Evolving the User Interface

Mark Withall
Dept. of Computer Science
Loughborough University
Loughborough
Leics. LE11 3TU
m.s.withall@lboro.ac.uk

Chris Hinde
Dept. of Computer Science
Loughborough University
Loughborough
Leics. LE11 3TU
c.j.hinde@lboro.ac.uk

Roger Stone
Dept. of Computer Science
Loughborough University
Loughborough
Leics. LE11 3TU
r.g.stone@lboro.ac.uk

Abstract

A method is presented for evolving Graphical User Interfaces using Genetic Algorithms. The fitness evaluation is based on a series of constraints, which must be met by the user interface. Examples are used to demonstrate the use of positional, style and functionality constraints and the final example shows the evolution of a complete (although simple) software application.

1 Introduction

In this paper a method for evolving Graphical User Interfaces (GUI) is presented. The method is loosely based on that used for evolving functions given in [13]. The method is applied to both desktop and web-based user interfaces.

Why evolve GUIs? Any interactive program requires some form of user interface. For a small program a simple text-based interface may be sufficient, but for larger systems (such as a word processor) a more complex user interface may be required. The evolution process allows some element of variety in the design of the interface as it is not a deterministic process. Furthermore, it is claimed that 50% of a project's implementation time is spent on the user interface[7]. In addition, if Genetic Programming is to be used for the creation of software, the user interface needs to be part of that evolution.

A number of systems have demonstrated the automatic generation of user interfaces from high-level specifications[3, 5, 9, 12]. Most of these systems are based on a data model specification. Lauridsen extends the approach to work with more abstract specification[6]. Other approaches include using declarative models[10] and conceptual graphs[4]. No work appears to have been done on using Genetic Algorithms (GA) or Genetic Programming (GP) to generate user interfaces.

The structure of this paper is as follows. Section 2 covers the required predefined aspects of the problem. Sections 3 and 4 describe how these are used to construct the representation for the user interface and the fitness function, used by the GA. Section 5 presents some example problems to show how the approach works in practice. Finally, a discussion on possible extensions to the method is given along with conclusions drawn from the work.

2 Requirements

What information is needed to be able to evolve a graphical user interface?

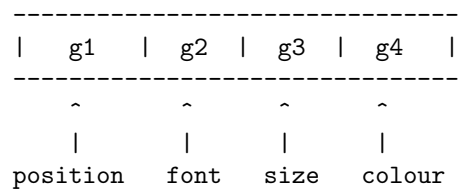
Firstly, some content to be manipulated is required. There are two types of content required: that associated directly with the interface, such as input widgets (buttons, text boxes, *etc*) and text, and that which provides the underlying functionality.

Secondly, some constraints on the use and implementation of this content are required. There are three main areas that can be constrained: the layout (the position of the widgets in relation to each other), the style (the fonts, colours, *etc*), and the functionality (what does each button do when pressed, where do the underlying functions get their input from and where do they put their output).

This information is needed to construct the representation and fitness function of the GA. The content is used to create the structure of the genome and the constraints are used as the basis of the fitness function.

3 Representation

The representation is based on the method for representing programs described in [13]. Each widget is represented by a fixed-length block of genes in the genome. Each of the genes represents one parameter of the widget (*eg* the font




```
e.g. 35,times,12pt,black
```

Figure 1: Example gene block

size). The whole genome is made up from all of the blocks of the genes, one for each of the widgets. Each widget always appears in the same place in the genome for each individual in the population.

For example, Figure 1 shows an example gene block, which gives the position, font, size, and colour of a widget. In a user interface language where there is a one-dimensional ordering of the widgets (*eg* HTML), a single position gene can be used. All the widgets are then sorted on this value to give their position within the interface. In the case where two widgets have the same positional value, the ordering is indeterminate.

Finally, a mapping from the genome to the actual program code is required. The genome can map to any language (or even multiple languages), such as HTML or Perl/Tk. From Figure 1, the gene block might map to (where its position in the program is decided as defined above):

```
.
.
<font face="times" size="12pt"
 color="black">Hello World</font>
.
.
```

In addition, there may be some functionality associated with the widget. Additional genes can be added to the gene blocks to represent the choice of function, where to get the input and where to put the output (see the example in Section 5.3).

4 Fitness Testing

It would be quite difficult to evaluate the entire user interface by executing the program and seeing how it looks and what it does. One possible solution to this problem is to evaluate all of the constraints individually by comparing the relevant parameters in the genome. The constraint scores are then combined to form the overall fitness of the individual. The combination operator in the examples shown in this paper is summation. Each constraint can be weighted to give greater importance to certain constraints if necessary, and other combination functions used.

