Event-based Graphical Monitoring in the EpochX Genetic Programming Framework

Loïc Vaseux
INSA Rouen
Haute-Normandie
France
loic.vaseux@insa-rouen.fr

Fernando E. B. Otero
School of Computing
University of Kent
Chatham Maritime, ME4 4AG
F.E.B.Otero@kent.ac.uk

Tom Castle
School of Computing
University of Kent
Canterbury, CT2 7NF
tc33@kent.ac.uk

Colin G. Johnson
School of Computing
University of Kent
Canterbury, CT2 7NF
C.G.Johnson@kent.ac.uk

ABSTRACT

EpochX is a genetic programming framework with provision for event management—similar to the Java event model—allowing the notification of particular actions during the lifecycle of the evolutionary algorithm. It also provides a flexible Stats system to gather statistics measures. This paper introduces a graphical interface to the EpochX genetic programming framework, taking full advantage of EpochX's event management. A set of representation-independent and tree-dependent GUI components are presented, showing how statistic information can be presented in a rich format using the information provided by EpochX's Stats system.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Design

Keywords

EpochX, genetic programming, visualization, framework

1. INTRODUCTION

EpochX is an open source genetic programming (GP) framework written in Java. It has been designed with a modular architecture, providing an infrastructure for modifying most aspects of the genetic programming algorithm. The framework provides full support for 3 popular representations: strongly-typed tree GP (STGP) [7, 9], context-free grammar GP (CFG-GP) [15], and grammatical evolution (GE) [10]. EpochX is intended primarily for researchers who are working on genetic programming theory and are interested in the distribution of depth/length/diversity etc. or a large range of other statistical data about their run. While it is built sensibly to avoid performance issues, speed is not the main goal. Several papers used EpochX in their experimental work [2, 5, 6, 13, 12].

One of EpochX's distinctive features is the provision of an event framework. Events are associated with particular actions in the lifecycle of an algorithm—e.g., the start/end of a generation, the start/end of a genetic operator, amongst others.

The framework provides a centralised mechanism for firing events. For example, the `GenerationalStrategy` class fires events at the start and end of each generation:

```
public class GenerationalStrategy
    extends Pipeline
    implements EvolutionaryStrategy {

    public Population process(Population pop) {

        int generation = 1;

        while (!terminate()) {

            EventManager.getInstance().fire(
                StartGeneration.class,
                new StartGeneration(generation);

            pop = super.process(pop);

            EventManager.getInstance().fire(
                EndGeneration.class,
                new EndGeneration(generation);

            generation++;
        }

        return population;
    }
}
```


GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.

The event management includes the use of listeners, which are objects that are notified when an event of interested is fired. Events and listeners are the basis for EpochX's Stats system [11].

The Stats system is a flexible facility to generate and retrieve information about the progress of an evolutionary run. The main idea is to use stat objects, which are implemented as listeners, to calculate/update information based on events and other stat objects. As an example, let us consider the implementation of the `GenerationBestFitness` stat:

```
public class GenerationBestFitness
    extends AbstractStat<EndGeneration> {

    private Fitness best;

    public void refresh(EndGeneration event) {

        Fitness[] fitnesses = AbstractStat
            .get(GenerationFitnesses.class)
            .getFitnesses();
        best = null;

        for (Fitness fitness: fitnesses) {

            if (best == null
                || fitness.compareTo(best) > 0) {

                best = fitness;
            }
        }
    }
}
```

The `GenerationBestFitness` is a listener of `EndGeneration` events, and when a new `EndGeneration` event is fired, the best fitness value is updated based on the fitness values from the current population—the fitness values of the current population is provided by a different `GenerationFitnesses` stat and retrieved from the central repository using the call `AbstractStat.get(GenerationFitnesses.class)`.

In this paper we present a new package that introduces GUI visualization components into the EpochX framework. These components take full advantage of the event management and the Stats system and can be plugged into an algorithm as listeners—there is no need to introduce GUI specific code in the core framework. All the information in the GUI interface is generated using event and stat objects, and presented in a rich format using different graphical representations, e.g., tables, graphs and charts.

