changing environment. The 'knowledge' that each species has gained is embodied in the makeup of the chromosomes of its members."

The basic principle behind genetic algorithms is to do what Nature does. The fitter the individual, the more likely it is to survive to be able to reproduce and thereby ensure that its genes survive to the next generation. While this section provides a brief introduction to genetic algorithms, more complete descriptions can be found in [Goldberg, 1989] and [Michalewicz, 1992].

Genetic algorithms use a vocabulary drawn from natural genetics. We talk about *individuals* (or *genotypes*) in a population. These individuals are sometimes also referred to as

chromosomes or *strings*. Having individual and chromosome as synonyms in this context is a little confusing in that in man, for example, an individual is made of cells each of which contain 46 chromosomes. In genetic algorithms, a population consists of a set of single chromosomes. Chromosomes are made up of units called *genes* or *features* most often arranged in a linear formation or string. A feature is just a set of genes. Each gene controls the inheritance of one or more traits. The genes are said to express a given feature. Genes of certain features are located at specific locations on the chromosome - these are called *loci* or string positions. A feature which can have several values means that the associated genes can take on many values. The values which a gene can take are called *alleles*.

In the context of this paper, each chromosome would represent a potential solution to an optimization problem. An evolution process run on a population of such chromosomes corresponds to a search through a space of potential solutions. The search process requires balancing two conflicting objectives: exploiting the best solutions at any point during the process and exploring the search space. Hill climbing is an example of a strategy which retains as much of the best solution to date as possible when moving to the next solution - only small perturbations are made and the results retained only if improvement is seen. Consequently, it neglects the exploration of the search space and only guarantees to obtain an optimal solution if the solution space contains a single peak. Random search, on the other hand, explores the solution space without regard to the structure of the best solution to date nor to the region of the space where it was obtained. Genetic algorithms combine both solution exploitation and exploration of the search space in an extremely effective way.

Genetic algorithms have been used in a diverse set of optimization problems such as wire routing, travelling salesman problems, network link design [Coombs, Davis, 1987], [Davis, Coombs, 1987], adaptive control and many others. Consult [Michalewicz, 1992] for more details on optimization problems in the areas of scheduling, strategy formulation and transportation problems.

Genetic algorithms differ from more convectional search algorithms in that they work with a population of chromosomes (solutions) rather than a single chromosome (solution). Genetic algorithms also work indirectly, in that problem parameters are encoded before being manipulated by genetic operators. Finally genetic algorithms differ from most conventional search strategies in that they use probabilistic transition rules, not deterministic rules. As such they belong to the class of stochastic search algorithms.

Restricting the discussion to chromosomes represented as bit strings, each chromosome is the same length and each gene can take values drawn from {0,1} - the so called binary alphabet. In this case, the chromosome is represented by a bit string and for a chromosome of length l, 2^l states can be represented. Generally, for an alphabet of cardinality m, m^l states can be represented[1].

1. The principle of minimum alphabets states that we should use the smallest alphabet which makes sense for a particular problem and it is for this reason that a binary representation is most often chosen. However, this principle has most recently come to be questioned by genetic algorithm theorists.

Genetic algorithms are often characterized as algorithms that are better than hill climbing algorithms because they manipulate combinations of bit strings called schemata[1]. If we consider the binary alphabet for the moment, the alphabet of associated schemata would be {0,1,#}, where the # symbol indicates unknown. Figure 1 shows a schema and two possible strings which are examples of it. A schema represents a class of strings.

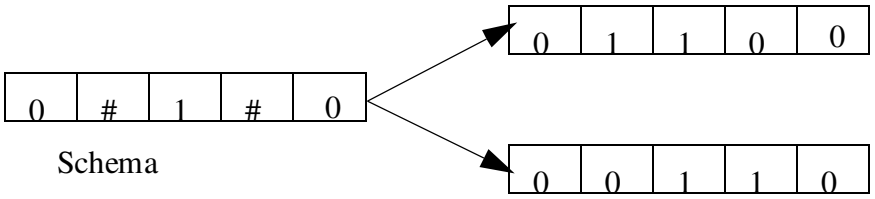

Figure 1 Examples of schema

The first important question to answer when using genetic algorithms is how do I encode the important parameters in my optimization problem? ***The degree of success in using genetic algorithms is strongly linked to solving the representation problem well.*** For example, choosing a representation where the genes describing a particular feature are not contiguous in the chromosome generally leads to poor genetic algorithm performance.

The next problem to resolve when encoding a problem in genetic algorithms form is to provide a *fitness function*. The fitness function takes the chromosome and generates a single value which represents the utility of a particular solution. The fitness function is analogous to the objective function found in Operational Research. In some domains such as function optimization, the fitness function is obvious - it is the function itself. In others, such as the partitioning of m objects into k classes, the fitness function is less obvious and requires considerable thought. The ideal fitness function should, as far as is possible, provide an accurate mapping between the utility of the solution in the real world and the modelled world such that the relative fitness values of potential solutions are accurately captured. As an example of a poor fitness function consider a function in which the fitness of solution A is less than the fitness of solution B while the real world utility of solution A is greater than that of solution B. Such a fitness function could never hope to find an acceptable optimal solution. This is discussed in a later section.

Having resolved the representation and utility mapping problems, all that remains is the mechanics of the genetic algorithm itself. The genetic algorithm can best be described by the program below:

```
Program GeneticAlgorithm()
Begin
            initializePopulation();
            evaluatePopulationFitness();
            while notFinished() do
```

1. The terms schemata and schema are used interchangeably.

```
        Begin
           for i:= 0 to populationSize by 2 do
           Begin
              offspring1 := randomRouletteWheel();
              offspring2 := randomRouletteWheel();
              if random() < crossoverProbability then
                 newOffspring := crossover(offspring1, offspring2);
              newPopulation[i] := mutate(newOffspring[0]);
              newPopulation[i+1] := mutate(newOffsptring[1]);
           End
           population := newPopulation;
           evaluatePopulationFitness();
        End
End
```

The various procedural elements of the program are described in the following sections.

1.1 initializePopulation

The initializePopulation procedure is quite simple; we create a population of chromosomes where each chromosome is a binary vector of *l* bits. All *l* bits for each chromosome are initialized randomly.

Program control parameters are: *lengthOfChromosome* and *populationSize*.

1.2 notFinished

The notFinished function returns true if the number of trials does not exceed a program threshold, and the fitness of the best individual is less than a program threshold.

Program control parameters are: *maxTrials* and *fitnessThreshold*.

1.3 evaluatePopulationFitness

The evaluatePopulationFitness procedure uses the fitness function referred to above in order to compute the fitness of each chromosome in the population. These values are stored as part of the population.

1.4 random

The random function returns a number in the range (0,1).