5 Example Problems

To show how GUIs can be evolved, three simple examples are presented.

1. The evolution of a simple text editor interface. This demonstrates the layout constraints.
2. The evolution of a simple personal details web form. This adds style constraints.
3. A complete (but simple) application, which is an interface to the set of list evaluation and manipulation functions evolved in [13]. This demonstrates the functionality constraints in addition to layout and style.

5.1 A Text Editor

The first problem is a simple interface for a text editor. This demonstrates the use of positional constraints in the fitness function. The text editor interface is simply a text input area and a menu to select options (such as load and save). The constraints for this interface are mainly the relative positions of the items in the menu.

The list of widgets needed for the interface are (specified with a label, a type and an optional value):

- Title: title “TextEdit”
- Menu: menubar
- MenuFile: menulevel1 “File”
- MenuEdit: menulevel1 “Edit”
- MenuHelp: menulevel1 “Help”
- MenuFileNew: menulevel2 “New”
- MenuFileOpen: menulevel2 “Open”
- MenuFileSave: menulevel2 “Save”
- MenuFileExit: menulevel2 “Exit”
- MenuEditCopy: menulevel2 “Copy”
- MenuEditCut: menulevel2 “Cut”
- MenuEditPaste: menulevel2 “Paste”
- MenuHelpAbout: menulevel2 “About”

- Textarea: textarea

This list of items is used to construct the genome for the GA and is also used for the conversion between the genotype (in this case a list of integers) and the phenotype (which is a program in the Perl/Tk programming language).

Each gene-block in the genome, for this problem, is made up from only one gene. That gene specifies the position of the widget in the interface.

The constraints used by the fitness function are as follows:

- 'Title' must be first
- 'MenuFile' must be before 'MenuFileNew'
- 'MenuFileNew' must be before 'MenuFileOpen'
- 'MenuFileOpen' must be before 'MenuFileSave'
- 'MenuFileSave' must be before 'MenuFileExit'
- 'MenuEdit' must be before 'MenuEditCut'
- 'MenuEditCut' must be before 'MenuEditCopy'
- 'MenuEditCopy' must be before 'MenuEditPaste'
- 'MenuHelp' must be before 'MenuHelpAbout'
- 'Textarea' must be last

To achieve a maximum fitness value all of these constraints must be met. Although this specification of constraints is fairly loose, it would be possible to specify them in a more formal manner. For example, `i must be before j` might look something like (only the post-condition is given):

$$MustBeBefore(i : \mathbb{W}, j : \mathbb{W}) \triangleq pos_i < pos_j$$

where $\mathbb{W}$ is the set of all widgets and pos_i is the position attribute of the widget i. A conjunction of constraints forms the specification, which in turn can be used to create the fitness function.

In the case of this example, all fully fit individuals evolved by the GA will look identical. This is not, however, the case with all interfaces that can be evolved and, therefore, the unconstrained parts will allow for novel designs to be evolved.

Table 1: Results for the text editor example

Seed	Generation	Time
2	481	3s
3	60	1s
5	77	1s
7	118	1s
11	387	3s
13	227	2s
17	329	2s
19	189	1s
23	199	2s
29	87	1s

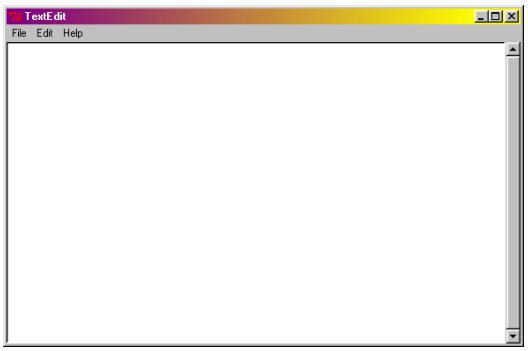


Figure 2: Text Editor GUI - Seed 2

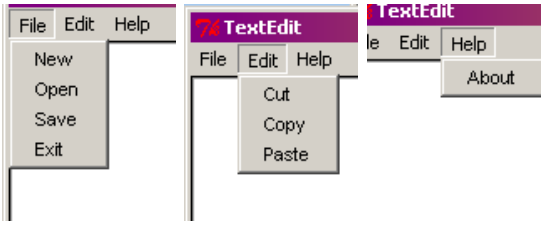


Figure 3: Text Editor GUI (The Menus) - Seed 2

Table 1 show the results of ten example runs and, as the problem is relatively simple, all runs produce a solution that completely satisfies the constraints within a very short time. The 'seed' column shows the seed used for the random number generator, to allow the experiments to be repeated. Figure 2 shows a user interface generated, but all fully fit user interfaces are constrained to be identical for this problem. The menus for the user interface are shown in Figure 3.