In general, the output of GP algorithms are reported as graphs characterising the evolution of the algorithm—e.g., the progress of best, average and worst fitnesses, diversity of fitness values, difference between parent and offspring fitnesses. There are GP frameworks and tools, such as Galapos [4], Opt4j [8] and HeuristicLab [14], that go beyond graphs—e.g., they provide facilities to visualize individuals of a particular iteration of the GP, track parent-offspring sub-trees (sub-trees preserved from the parent and sub-trees generated by mutation)—to allow not only the visualization of the evolution of the population, but also the inner working of the evolutionary process. By introducing GUI visualisation components to EpochX, our motivation is to allow users to better understand how the evolution works and to get insights about the modifications that are introduced by genetic operators in the individuals of the population, either to understand the operations involved (beginner users) or to aid the development of new operators and other algorithmic components (experienced users). At the same time, these visualization components also demonstrate how the information from the event management and the Stats system can be used to generate elaborate output, without requiring directly changes to the GP algorithm.

The remainder of the paper is structured as follows. Section 2 introduces the new graphical interface with consideration for how to use the event management and Stats system to display information on the screen in a friendly way. Section 3 presents several components available for any type of representation whereas elements presented in Section 4 are restricted to the classic tree-based representation. Section 5 lists a complete example of how to run EpochX with the proposed graphical interface. Finally, Section 6 concludes the paper.

2. EVENT-BASED GUI USING STATS INFORMATION

As described in [11], the use of events and stats makes it easier to gather a vast range of information about the execution of an evolutionary algorithm. In most cases, users are interested in visualising this information in a graphical form, either to check the progress of the algorithm or to better understand how the solutions are created.

In EpochX, GUI elements could be updated by events triggered from the Stats system in order to display information in a friendly way—this was briefly discussed in [11]. In this paper, we extend this idea to combine the flexibility of EpochX's Stats system and event management with existing GUI libraries—e.g., Swing and JChart2D—to create a rich graphical user interface.

2.1 Graphical Output

To introduce the idea of using an event-based graphical interface, let us consider a simple example of a customized `JTextArea` component, which displays the information from a stat object specified in the constructor. The stat is registered in the `AbstractStat` repository in the constructor, and the `onEvent(Event)` method is implemented to set the text according to the stat's `toString()` method:

```
public class StatText
   extends JTextArea
   implements Listener<Event> {

   private Class<? extends AbstractStat<?>> klass;

   public StatText(
      Class<? extends AbstractStat<?>> klass) {
      AbstractStat.register(klass);
      this.klass = klass;
   }

   public void onEvent(Event event) {
      setText(AbstractStat.get(klass).toString());
   }
}
```

After creating a `StatText` object, it must be registered in the `EventManager` to start receiving events that trigger the update to the `JTextArea` component's text:

```
StatText text = new
   StatText(GenerationNumber.class);

EventManager.getInstance()
    .add(EndGeneration.class, text);
```

The above example shows how custom GUI components can be updated by implementing the `Listener<Event>` interface to receive event notifications and display the information from a stat object.

2.2 Monitor Package

From the general idea of updating graphical components using events, we have designed a new `Monitor` package—working outside EpochX's core framework—to centralise the creation of graphical visualisation components. The core class of this new module is the `Monitor` class, which extends a `JFrame` to manage and display several visualization components—such as tables, charts or any other Swing customized components.

The layout is a grid of tabbed-panes whose rows and columns' counts can be specified in the constructor. Below is sample code which creates a `Monitor` frame with one row and two columns and adds a monitoring table (described in detail in section 3.1) to the first column:

```
Monitor monitor = new Monitor("GP Monitor", 1, 2);
Table table = new Table("Progress Table");
// table settings
monitor.add(table, 0, 0);
```

The `Monitor` can receive any subclass of `JComponent` (e.g, `JPanel`, `JScrollPane` and `JTable`) through the `add` method. The row and the column index can also be specified, otherwise the component is not visible, but it can be later added to the monitor using the JFrame's menu or the "Add" button. The menu also provides a convenient way to perform usual operations such as exporting a component (e.g. a table in a spreadsheet file, a chart in graphic file) or resizing the frame to the best fit.