1.5 randomRouletteWheel

The randomRouletteWheel function is used to choose which chromosomes from one generation should be reproduced in the next one. This is done by calculating an imaginary roulette wheel where each chromosome has a probability of selection calculated from:

$$p(i) = \frac{f(i)}{f_{tot}}, f_{tot} = \sum_{i=0}^{populationSize} f(i)$$

This is called fitness proportional selection. Viewed as a roulette wheel, for a population of five chromosomes, Figure 2 might be visualized.

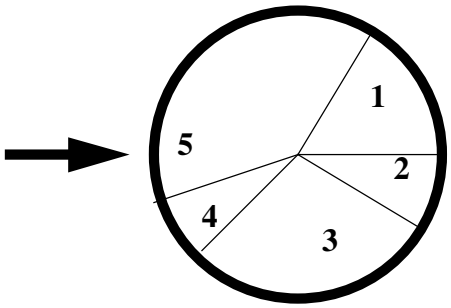

Figure 2 Population Roulette Wheel

In order to generate the new population, the imaginary roulette wheel is spun a total of *populationSize* times and each time wherever it stops, that chromosome is reproduced in the new population. Referring to Figure 2, chromosome 5 should be reproduced in the next generation. In fact, chromosome 5 occupies approximately 40% of the roulette wheel and we would expect in the neighborhood of that percentage of the new population to be chromosome 5. Obviously, several chromosomes will appear multiple times in the new population and it is just this mechanism - survival of the fittest - that causes highly fit members of the population to dominate subsequent generations.

1.6 crossover

The crossover procedure takes two chromosomes as input and generates two new chromosomes as output. The crossover operation occurs by selecting a position between 1 and l-1 in the first string. This identifies two substrings A_1 and A_2, where A_1A_2 is the same as the original string A. The same position is used for string B, generating the substrings B_1 and B_2. The crossover procedure returns the strings A_1B_2 and B_1A_2.. Graphically, this procedure is shown in Figure 3:

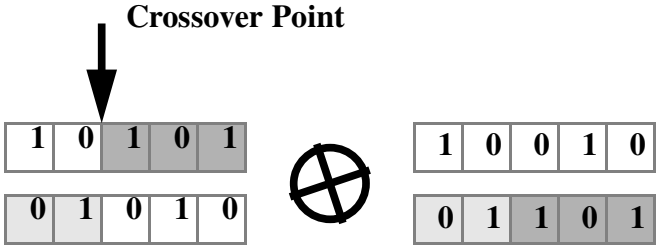

Figure 3 Crossover in action

In the above figure, is meant to represent the crossover operator. As can be seen from the above example, completely different chromosomes can be returned as a result of the action of this procedure. It should also be noted that the crossover procedure occurs under the control of a program parameter - *crossoverProbability.*

1.7 mutate

The mutate procedure allows modified alleles to be introduced to the new population after crossover. The mutate procedure takes a chromosome and *for each bit* executes the following:

```
if random() < mutationProbability then
                toggleBit();
```

The mutation of an input chromosome occurs under the control of a program parameters - *mutationProbability*. The mutation operation ensures that the population maintains some diversity as we proceed from generation to generation.

2 Basic Genetic Programming

GP [Koza, 1990, 1992] is an extension of the basic GA model where simple bit strings are replaced with tree structures. These tree structures represent a parse tree for a computer program. While the GA remains unchanged, the operators used in exchange of genetic information between candidate solutions now change. Rather than deal with bits, GP deals with *functions* and *terminals*. The set of functions used in a particular problem are those program functions which have arguments and therefore are represented by the interior nodes of a tree. The terminal functions are those program functions which have no arguments and hence represent the leaf nodes of the tree. For example, the expression 2+3*4 would be represented as shown in the figure below.

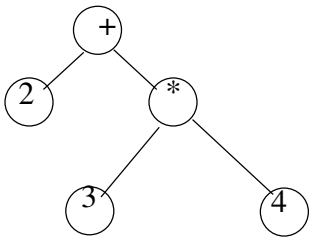

Figure 4 An expression parse tree

The crossover operator now exchanges subtrees between two solutions are demonstrated in the figure below.

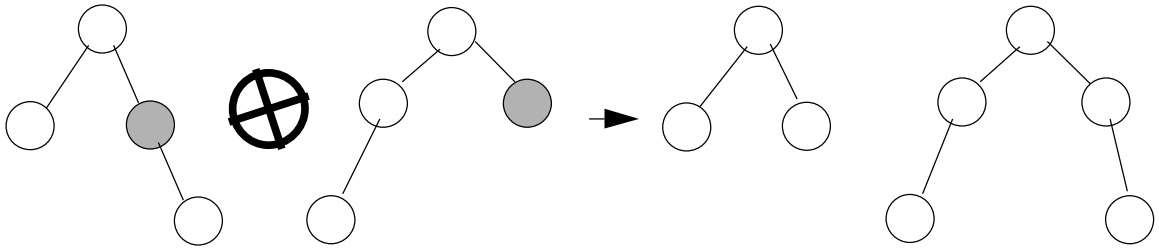

Figure 5 An example of the GP crossover operator

The mutation operator requires similiar redefinition in that rather than a single position being “flipped”, a subtree is replaced with another randomly generated subtree. In GPEIST, the functions and terminals are represented as Smalltalk methods. The function methods are

implemented within the behaviour protocol of a subclass of the GeneticProgram class. The terminal methods are implement within the behaviour protocol of a subclass of the GeneticProgramTerminal class. The names of the two classes are related. The terminals class name is the name of the functions class with the text 'Terminal' appended.

3 Design of GPEIST

The design of GPEIST has a number of distinct elements. In the following sections, the important classes within the system will be briefly described; the behaviour which is defined within the application classes highlighted and a overview of important inter-object messaging presented. Finally, elements of the user interface design will be described.

3.1 Class Hierarchy

As may be inferred from the previous sections, GP and GA share a lot in common. The basic algorithm is identical and fitness measures are required for both techniques. Obviously, the algorithm should not be concerned with the nature of the objects being manipulated -- objects being able to manipulate themselves appropriately when asked to. Although this paper deals with the details of the Genetic Programming environment, care has been taken in order to ensure that generation of an analogous Genetic Algorithm Environment in Smalltalk (GAEIST) will be straightforward. The principal algorithm classes are GeneticBitStringAlgorithm and GeneticProgramming. The relationships between these, and the Smalltalk class hierarchy, is shown in the figure below.

Figure 6 Algorithm class hierarchy

As can be seen from Figure 6, both GP and classical GAs are supported within the G*EIST environment. The GeneticAlgorithm class is a subclass of the class Model in order to make use of the dependency mechanism. It is a meta class, existing only to capture behaviour and attributes common to its subclasses. The principal object classes -- as relate to the GeneticAlgorithm -- are the GeneticBitString and GeneticProgram classes. The GeneticBitString class is used within a conventional GA environment, where bit strings are manipulated genetically. The GeneticProgram and GeneticProgramTerminal classes are used to represent internal and leaf nodes respectively in a GP environment. The relationships between these various classes is shown in the figure below.