5.2 A Personal Details Web Form

The second problem is a "Personal Details" web form. This introduces style constraints into the fitness evaluation. The form contains a title and instructions, and a series of input boxes of different types with labels. Again, there is a set of positional constraints (such as labels must be immediately before the corresponding input). There are also style constraints that deal with fonts, colours, *etc.* For example, the title must be the largest font size and the labels must all be the same style.

The list of widgets used is as follows ('p' represents a paragraph of text):

- Title: title "Personal Details"
- Instructions: p "Enter your personal details in the form provided"
- NameLabel: p "Name"
- NameInput: text
- AddressLabel: p "Address"
- AddressInput: text
- TownLabel: p "Town"
- TownInput: text
- GenderLabel: p "Gender"
- GenderInput: select ("Male"/"Female")
- SubmitInput: submit
- ResetInput: reset

The positional constraints are:

- 'Title' must be first
- 'Title' must be immediately before 'Instructions'
- 'NameLabel' must be immediately before 'NameInput'
- 'AddressLabel' must be immediately before 'AddressInput'
- 'TownLabel' must be immediately before 'TownInput'
- 'GenderLabel' must be immediately before 'GenderInput'
- 'SubmitInput' must be immediately before 'ResetInput'
- 'NameInput' must be immediately before 'AddressLabel'
- 'AddressInput' must be immediately before 'TownLabel'
- 'TownInput' must be immediately before 'GenderLabel'
- 'ResetInput' must be last

Finally, the style constraints are:

- 'Title' must have the largest font size
- all labels must have the same style
- font colours must be much darker than the background colour

The gene-blocks for each widget consisted of five genes. The first gene represents the position of the widget (with the widgets being sorted on the gene value), and the remaining four genes represent the style attributes: size, colour, font, and alignment respectively. The size is chosen from the list (`12pt`, `14pt`, `16pt`, `18pt`, `24pt`, `32pt`), the colour is chosen from the list (`white`, `lightgray`, `gray`, `darkgray`, `black`), the font is chosen from the list (`serif`, `sans-serif`, `monospace`) and the alignment is chosen from the list (`left`, `right`, `center`). To convert the gene value into the attribute, the modulo of the gene value and the number of elements in the attribute list is taken. The position and attributes are then mapped to an HTML script.

Table 2 shows the results of ten example runs. It can be seen from the times that the problem is much harder than the previous example. The run that reached 50000 generations did not achieve a maximum fitness value within the allowed number of generations. However, the other nine runs did achieve a maximum fitness score. Figure 4 and Figure 5 show two user interfaces generated from different runs. The text boxes have a random alignment due to having

Table 2: Results for the personal details example

Seed	Generations	Time
2	45165	10m04s
3	7521	1m40s
5	8441	1m51s
7	50000	10m45s
11	12277	2m39s
13	5930	1m16s
17	2842	37s
19	2517	33s
23	6964	1m31s
29	7294	1m35s