The `Monitor` package has ready-to-use customisable GUI components, that in combination with stat objects, can be used to display information and track the progress of the execution of a GP algorithm. These components are divided in to two groups: a set of representation-independent components (Section 3), which provide general information about a GP execution, and a set of tree-dependent components (Section 4), which are specific components for tree-based GP algorithms.

3. REPRESENTATION-INDEPENDENT COMPONENTS

Representation-independent GUI components can be used to display general information about the execution of a GP algorithm, independent from the individual representation used—i.e., these components can be used in combination with the GP, GE, CFG-GP and any other user-specified representation.

3.1 Tables

The `Table` component responds to a primary need: display the raw data in a clear way and refresh it in real-time during the evolutionary run. This is similar to the `StatPrinter` [11], which prints one or several statistics in a specified `OutputStream`—the standard output stream (`System.out`) in most cases—each time a specific event is triggered.

The implementation of the `Table` component, which extends a `JTable`, respects the Model-View-Controller (MVC) design pattern of the Swing architecture. The data model and the controller have been joined in the `TableModel` class to store the raw data, which extends `javax.swing.table.AbstractTableModel`. The stats to be displayed in the table are registered in the `TableModel` and the `onEvent(Event)` method is implemented to perform the table refreshment. Each column of the table represents a different stat object and new columns are added using the `addStat(Class)` method. The method `addListener(Class)` is used to specify the event that triggers the refresh of the table.

Let us consider an example of a commonly used table to show the evolution of the fitness diversity throughout the generations:

```
Table table = new Table("Fitness Diversity", 1000);
table.addStat(GenerationNumber.class);
table.addStat(GenerationFitnessDiversity.class);
table.addListener(EndGeneration.class);

// adds the table to the monitor frame (optional)
monitor.add(table, 0, 0);
```

First of all, a new `Table` is created with two optional arguments: the table name and the refreshing period (time in milliseconds between successive refreshments of the table's view). The user then registers the stats in the `Table`, which will automatically register them in the repository, and each of them will define a new column—the order is determined by the order in which the stats are registered. In the fourth line, the `EndGeneration` event is registered as a listener, specifying that a new row will be added each time an `EndGeneration` event is fired.

Finally, the last (optional) line adds the table to the monitor. In some cases we could want to run a simulation without any interface but keeping an 'off-line' trace of the generated data. One use of this facility is that the gathered data could be later exported to a file on the `EndRun` event (the event fired at the end of the execution of a GP algorithm):

```
File file = new File("output.csv");

EventManager.getInstance().add(EndRun.class,
   new Listener<EndRun>() {
      public void onEvent(EndRun e) {
         table.export(file, Table.FORMAT_CSV);
      }
   });
```

This sample code provides a convenient way to automatically generate a dataset at the end of an evolutionary run without using any graphical output. It also highlights the `export(File file, String format)` method which saves the table according to the specified format. There are two built-in formats available: `FORMAT_CSV` (comma-separated

