GUItar and FAgoo: Graphical interface for automata visualization, editing, and interaction*

André Almeida Nelma Moreira Rogério Reis
{bernarduh,nam,rvr}@ncc.up.pt

DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal

Abstract. GUItar is a graphical environment for graph visualization, editing, and interaction, that specially focuses in finite automata diagrams. The application incorporates mechanisms to facilitate the editing of these graphs. It also provides a style manager that allows the creation of rich state and arc styles to be used in the drawing of its objects. This style manager allows the system to cope with complex styles, broaden the application scope to graphical representations of other computational models like transducers or Turing machines. GUItar also has a foreign function call (FFC) mechanism for the easy integration of external modules and libraries like automata symbolic manipulators or graph drawing libraries. For automatic graph drawing we are developing FAgoo, a package that seeks to provide tools capable of finding pleasant graph drawings. FAgoo implements graph drawing algorithms that find embeddings which the user, with minimal manual changes, can adjust to its aesthetically taste. Both GUItar and FAgoo are on going projects licensed under GPL.

1 Introduction

GUItar [1] is a graphical environment tool for finite automata visualization and editing. This application incorporates mechanisms, like the auto adjustment of the nodes to avoid overlaps and the automatic positioning of the arcs which assist the user through the graph drawing and visualization. GUItar also provides powerful styling tools that not only allow the editing of node and arc styles but also allows the creation of new node structures. Furthermore we present the foreign function call (FFC) mechanism which is used to access external modules or libraries as FAdo and FAgoo [1].

FAdo is a tool for symbolic manipulation of formal languages and specially finite automata that can be incorporated with GUItar. Since most FAdo manipulations result in finite automata diagrams with no embedding, we are developing FAgoo which is a graph drawing library that specially focuses in that type of diagrams. Finite automata diagrams require additional aesthetic and graphical

* This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI, and by project ASA (PTDC/MAT/65481/2006).

constraints over other type of graphs. Finite automata diagrams, for example, are normally read from the left to the right therefore initial states are placed on the left and final states more to the right. FAgoo is a Python module written in C allowing us to maintain good performance and at the same time provide a high-level interface. In this paper we will describe the main components of GUItar as well as some algorithms already implemented in FAgoo.

2 Graph Drawing Libraries and Applications

There are several graph drawing libraries available with many layout algorithms for generic and specific types of graphs. Most of these libraries focus in a specific type of graphs in order to achieve better drawings. Restricting the type of graphs that an algorithm have to deal with, results in having graphs with particular properties which usually facilitates their drawings. Although there are many applications and libraries as aiSee [2], yED from yWorks [3], Open Graph Drawing Framework (OGDF) [4] and Graphviz [5] for automatic graph drawing, the algorithms implemented by these software do not fit the drawing conventions of finite automata drawings. This is because they were not specially designed to deal with finite automata drawings. JFLAP [6] is an application that aims to provide a way to experiment with formal languages representations, in particularly finite automata. Clearly JFLAP do not focus its work on the visualization and layout of the finite automata, thus, the available layout algorithms are very basic and simple.

3 GUItar

GUItar is an ongoing project which aims to provide a software tool for finite automata visualization and editing. Although GUItar specially focuses in finite automata diagrams, it supports other types of diagrams. Currently GUItar is implemented in Python and uses wxPython [7] graphical toolkit. The graphical interface basic frame is composed by a menu bar, a tool bar and a notebook. The menu bar is dynamically built from XML [8] configuration files. The notebook can handle multiple pages, each one containing a canvas. The canvas is implemented using the wxPython's FloatCanvas module which provides a set of graphical objects that can be bound with mouse events. To be able to draw labeled arcs, a new object called ArrowSpline had to be created.