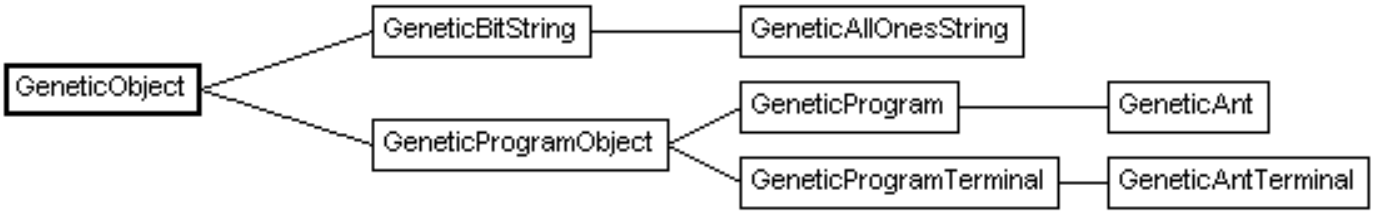

Figure 7 Genetic object class hierarchy

The GeneticObject class is a subclass of Object. The GeneticAllOnesString, GeneticAnt and GeneticAntTerminal classes are all application classes. Typically, in designing a GP application, a subclass of GeneticProgram would be generated containing the behaviour of the Function set and a subclass of GeneticProgram terminal would be generated containing the behaviour of the Terminal set.

All programs, or problems, are executed or evaluated within an environment which tends to be specific in construction to the problem at hand. An environment may contain all of the test data base against which a GA or GP is to be evaluated, or as in the case of the GeneticAnt, a simple 2-dimensional world where ants forage for food. The environment class hierarchy is shown in the figure below.

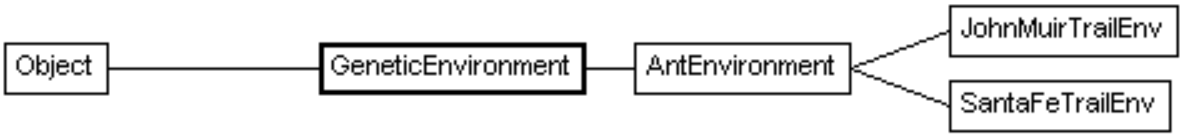

Figure 8 Environment class hierarchy

The JohnMuirTrailEnv and SantaFeTrailEnv classes are application classes relating to the Ant Foraging problem [Collins and Jefferson, 1991], and provide the specification for two different trails.

3.2 Behaviour

Three aspects of behaviour have to be defined for GeneticObject subclasses. Firstly, the crossover and mutation operations have to defined. This behaviour is provided by the **crossover:Using:** and **mutate:Using:** methods. Secondly, when an initial random population is being generated, the **generateUsing:** method is invoked. In the case of a GeneticProgram, when a terminal node is to be created, object mutation occurs in order to have a terminal node created. Finally, the application class has to provide an **eval:** method. The **eval:** method is used in order to evaluate the genetic object in the environment of choice and returns the fitness of that object in the associated environment. Optionally, **preEval:** and **postEval:** methods can be defined in order to provide pre and post processing of the GeneticObject. Typically, the **preEval:** method is used to set up the initial object environment.

An example of post-processing in the case of GP might be to simplify the program - removing subtrees that do not provide any meaningful functionality. For example, in the case of the Ant Foraging problem, the code (ifSensor: (advance: advance)) reduces to (advance:).

In the case of GP, evaluation protocol methods have to be defined for both the interior nodes class and the terminals nodes class. Referring to Figure 7, these two classes are related by the latter having the string 'Terminal' appended to it. The **eval:** method in the case of the GeneticProgram class is quite straightforward:

eval: anEnv

```
"This evaluates the GeneticProgram. We do not have to test for nil
arguments as all terminal nodes are GeneticProgramTerminals"
^self perform: self function with: anEnv.
```

The same code is defined in the terminals class. The code within the various function methods then evaluates the functions and arguments associated with the subtree. An example, taken from the GeneticAnt class is easy to follow, as shown below:

eval: anEnv

```
"This just runs the ant foraging program"
| foodLeft |

foodLeft := anEnv foodTotal - anEnv foodFound.
[foodLeft > 0 and: [anEnv steps > 0]] whileTrue: [
        super eval: anEnv.
        foodLeft := anEnv foodTotal - anEnv foodFound.
].
^foodLeft.
```

The eval: method evaluates the program representing foraging behaviour for as long as time is left or food remains to be gathered. When exhausted, the amount of food left is returned as the fitness of the ant foraging program[1]. The function set for this problem, {**ifSensor:**, **progn:**}, is shown below:

ifSensor: anAnt

```
"This looks at the food at the next location (as determined by advance)
and if food is present executes the 'true' code, otherwise the 'false'
code is executed."
anAnt senseFood
        ifTrue: [(arguments at: 1) eval: anAnt]
        ifFalse: [(arguments at: 2) eval: anAnt].
```

The **isSensor:** method looks at the ant environment and determines whether food is present at the location next to the ant, with the correct orientation. For example, if the ant were to be pointing south and food was present in the adjacent west square, it would not be detected.

progn: anAnt

```
"This evaluates the function in the current environment"
arguments do: [ :arg | arg perform: arg function with: anAnt].
```

The **progn:** method is rather like a block with a sequence of instructions being executed. The progn: method does not directly affect the state of the ant.

The terminal set for this problem, {**advance:**, **turnRight:**, **turnLeft:**} provides the state affecting behaviour for the ant. The various methods are shown below:

advance: anAnt

```
"This causes the ant to advance"
anAnt advance.
```

turnLeft: anAnt

```
"This causes the ant to turn left"
anAnt turnLeft.
```

turnLeft: anAnt

```
"This causes the ant to turn left"
anAnt turnLeft.
```

1. Remember that GPEIST is designed such that fitness minimization is the goal of the GA.

In the case of the ant foraging problem, the environment performs the computation for changes in state as indicated by the messages being sent to the ant itself. The ant environment object then performs state-affecting computation as shown below:

advance

```
"This moves the ant in a given direction"
self decrementSteps.
self changed: #trail.
position := position + orientation.
position x > 32 ifTrue: [position x: 1].
position x <  1 ifTrue: [position x: 32].
position y > 32 ifTrue: [position y: 1].
position y <  1 ifTrue: [position y: 32].
self changed: #position.
self pickUpFood.
```

The “self changed: #trail”, and “self changed: #position” code fragments are important in that they provide the animation hooks in the code. More will be said about this aspect of the system in a later paragraph. The turnLeft method causes the orientation of the ant to change by -90 degrees.