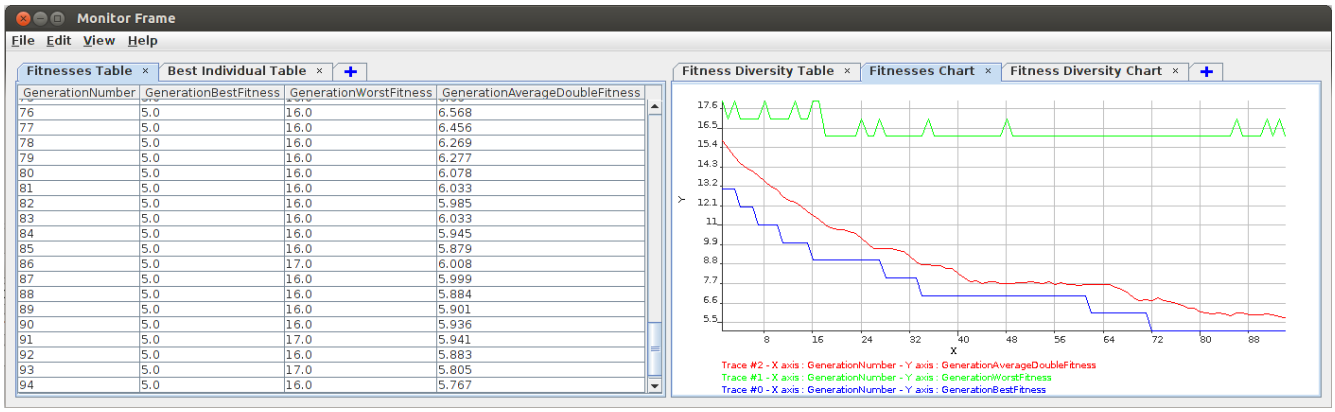

Figure 1: A sample monitor with one rows and two columns. On the left side, Statistics are displayed in a table and refreshed on the `EndGeneration` event. On the right, the same statistics are printed in a chart.

file) and `FORMAT_XLS` (Excel 2000 file). The data from a table can also be exported from the `Monitor` JFrame's menu.

3.2 Charts

While the `Table` component provides a flexible way to display information to the screen in a tabulated form, presenting information in a chart provides a graphical representation of numeric-valued statistics. This feature is implemented as the `Chart` and `ChartTrace` components, which extends the `Chart2D` and the `Trace2DSimple`, from `JChart2D` [1], respectively.

A `Chart` is a component which contains one or several `ChartTrace`. A `ChartTrace` is defined by an X-axis stat, an Y-axis stat and a list of listeners. When an event triggers one of the registered listeners, a point (`TracePoint2D`) is added to the chart, where the (x,y) coordinates correspond to the numeric values of the associated stats.

The following code creates a basic chart which displays the fitness diversity through generations:

```
Chart chart = new Chart("Fitness Diversity");

ChartTrace trace = new ChartTrace("Gen-Fit");
trace.setXStat(GenerationNumber.class);
trace.setYStat(GenerationFitnessDiversity.class);

chart.addTrace(trace);
chart.addListener(EndGeneration.class);
```

A `Chart` component can also be exported as an image by using the export method or the export menu of the `Monitor` JFrame's menu.

3.3 Evolutionary Graph

The `Graph` component provides a clear way to visualize the entire population and kinship networks through generations. Individuals of each generation are represented in a row by round vertices. Those are coloured from blue to red according to the individual's fitness—the color of the best individuals is red—and linked by family ties with their parents. Figure 2 illustrates an example of the evolutionary graph component. The following code adds a new `Graph` component to an existing `Monitor` object:

```
Graph graph = new Graph("Visualization Graph");
monitor.add(graph);
```

The `String` parameter in the `Graph` constructor is optional and sets the name of the component.

The `Graph` component does not only represent data in a convenient form, but also allows user interaction. When an individual is selected by a user, its family links are highlighted and additional information (e.g. its string representation, fitness, genetic operator that originated the individual) are displayed in the bottom panel.

The individuals from a generation (row) are sortable according to:

Fitness: individuals are sorted according to their fitness in an increasing order, so it is possible to visualise from which part of the population the best individuals originated.

Operator: individuals are sorted according to their parent operator, but still coloured depending on their fitness, it is possible to identify which genetic operators produce better individuals—e.g., does the crossover operation provide better offspring than the mutation?

Parent relative position: the parent position determines the order of the next generation in order to reduce link overlaps, which in some cases can facilitate the visualisation.

The implementation of the `Graph` component is divided into three main parts:

Data model (`GraphModel`): organises and stores data information retrieved from the event system to be displayed in the graph. For each individual, the model creates and stores a `GraphVertex` instance which encapsulates the individual and additional information, such as the `Operator` and the parent `GraphVertex`.

View model (`GraphViewModel`): stores and manages all attributes related to the graphical representation of an individual, such as the diameter of vertices, gaps and colours.

Main view (`Graph` and `GraphView`): displays vertices according to the information stored in both `GraphModel` and `GraphViewModel`.