3.1 Styles

In order to have a good platform for graph visualization and editing, we need to cope with a wide range of node and arc styles. GUItar provides a node and arc style manager that not only allows management of multiple styles, but also provides interactive creation and editing of complex node structures. The graphical representation of a node on GUItar consists in a set of objects from ellipses,

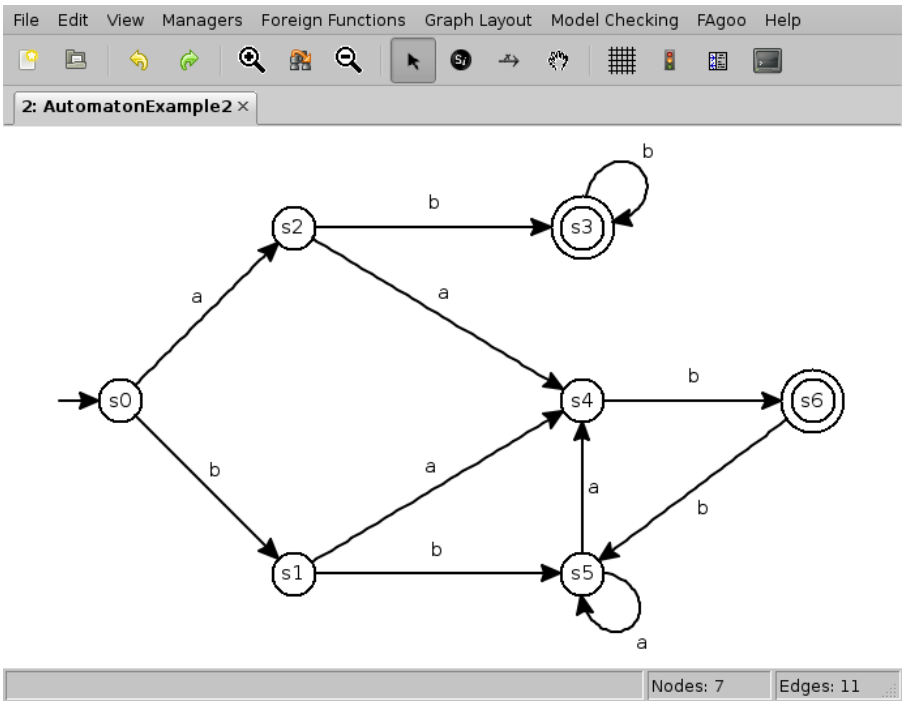

Fig. 1. An automaton created in GUItar.

rectangles, arrow splines and scaled texts. There must exist at least an ellipse or a rectangle to ensure that the node has a place to dock the incoming and outgoing arcs. It must also have one scaled text to place the node's label. This node structure allows the creation of complex nodes, enriching the graph visualization. For example, in finite automata diagrams we can represent final states by using two concentric ellipses, or initial states using a arrow and a ellipse. The Fig. 2 shows the node style manager of GUItar, editing a style that could be used to represent a state which is initial and final.

The node and arc labels can be either simple or compound. Simple labels are just text strings, while compound labels have custom fields with values specified by the user. The user can choose either to display or not each label field, and in this way, extra data can be associated to nodes and arcs.

These features can accommodate many purposes, expanding GUItar's scope to a larger range of graph types such as transducer diagrams, Turing machines, and others.

3.2 Visualization

Large graphs are difficult to visualize. If we are focusing in a part of a graph and we want to abstract ourselves from the rest of the graph we can select that part