turnLeft

```
"This changes the ant’s direction by turning left."
self decrementSteps.
orientation = (1@0)
        ifTrue: [orientation := 0@-1]
        ifFalse: [
                orientation = (0@-1)
                        ifTrue: [orientation := -1@0]
                        ifFalse: [
                                orientation = (-1@0)
                                        ifTrue: [orientation := 0@1]
                                        ifFalse: [
                                                orientation = (0@1) ifTrue: [
                                                        orientation := 1@0
                                                ]
                                        ]
                        ]
        ].
self changed: #position.
```

The **turnRight** method causes the orientation of the ant to change by +90 degrees.

turnRight

```
"This changes the ant’s direction by turning right."
self decrementSteps.
orientation = (1@0)
        ifTrue: [orientation := 0@1]
        ifFalse: [
                orientation = (0@1)
                        ifTrue: [orientation := -1@0]
                        ifFalse: [
                                orientation = (-1@0)
                                        ifTrue: [orientation := 0@-1]
```

```
                                        ifFalse: [
                                                orientation = (0@-1) ifTrue: [
                                                        orientation := 1@0
                                                ]
                                        ]
                                ]
                ].
        self changed: #position.
```

Again, in the **turnLeft** and **turnRight** methods, the "self changed: #position" code is used for the animation of an ant program. In order to provide the ability to synchronize processes, the **changed:with:** method has been refined in the GeneticAlgorithm and GeneticEnvironment classes as follows:

changed: aSymbol with: anObject

```
"This allows a changed message to be sent to my dependents"

super changed: aSymbol with: anObject.
self dependents do: [:obj | self semaphore next].
```

As can be seen, the usual change dependency mechanism is invoked by use of the parent method, but refined by the addition of messaging which requires the process to wait until all of its dependent objects have written to the shared queue indicating that they have completed their output processing.

3.3 Process architecture

A number of processes exist within the GPEIST architecture. The number and type of these processes depends heavily upon the nature of the use of the system. The process types fall essentially into three categories.

Firstly, simulation processes are used. The main simulation process is created for the GP run itself; that is the process which manipulates the genetic material and evaluates genetic object fitness. Another type of process is an animation process. An animation process is used to evaluate a specific genetic object in an environment but provides output to an animation reporter after appropriate changes in state. Entities capable of simulation support are distinguished by their use of a SharedQueue; this being necessary in order to synchronize updates between ApplicationWindows and simulation processes. Two examples are provided in the current GPEIST implementation -- JohnMuirTrailEnv and SantaFeTrailEnv.

Finally, ApplicationModels are opened in order to provide windows on the simulation experiment being designed or run. Many such windows are available -- including inspectors, browsers and compiler windows -- and these will be described in a later section.

The dependency mechanism is used in order to have the data models underlying the various processes modified. For example, when the appropriate number of genetic object fitness evaluations have been performed, the **changed: #output with: anObject** message is sent to all dependents of the simulation process. This message is eventually received via the **update:with:from:** method of the ApplicationModel.

Typically, the update:with:from: method within the dependent ApplicationModel objects -- such as a Program Viewer or a Population Inspector -- have the following form:

update: aSymbol with: anObject from: aSender

```
"This allows us to update fields on the dependent object"

super update: aSymbol with: anObject from: aSender.
aSymbol == #output ifTrue: [
        self updateFields: anObject.
        aSender continue.
].
```

The updateFields: method typically queries the underlying simulation model for some aspect and updates screen-related instance variables. The "aSender continue" code fragment ensures that the SharedQueue of the simulation process is updated in order that the simulation can continue when all output activity has been completed.

An example of a typical process architecture is shown in the figure below.

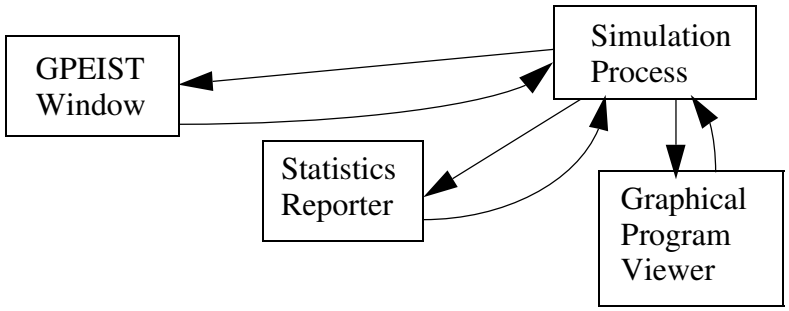

Figure 9 Inter-process messaging

In Figure 9, the straight arrows represent the **changed: #output with: anObject** message being sent to the dependent processes -- in this case three windows. Actually, the message is sent the ApplicationModel which has a scheduled ApplicationWindow. The updates to the ApplicationWindow are only seen when the simulation process relinquishes control of the CPU which it does by waiting for information to be written to its shared queue. As each ApplicationModel completes its output activity, it writes to the simulation process shared queue. This is shown by the curved arrows going from the dependent process to the simulation process. When all output activity is complete, the simulation process continues.

In the case where an animation process is active, the scenario is more complex in that the animation window communicates with an animation process. For example, consider the Ant Foraging problem. When a simulation sends the **changed: #output with: anObject** to the animation window, an animation process is first created and scheduled. With each movement of the ant, two messages are sent to the animation window; one to erase the old ant position and one to draw the new ant position. After each message, the animation process is blocked on its shared queue until the animation window writes to it. When the animation completes, the shared queue of the simulation process is written to in order that it may continue.

The purpose of this synchronization strategy is to have the simulation process continue *only* when all output functions have been completed.

3.4 User interface design

All user interface dialogs and windows are subclasses of the GASTEUI metaclass. This has been done primarily to provide specialized initialization and dialog closing behaviour. A general purpose **update:with:from:** method is also provided.

The main window in the GPEIST system is the GPEIST window which is shown below.