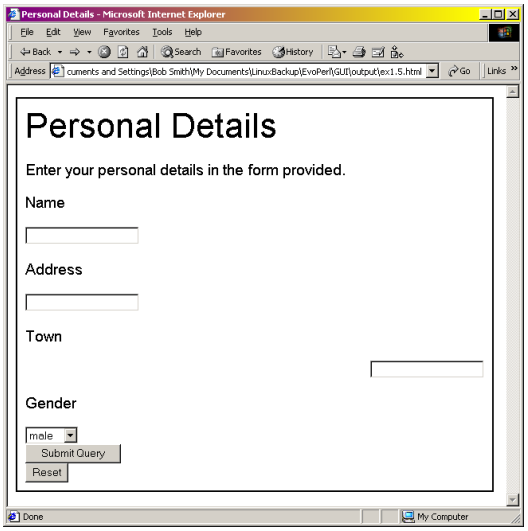


Figure 4: Personal Details GUI - Seed 5

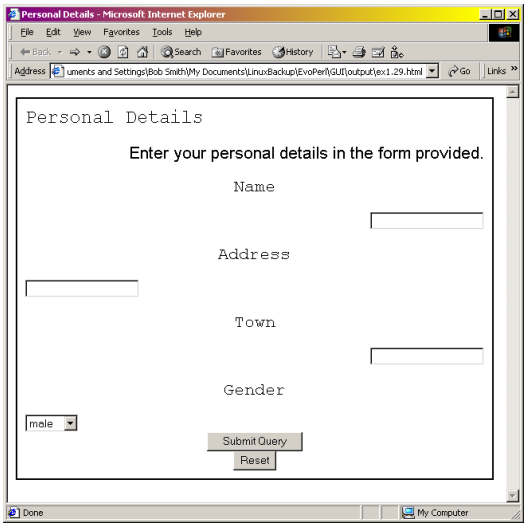


Figure 5: Personal Details GUI - Seed 29

no constraints. Additional constraints could be added to solve any problems with the interfaces evolved.

5.3 A Front-end for a set of List Functions

Finally, the third problem is a complete (although simple) application, which is an interface to a series of list evaluation and manipulation functions evolved in [13]. This introduces functionality to the interface. In addition, this example also demonstrates a possible approach to the problem of scalability by evolving component functions and then evolving an interface to connect them. The interface contains an input box, an output box, a list of functions, and a button. The positional and style constraints are similar to the previous problem, but now there are constraints on the functionality, which determine where the function gets its input and puts its output.

There are two choices when dealing with the functionality, either it can be hard-coded into the program or it can be evolved by the GA. The example shows the use of both options:

- Hard-coded functionality is demonstrated with the choice of function being chosen from the listbox (which returns the function name).
- Evolved functionality is demonstrated with the choice of widget that the input list for the function is taken from, and the choice of widget that the output list for the function is given to. Two extra genes are added to the gene-block from the previous example to allow this functionality to be evolved.

The list of widgets used is as follows:

- Title: title “sort”
- Instructions: p “Type in a list of integers, separated by spaces, into the Input box, select a function from the listbox and press the Run button.”
- List: listbox (“evolistmax”/ “evolistmin”/ “evosumlist”/ “evoavelist”/ “evoreverse”/ “evosort”)
- SourceLabel: p “Input”
- SourceInput: text
- TargetLabel: p “Output”
- TargetInput: text

Table 3: Results from the list front-end example

Seed	Generations	Time
2	1060	12s
3	310	4s
5	226	2s
7	337	4s
11	1419	16s
13	347	4s
17	1686	18s
19	2627	30s
23	424	5s
29	569	6s

- RunButton: button “Run!”

The positional constraints are:

- ‘Title’ must be first
- ‘Title’ must be immediately before ‘Instructions’
- ‘SourceLabel’ must be immediately before ‘SourceInput’
- ‘TargetLabel’ must be immediately before ‘TargetInput’
- ‘SourceLabel’ must be before ‘TargetLabel’
- ‘RunButton’ must be last

The style constraints are:

- ‘Title’ must have the largest font size
- all labels must have the same style
- font colours must be much darker than the background colour

And finally, the functionality constraints:

- ‘SourceInput’ must be the input for ‘RunButton’
- ‘TargetInput’ must be the output for ‘RunButton’

Table 3 shows the results of the ten runs. The results show that the problem was quite simple, with all runs producing a result with maximum fitness in a short time. This is due to the relatively small number of constraints. Figures 6 and 7 show examples of the user interfaces generated.