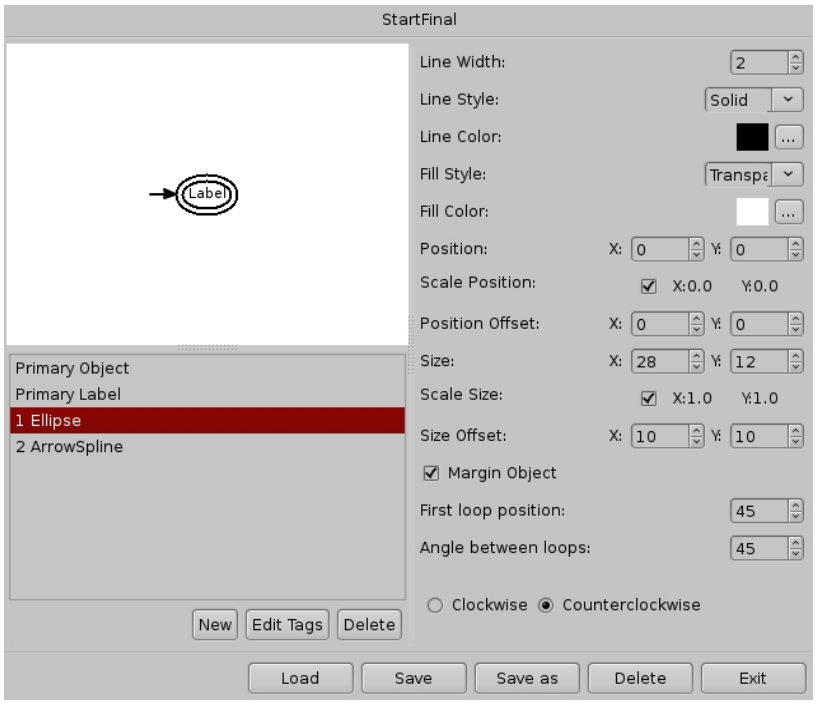

Fig. 2. GUItar's node style manager.

of the graph and ask the application to find a specific embedding that favours its visualization. There is also the case where we may want to have a overview and simplify some parts of the graph collapsing a subgraph into a node. One solution is to replace a subgraph by a node and transforming all the external arcs of the subgraph in the respective arcs to the node.

3.3 Graph Manipulation

In GUItar it is possible to collapse multiple arcs between two nodes. To collapse multiple arcs their labels must be merged. By default the concatenation operation is used, but other operations can be defined. As for the resulting style, if all arcs share the same style then that one is used. If different styles are present, one is arbitrarily chosen or the user is prompted to do it.

Two or more nodes can be merged into one. To do this there are three aspects to take in consideration: the labels, the styles, and the arcs. The labels' merging and the style selection are done as above. The arcs of the merging nodes are replaced by arcs to the resulting node, which can lead to the creation of multiple arcs. Further collapsing of these resulting arcs can then take place.

3.4 FFCs

We do not intend this project to be a new monolithic graph visualization and editing tool, but we see it more as a hub where graph manipulation libraries can, together, provide better visualization and manipulation tools. This is achieved by a FFC mechanism, using a Python interface to access the external tools (see Fig. 3). There are three types of FFC: module FFC, object FFC and interactive FFC. In the first case, the FFC calls a function directly from an external Python module. In the second case, it creates a foreign object and then calls methods of that object. In the last case, when a specified event occurs, the FFC triggers the respective handler function from an external Python module that will return a sequence of actions as script commands. FFCs require an XML configuration file that specifies the available methods. FFCs can create their own menu entries which makes its integration in GUItar smooth and practical. Most of the GUItar tools are implemented using FFCs that interface FAdo and FAgoo.

3.5 Animations

Animation can be a good way to illustrate a complex algorithm behavior. One application of interactive FFCs is algorithm animation. A simple algorithm as the one that finds a path between two nodes can be annotated with commands and events to animate its execution. These annotations allow to control the canvas behavior and its contents. To control the animation flow, GUItar can provide a set of interactive controls or the FFC can provide its own external interface. An example of a more complex animation is the deterministic finite automata minimization that uses nodes merging to illustrate some of the transformations during the algorithm execution.

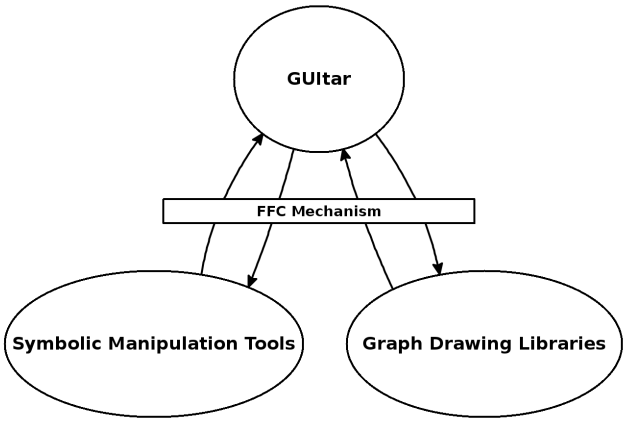

Fig. 3. A FFC mechanism overview.

3.6 Graph Classification

The GUItar classification mechanism allows to test if a graph belongs to a certain class by checking if the graph verifies a set of properties. These properties can test graphical properties (e.g. if arcs have arrows) or semantic properties (e.g. if a finite automaton is deterministic). A few of these methods are predefined in GUItar to check the most usual graphical properties of a graph. Access to external libraries with FFCs can be used to test graph properties, broaden the class range. Biconnectivity and planarity tests can be done, for example, using FAgoo.