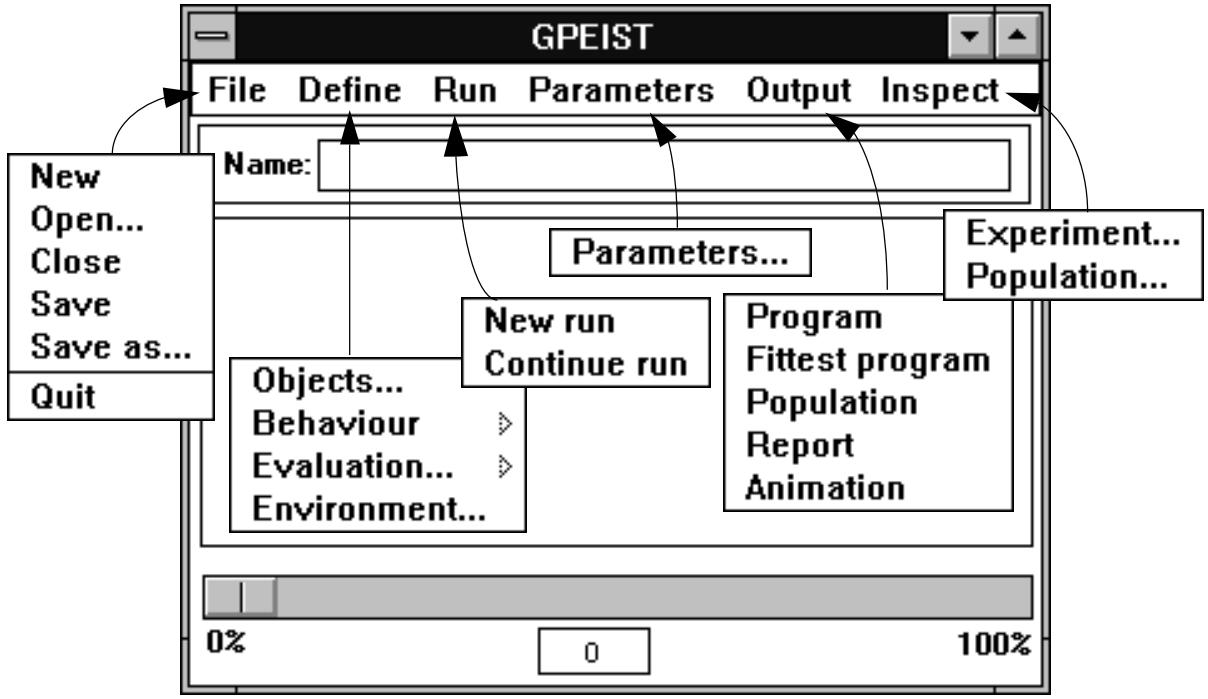

Figure 10 GPEIST main window

The GPEIST window provides access to all functionality available within the system. Functionality includes:

- loading and saving GP experiment definitions;
- defining new GP object classes and environments;
- defining functions and terminals;
- defining evaluation methods;
- running simulations;
- modifying experimental parameters;
- monitoring output, including program, statistics and environmental animations;
- experiment and population inspectors.

The slider widget monitors the progress of the simulation while the name field displays the currently active experiment. Experiments are stored on disk in ASCII format and are selected via a file selection dialog. ASCII was chosen for experiment storage in order to support ease of transfer of experiment specification from one platform to another. GP populations are stored in BinaryObjectStorage format in order to facilitate rapid recovery from disk.

Experimental parameters can easily be modified via the Parameters dialog which is shown on the next page. This dialog allows the user to select crossover and mutation probabilities,

number of evaluations, population size, output frequency and random number seed. The class of object and environment being used by the underlying simulation can also be specified.

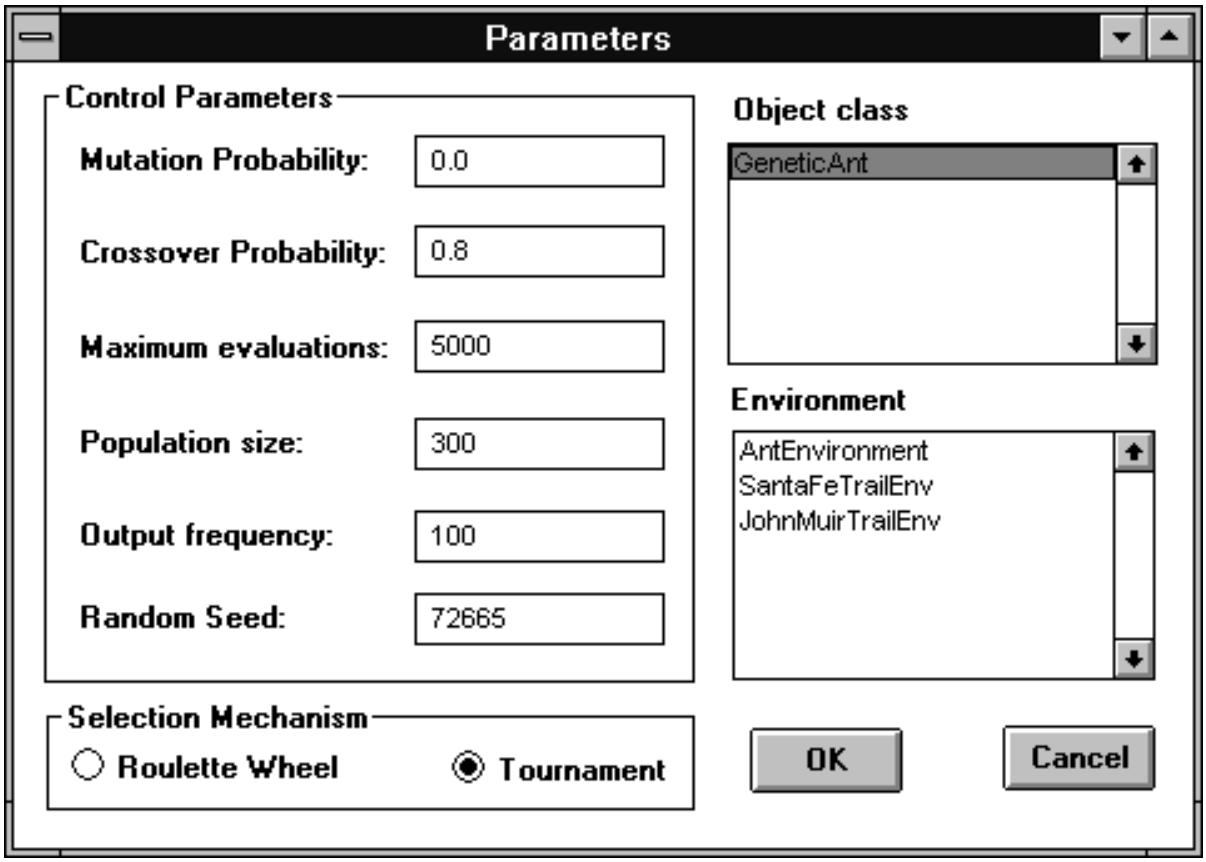

Figure 11 Parameters dialog

The Parameters dialog is implemented in the GASTEParameters dialog. It is accessed via the GPEIST Parameters menu entry.

Several output browsers have been implemented within the GPEIST system. For example, two browsers are provided which cause the saving of the fittest population and entire population to disk respectively. While their principal function is I/O to disk, they report their activities within a browser as shown below.:This browser is implemented using the GeneticBrowser class.