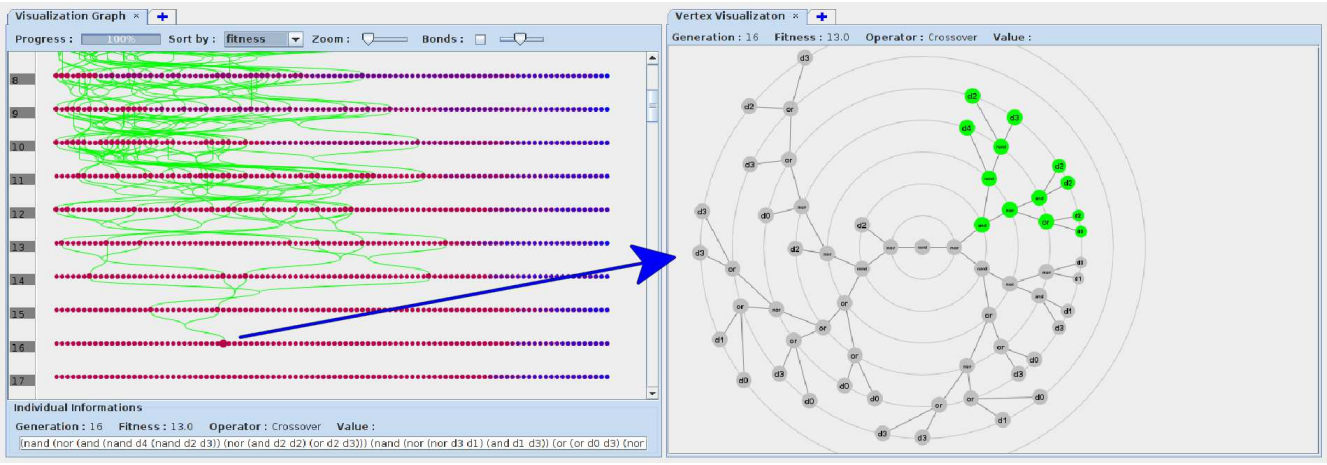

Figure 2: On the left side, the evolutionary graph: individuals are represented in generation rows, sorted by their fitness—in a left-to-right decreasing order—and linked by family bonds. On the right side, the `Tree` component gives the tree representation of the selected individual in a radial form—a user selected subtree is displayed in green.

Since the data model part is entirely separated from the view, it is possible to export and import a set of data. It is also possible to complete a GP run without the graphic interface, but generating a `GraphModel`. The generated model can later be visualized.

The `GraphMonitorWriter` class is responsible for saving the data of a `GraphModel`. The `GraphMonitorWriter` acts as a listener of an event and the data is saved when the specified event is triggered:

```
GraphModel model = new GraphModel();
EventManager.getInstance().add(
   EndRun.class,
   new GraphModelWriter(model, "backup.ser"));
```

The above sample code creates a new graph data model which will be automatically generated during the run of the GP. A new `GraphMonitorWriter` is created for the graph model and the output file is specified. The `EventManager` registers the `GraphMonitorWriter` listener to receive `EndRun` events. When the `GraphMonitorWriter` receives the event, all the data stored in the `GraphModel` is saved to the file. To load the data from a file, a convenient static method is provided by the `Graph` class:

```
GraphModel model =
    Graph.loadInputModel("backup.ser");
Graph graph = new Graph(model);
monitor.add(graph);
```

The above code loads a model from the file "backup.ser", then creates a new `Graph` with the loaded model and adds it to the monitor.

4. TREE-REPRESENTATION COMPONENTS

So far we introduce several generic components available for any GP representation. This section presents GUI components that are specific to tree-based GP algorithms. These components take advantage of the additional information available in the `EndOperator` event about the individuals undergoing a genetic operator. The information is used to determine:

First parent (*Parent1*): the first individual selected to undergo the genetic operator.

Second parent (*Parent2*): the second individual selected to undergo the genetic operator; in case of a mutation or a reproduction, this is `null`.

First point (*Point1*): the index of the node from the first individual selected for crossover or mutation.

Second point (*Point2*): the index of the node from the second individual selected for crossover.

Let us consider the subtree crossover operator, which takes two individuals and produces two offspring. The operator starts by selecting two parent individuals, then it randomly selects a crossover point in each of the parent individuals—*Point1* from the *Parent1* and *Point2* from the *Parent2*. The parent individuals are then cloned and the sub-trees indicated by *Point1* and *Point2* are exchanged, generating two new individuals. This process is illustrated in Figure 3. The subtree mutation operator is similar, except that only one parent (*Parent1*) is involved and the sub-tree indicated by *Point1* is replaced by a randomly generated sub-tree.