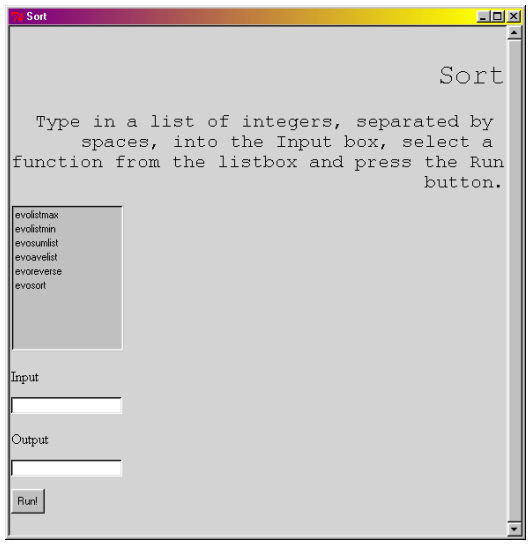


Figure 6: Sort GUI - Seed 13

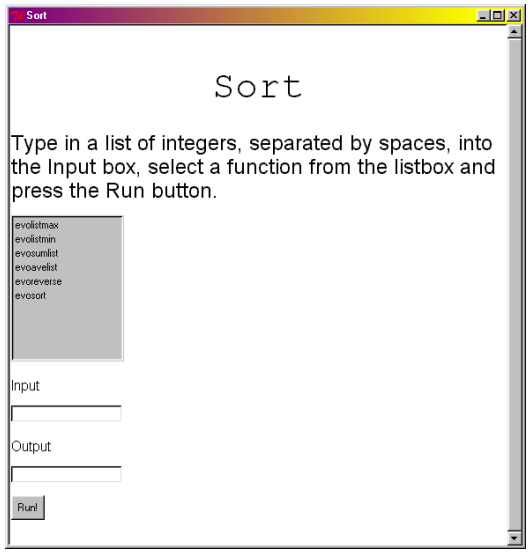


Figure 7: Sort GUI - Seed 23

6 Extensions

This method can easily be extended to cater for different types of user interface by introducing genes for the required attributes of the interface using the relevant constraints.

In addition, the constraints can be layered, so that there can be some global constraints that all interfaces must follow (such as those published by Apple[1] and Microsoft[8] or more generally following Sneiderman's 'Eight golden rules of dialog design'[11]). Then there can be a layer of constraints for specific groups of people. For example, dyslexic people prefer buttons instead of menus, whereas non-dyslexic people generally prefer the menu[2]. There can even be constraints to the level of the specific user. Finally, there are the problem specific constraints.

The content can also be abstracted, so instead of specifying a list box, a widget that chooses one item from many could be specified (which could be evolved to be a list box or a set of radio buttons, *etc*). An intermediate level of functionality can be evolved, which deals with changes to the user interface (to produce dynamic interfaces). For example, certain menu items might need to be disabled or enabled as a function of the state of the program after a particular action by the user (*eg* disable some menu items if there is no difference between saved and current versions).

The problem of scalability can be addressed by evolving the layout, style and functionality separately for more complex interfaces. In addition, each screen or window can be evolved separately when there are multiple screens in an interface. All the above contribute to improved performance of the evolution process on larger problems.

7 Summary and Conclusions

To summarise, a method for evolving Graphics User Interfaces has been presented. This method is loosely based on a previous method for evolving functions. This method can be applied to many different problems for many types of interfaces, in any required target language. In addition, the method can be used where there are contradictor constraints. The front-end for the list functions also addresses the problem of scalability as the interface and each of the functions are all evolved separately to create the whole system. The examples presented are very simple, and it is probably more difficult to specify the constraints for the GA fitness function than it would be to write the user interface in the first place. This may not be the case for more complex user interfaces. The list function front-end is probably the first complete software application to be evolved by Genetic Algorithm.

References

[1] Apple Computer, Inc. *Aqua User Interface Guidelines*, 2002.

[2] M. Carter. *Computer based writing support for dyslexic adults using language constraints*. PhD thesis, 2003.

[3] D. J. M. J. de Baar, J. D. Foley, and K. E. Mullet. Coupling application design and the user interface design. In *CHI '92 Conference Proceedings*, pages 259–266. ACM Press, 1992.

[4] O. Gerbé and M. Perron. Presentation definition language using conceptual graphs. In *Proceedings of PEIRCE Workbench*, pages 48–57, 1995.

[5] C. Janssen, A. Weisbecker, and J. Ziegler. Generating user interfaces from data models and dialogue net specifications. In *INTERCHI '93 Conference Proceedings*, pages 418–423. ACM Press, 1993.

[6] O. Lauridsen. Abstract specification of user interfaces. In *CHI '95 Conference Proceedings*. ACM Press, 1995.

[7] B. A. Meyers and M. B. Rossen. Survey on user interface programming. In *Proceedings of the Conference on Human Factors in Computing Systems*, 1992.

[8] Microsoft Corporation. *The Windows Interface Guidelines for Software Design: An Application Design Guide*. Microsoft Press, 1995.

[9] A. R. Puerta, H. Eriksson, J. H. Gennari, and M. A. Musen. Beyond data models for automated user interface generation. In *People and Computers IX, Proceedings of HCI '94*, pages 352–366. Cambridge University Press, 1994.

[10] E. Schlungbaum and T. Elwert. Automatic user interface generation from declarative models. In *CADUI '96*, 1996.

[11] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Publishing Company, 1987.

[12] J. Vanderdonckt and F. Bodart. Encapsulating knowledge for intelligent automatic induction object selection. In *INTERCHI '93 Conference Proceedings*, pages 424–429. ACM Press, 1993.

[13] M. S. Withall, C. J. Hinde, and R. G. Stone. Evolving perl. In *Late Breaking Papers, GECCO 2002*, 2002.