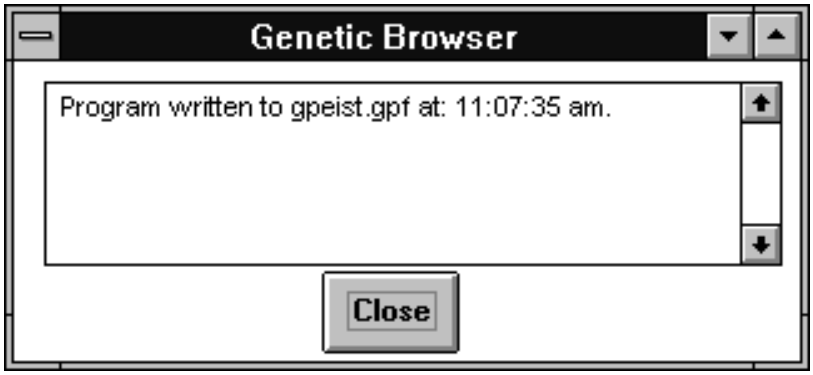

Figure 12 The "fittest program" browser

Genetic Program objects can be viewed using the graphical viewer, an example of which is shown below. This browser is accessed via the Output menu from the GPEIST window.

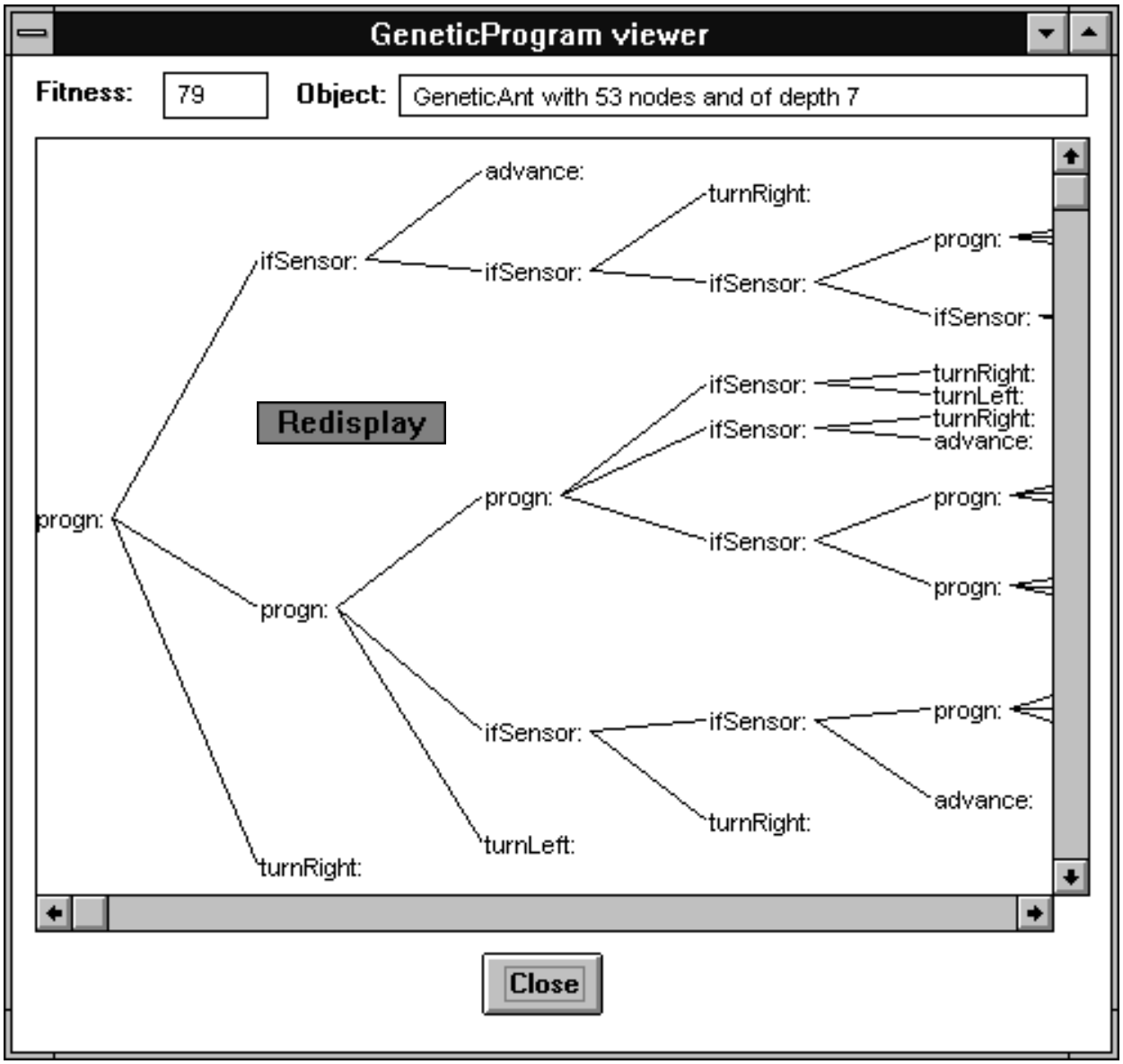

Figure 13 The Genetic Program browser

The GP browser provides the ability to review the structure of the GeneticProgram in a graphical format. The size (in nodes) and depth of the tree are displayed, as is the computed fitness of the program in the environment under test. As implied by a previous section, this dialog changes automatically when the appropriate output message is sent from the simulation process. The dialog above is implemented by the GeneticProgramViewer, GeneticProgramView and GeneticProgramController classes. This is probably the most useful browser in the entire system as it provides an easy to understand format for the programs being manipulated. As this browser is updated periodically, it is easy to see how code subtrees are propagated from parent to child.

An extension to the GP browser is provided in the Population inspector. The purpose of the inspector is to allow the whole population to be reviewed; with one program being displayed

at any time. The population inspector allows whole populations to be retrieved from disk, or saved in BinaryObjectStorage format. This allows complete flexibility in reviewing a GP population.

While no general animation browser is provided in the system -- and could not be as animation is problem specific -- an example has been provided for the Ant Foraging problem. It is implemented in the classes GeneticAntViewer, GeneticAntView and GeneticAntController classes. An example of the dialog is shown in the figure below:

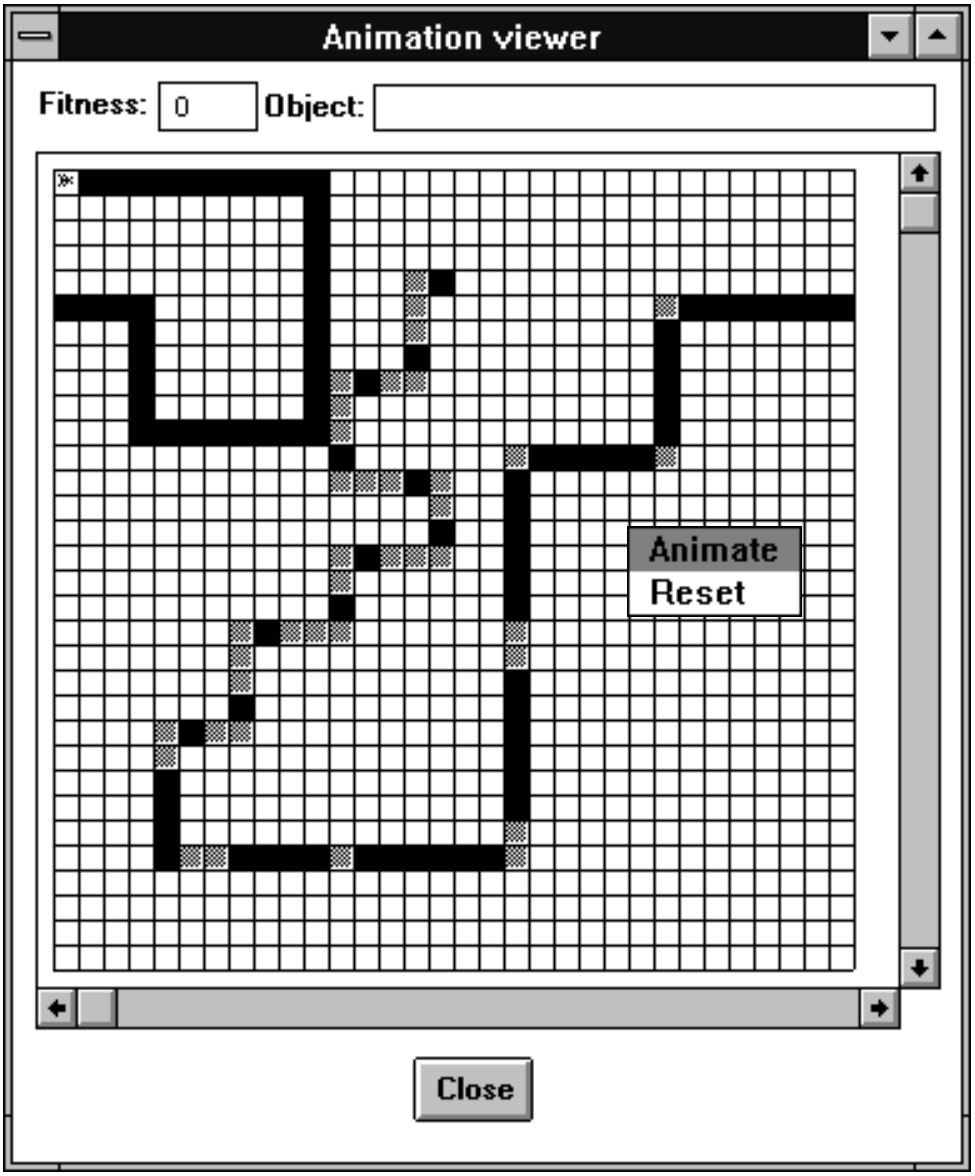

Figure 14 The Ant Foraging animation viewer

The animation viewer provides the ability to review the execution of an ant foraging program. This ApplicationModel object interacts heavily with an AntEnvironment object is order to have each state change mirrored on the viewer.

Run statistics -- such as the mean and variance of GP fitness -- have also been provided. The reporter allows such statistics to be hard copied or saved to disk in text format for use in

analysis by other applications such as a spreadsheet. In this way, the addition of sophisticating charting facilities has been deemed unnecessary.

4 Extending GPEIST

GPEIST can be extended easily through a use of the Define menu entries. New classes of experiment can be added via the Object Definition Dialog. This dialog, shown below contains lists of existing GP program and environment classes.

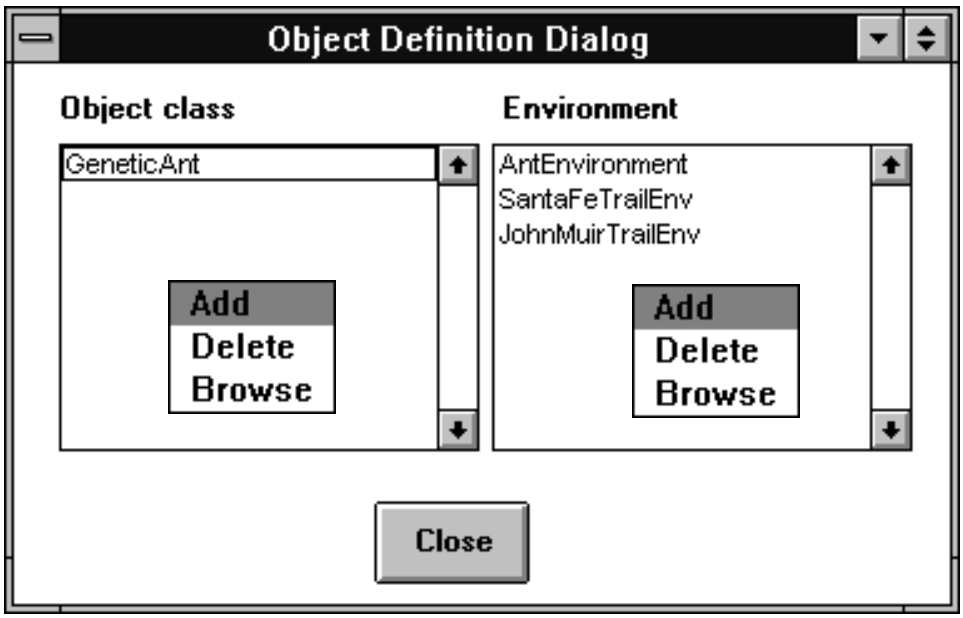

Figure 15 The Object Definition Dialog

Selecting Add from the pull down menu on either list generates the Class Addition dialog.

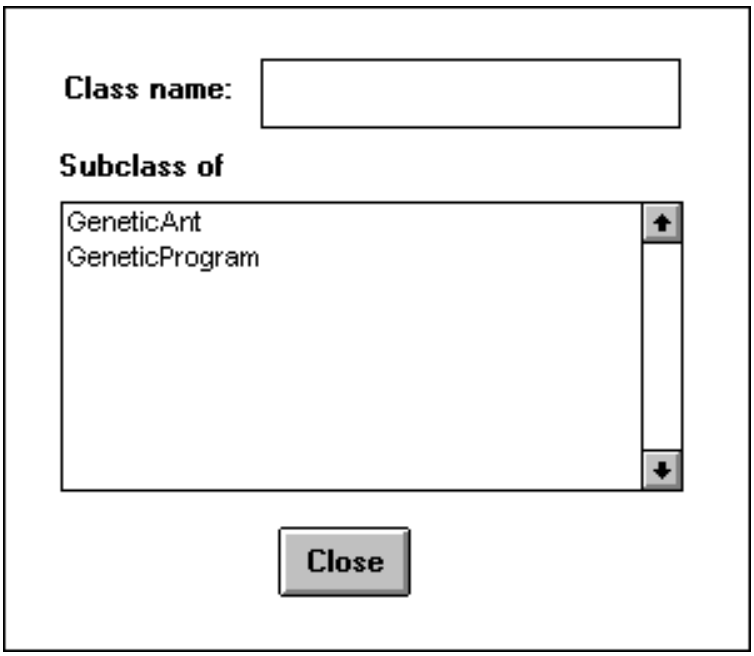

Figure 16 The Object Class definition dialog

By providing a valid class name, and selecting a subclass, two classes will be added to the system. Using as an example ‘Test’, and assuming selection of the GeneticProgram class, a Test class will be added as a subclass of the GeneticProgram class and a TestTerminal class will be added as a subclass of the GeneticProgramTerminal class.