As we will discuss in the next sub-sections, the tree-based GUI components display all the information related to how individuals are created—e.g., if the individual was created from *Parent1* or *Parent2*, what was the crossover/mutation point selected (*Point1* and *Point2*), and the origin of subtrees in the individual genotype.

4.1 Tree

The `Tree` component provides a clear way to visualise an individual as a radial graph [3]. Nodes are displayed in circular levels according to their depth. The `Tree` component allows user interaction. It is possible to zoom-in/zoom-out, drag the tree around; nodes of the tree can also be selected

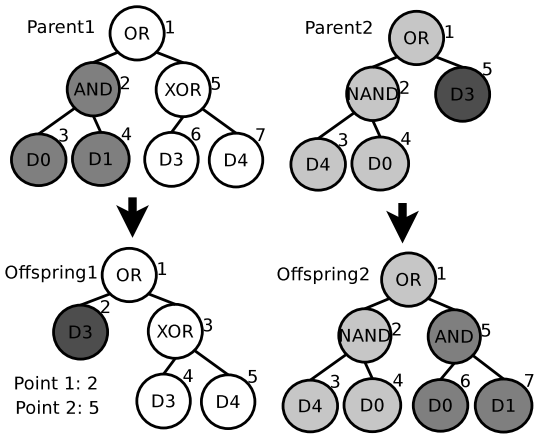

Figure 3: The crossover process: two offspring are produced by exchanging subtrees in the genotype of two selected parent individuals. All the information related to the crossover—*Parent1*, *Parent2*, *Point1* and *Point2*—is displayed in a graphical form by the `OperationPanel` component.

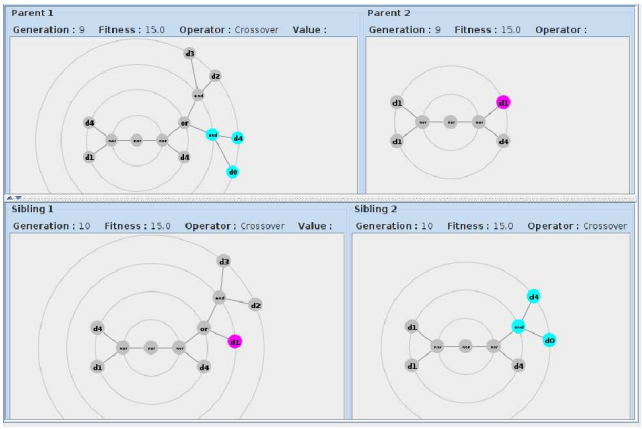

Figure 4: The `OperationPanel` uses `Tree` components and information provided by the `EndOperator` event to represent a crossover operation.

from the graph. In the Figure 2, the `Tree` is linked to the Evolutionary Graph component: when an individual is selected by the user, its tree representation is displayed.

4.2 Operation Panel

The `OperationPanel` component uses a set of `Tree` components to show the behaviour of genetic operators. It displays the parent individual(s) and the offspring, together with the information about their fitness and the genetic operator used. Users can select sub-trees in the offspring, and the corresponding sub-tree in the parent individual will be highlighted. This is possible using the information about *Parent1*, *Point1*, *Parent2* and *Point2* from the `EndOperator` event—*Parent2* and *Point2* only in the case of crossover. Given that each individual is displayed using `Tree` components, users can zoom-in/zoom-out and drag the tree around.

The mutation visualization is similar, the main difference is that there is only one parent and offspring. If a user selects a sub-tree that is not part of the parent individual (i.e. the

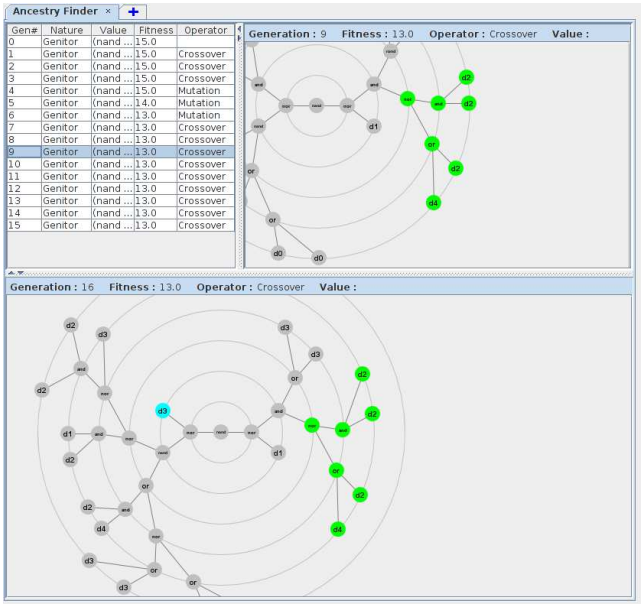

Figure 5: The `AncestryFinder`: After a sub-tree is selected in the bottom panel, the list of ancestor individuals is displayed in the top table, and they can then be visualised in the top-right panel.

randomly generated sub-tree), no sub-tree is highlighted in the parent individual.

The `OperationPanel` is also linked to the Evolutionary Graph component: when an individual is selected by the user, the `OperationPanel` shows the information about how the individual was created. In order to receive the notification when a user selects an individual, the `OperationPanel` is registered as a listener of the `Graph` component's view model:

```
OperationPanel panel = new OperationPanel();
graph.getViewModel().addGraphViewListener(panel);
monitor.add(panel, 0, 1);
```

4.3 Ancestry Finder

The `AncestryFinder` component takes advantage of the `Tree` component and allows a user to visualise where sub-trees originated. The idea is to allow the user to select a sub-tree from an individual and trace back the origin of the sub-tree. The principle is to look for the occurrence of the selected sub-tree of the individual in its parent(s) using the *Parent1* and *Parent2* information—the latter in the case of crossover—and repeating this procedure until the search reaches an individual of the initial population or when it is determined that the selected sub-tree originated from a mutation operator.

The `AncestryFinder` component is also registered as a listener of the `Graph` component, so when an individual is selected in the `Graph`, it is displayed using a `Tree` component at the bottom of the `AncestryFinder` component's panel. This allows the user to visualise the selected individual and select a sub-tree. After a sub-tree is selected, the list of ancestor individuals (all previous individuals that contain the sub-tree) is displayed, together with the information of their generation, fitness and the genetic operator. The user

can then visualise any of the ancestors by selecting individuals from the ancestor list—the ancestor list allows multiple selections.

5. PUTTING IT ALL TOGETHER

Since all the GUI components receive notifications from the run of the GP using the event mechanism, the only requirement to generate a graphical output is to create the GUI components objects and register them to the `Monitor`. Each component automatically registers itself to receive events from the `EventManager` and it retrieves the required data from stat objects.

The first step is to configure the GP algorithm. The listing below shows the sequence of statements required to configure EpochX to run the even-5 parity problem:

```
Config config = Config.getInstance();
// sets the default generational parameter values
config.set(Template.KEY,
    new GenerationalTemplate());
config.set(TreeFactory.MAX_DEPTH, 5);
config.set(TreeFactory.INITIAL_DEPTH, 3);

// the problem instance
Problem problem = new EvenParity(5);
config.set(FitnessEvaluator.FUNCTION, problem);

// operators
List<Operator> operators =
    new ArrayList<Operator>();
operators.add(new Mutation());
operators.add(new Crossover());
config.set(BranchedBreeder.OPERATORS, operators);

// termination criteria
List<TerminationCriteria> criteria =
   new ArrayList<TerminationCriteria>();
criteria.add(new MaximumGenerations());
config.set(
   GenerationalStrategy.TERMINATION_CRITERIA,
   criteria);
```

After the configuration statements, the GUI components can be created:

```
// Monitor frame with 1 row and 2 columns
Monitor monitor = new Monitor("Monitor", 1, 2);

Graph graph = new Graph();
monitor.add(graph, 0, 0);

OperationPanel panel = new OperationPanel();
graph.getViewModel().addGraphViewListener(panel);
monitor.add(panel, 0, 1);

AncestryFinder finder = new AncestryFinder();
graph.getViewModel().addGraphViewListener(finder);
monitor.add(finder, 0, 1);

// export the graph model to "backup.ser"
EventManager.getInstance().add(
   EndRun.class,
   new GraphModelWriter(graph.getModel(),
      "backup.ser"));
```