Once the class associated with the experiment has been created, its functions and terminals can be specified. This is done through a use of the Behaviour Editor - an example of which is shown below.

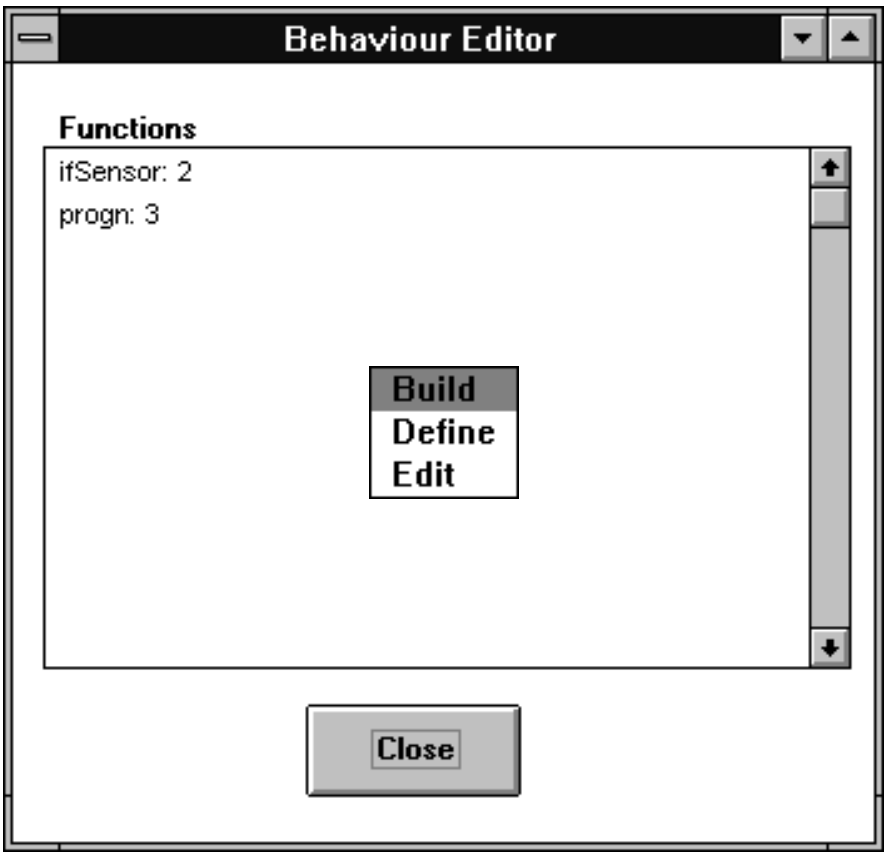

Figure 17 The Behaviour Editor dialog

Selecting ‘Define’ from the text editor pull down menu, causes generation of stub methods in the behaviour protocol of the newly-created object class and associated terminal class.

The newly-created object class **eval:**, **preEval:** and **postEval:** methods can also be created. This is also done through a use of the Behaviour Editor. Default text is provided for these method, for example for preEval:

preEval: anEnv

```
“Default preEval: source”
“This method was created by GASTE”
^(Smalltalk at: anEnv) new.
```

As above, **preEval:** methods are generated automatically for both object and object terminal classes when ‘Define’ is selected.

Finally, an environment can be defined for the experiment under construction. Given the diverse nature of environments -- being problem specific -- a standard Smalltalk 80 class browser is opened when the user decides to define the experiment environment.

5 Adding viewers to GPEIST

It is relatively straightforward to add new viewers into the GPEIST system. The 'Output' menu can easily be modified by accessing the Menu Editor tool and editing the windowMenu for the GASTE class. This is found in the resources protocol for the class. Any viewer classes which are added should be added as subclasses of the GASTEUI meta class in order to inherit the initialization and closing behaviour. Should a 'Close' button be added to such a viewer, its associated action should be the method **clickedClose**. In order for the viewer to receive output notifications from the simulation process an update:with:from: method needs to be provided. The examples provided earlier in the paper and in the various GASTEUI subclasses provide the necessary information on how to do this.

6 Summary

In this paper, the essential design and implementation elements of a Genetic Programming Environment in Smalltalk (GPEIST) have been described. It has been designed to provide support for the specification of new GP experiments -- with the user needing to provide function and terminal method coding -- and ease of use in that experiments, populations of programs and individual programs can be saved on disk. Further, a number of sophisticated browsers and inspectors are provided in order to allow user-insight into the dynamics of evolving programs.

A number of facilities remain to be added, however. Firstly, improved support for animation should be implemented. Also, and probably more importantly, better instrumentation should be added in order to allow (a) the genealogy of a particular program to be reviewed and (b) to allow more information on programs to be obtained. An example of this latter requirement is the need to know how many times a particular node in a program is visited during its execution.

In conclusion, GPEIST provides a object oriented implementation of the Genetic Programming paradigm. It is significantly richer in functionality than systems such as GPQUICK [Singleton, 1994] and, being implemented in Smalltalk 80, is source code transportable between a number of platforms.

7 References

[Collins and Jefferson, 1991] Collins, R., and Jefferson, D., *AntFarm: Towards Simulated Evolution*, in Artificial Life II, Addison-Wesley, pgs 579-601, 1991.

[Coombs, Davis, 1987] Coombs, S., and Davis, L., *Genetic Algorithms and Communication Link Speed Design: Theoretical Considerations*, Proceedings of Second International Conference on Genetic Algorithms, pgs. 252-256, 1987.

[Davis, 1991] Davis, L., (editor), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

[Davis, Coombs, 1987] Davis, L., and Coombs, S., *Genetic Algorithms and Communication Link Speed Design: Constraints and Operators*, Proceedings of Second International Conference on Genetic Algorithms, pgs. 257-260, 1987.

[Goldberg, 1989] Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[Holland, 1975] Holland, J., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

[Kirkpatrick et al, 1983] Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P., *Optimization by simulated annealing*, Science 220(4598), 671-680 (1983).

[Koza, 1990] Koza, J. R., *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*. Stanford University Computer Science Department technical report STAN-CS-90-1314.

[Koza, 1992] Koza, J.R., *Genetic Programming: On the Programming of Computers by means of Natural Selection*, MIT Press, Mass, 1992.

[Singleton, 1994] Singleton, A., *Genetic Programming in C++*, Byte, pgs. 171-176, February, 1994.

[Michalewicz, 1992] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer Verlag Publishers, 1992.