The above listing creates a `Monitor` frame with 1 row and 2 columns: on the left, it displays a `Graph` component (Evolutionary Graph); on the right, it will display a tabbed panel with the `OperationPanel` and `AncestryFinder` components. It will also export the `Graph`'s model to a file at the end of the run. Finally, we just need to run the GP algorithm:

```
Evolver evolver = new Evolver();
evolver.run();
```

6. CONCLUSION

In this paper we presented the Monitor package, which introduces GUI visualization components into the EpochX framework. These components make use of the event management and Stats system to provide a graphical representation of the progress of an evolutionary run. We discussed representation-independent components and also tree-representation components, which provide a graphical representation of tree-based individuals.

Representation-independent components included: `Table` and `Chart`, which are common facilities to follow the evolution of a GP algorithm in terms of fitness; and the `EvolutionaryGraph`, which is a more elaborated component showing the entire population and allowing the visualization of parent-child relationships and detailed information of each individual in the population. Tree-representation components included: `Tree` component, which shows individuals in a radial structure; the `OperationPanel` component, which allows the visualization of the effect of genetic operators (mutation and crossover) in the structure of the individuals; and the `AncestryFinder` component, which allows the user to select a sub-tree from an individual and trace back the origin of the sub-tree, navigating through parent-child (ancestry) relationships.

The Monitor package is highly customizable, making it easier to introduce rich graphical components. We are currently evaluating the implementation of specific components for other GP representations.

EpochX is available for download, including source code and documentation, from http://www.epochx.org/.

7. ACKNOWLEDGEMENTS

The authors gratefully acknowledge the financial support from the *EPSRC* grant EP/H020217/1. Loïc Vaseux thanks the *ERASMUS Student Mobility for Placements* and the *Conseil Général de Haute-Normandie* for their scholarships.

8. REFERENCES

[1] Jchart2d, precise visualization of data. http://jchart2d.sourceforge.net/usage.shtml, 2012.

[2] L. Beadle and C. G. Johnson. Semantic analysis of program initialisation in genetic programming. *Genetic Programming and Evolvable Machines*, 10(3):307–337, Sept. 2009.

[3] G. Book and N. Keshary. Radial tree graph drawing algorithm for representing large hierarchies. University of Connecticut, December 2001.

[4] G. Brunoro, G. Pappa, J. Palotti, and R. Melo-Minardi. Galapagos: Understanding genetic programming evolution. GECCO'11 Visualizing Evolution Competition, 2011.

[5] T. Castle and C. G. Johnson. Positional effect of crossover and mutation in grammatical evolution. In *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*. Springer, Apr. 2010.

[6] T. Castle and C. G. Johnson. Evolving high-level imperative program trees with strongly formed genetic programming. In *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012*, volume 7244 of *LNCS*, pages 1–12. Springer, Apr. 2012.

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

[8] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich. Opt4J - A Modular Framework for Meta-heuristic Optimization. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*, pages 1723–1730, Dublin, Ireland, 2011.

[9] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[10] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, Aug. 2001.

[11] F. Otero, T. Castle, and C. G. Johnson. Epochx: Genetic programming in java with statistics and event monitoring. In *GECCO'12 Companion*, Philadelphia, PA, USA, July 2012.

[12] F. Otero and C. G. Johnson. Automated problem decomposition for the boolean domain with genetic programming. In *Proceedings of EuroGP 2013, LNCS 7831*, pages 169–180, 2013.

[13] S. van Berkel, D. Turi, A. Pruteanu, and S. Dulman. Automatic discovery of algorithms for multi-agent systems. In *Proceedings of GECCO'12 Companion*, pages 337–344, 2012.

[14] S. Wagner. *Heuristic Optimization Software Systems – Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment*. PhD thesis, Institute for Formal Models and Verification, Johannes Kepler University Linz, Austria, 2009.

[15] P. Whigham. Grammatically-based genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, 1995.