A friendly interface is available for graph classification (see Fig. 4). This interface lists the graph properties and identifies the ones that are verified for the current graph. The user can create his own classes by stating the properties that the class must comply. It is also possible to export and import these class definitions.

Fig. 4. GUItar's interface for graph classification.

3.7 Semaphores

When editing a graph it can be useful to constraint the actions performed such that the resulting graph does not leave a certain class. The GUItar Semaphore tool assists this task by warning the user, or even restricting his actions. For example, suppose that we have a deterministic finite automata (DFA) as the result of some manipulation, and we want to edit it. We can enable the semaphore for DFAs to ensure that the changes that we apply to the graph do not compromise the DFA class definition.

New Semaphores can be created by extending the Semaphore base class and declaring them in a XML configuration file. An image of a traffic sign is associated to each semaphore which its light color represent the current state of the graph evaluation. There is also an image of a small padlock that when closed means that actions are restricted, i.e., do not allow actions that compromise the desired graph properties.

3.8 Import and Export

GUItar store its graphs using GUItarXML [9], which is an XML format specially designed for this application and based on GraphML [10]. GUItar also imports and exports to other formats, converting from and to GUItarXML. Currently the available exporting formats are GraphML, dot [11], Vaucanson-G [12] and FAdo. It is also possible to import from all these formats with the exception of Vaucanson-G. The Xport mechanism provides an easy way to add new export and import methods to GUItar. These methods can be coded in Python or use XSL transformations [13].

3.9 Object Library

Automata manipulation involves many operations with automata which result in large sets of new automata. What Object Library offers is a way of tracing these operations (methods) and all the objects involved. With this information it is possible to maintain an history of these operations and know the origins of each object, as well as recreate them from the original object. This information can be used to enrich an automata database by adding complementary information about the automata origins. Another feature is the possibility of create scripts with these operations, which the user can save and then apply to other objects. An interesting application for this tool is the creation of scripts with sequences of graph drawing algorithms, to generate specific layouts that then could be applied to several graphs.

4 FAgoo

Graph drawing is an active area of research with a lot of documented algorithms for generic and specific graph types. However, there are not many specifically designed for finite automata diagrams. To enhance the readability of each type of diagrams, they are normally drawn according to a set of conventional rules. A finite automata is better read if it flows from left to right. Initial states must, thus, be placed in the left and final states tend to be pushed to the right. The labels are another particularity of finite automata diagrams. Finite automata labels can be very complex and large. A single arc can have a label with several strings attached, each one being complex, like a regular expression. Another constraint is the arcs and its labels placement. Arcs from the left to the right are placed above the ones from the right to the left, with labels placed on their left side. These constraints benefit the readability of these diagrams. Finally the frequent occurrence of loops which is not so usual in another type graphs, is another characteristic of finite automata diagrams. Generic graph drawing algorithms usually discard loops during the layout process and arrange them in a final stage, but loops are frequent in finite automata diagrams and can have complex labels which hardens its positioning task.

4.1 Drawing Planar Graphs

When drawing a graph, edge crossing reduces its readability, thus, making this an important aspect to consider [14]. A planar graph can be drawn on a plane without edge crossing, in particular it can be drawn only using straight-line edges. The algorithm implemented for planarity test is the one presented by Hopcroft and Tarjan [15], which has a linear time execution and can be extended to either construct a planar embedding (if the graph is planar) or determine the Kuratowski subgraph (if the graph is non-planar) [16].

The implemented straight-line drawing algorithm assumes that the input graph is triangulated, i.e., every face has exactly three vertices. To triangulate a planar graph while minimizing the maximum degree, Kant presented a algorithm that is a good approximation of the optimal solution, but this algorithm takes as input a triconnected graph. Since FAgoo currently does not implements a triconectivity augmentation algorithm, the canonical triangulation algorithm presented by Kant [17] was implemented. This algorithm only requires the input graph to be biconnected and computes a canonical ordering while triangulating a planar graph. This algorithm was slightly modified to compute a left most canonical (lmc) ordering. This ordering is a generalization of the canonical ordering of de Fraysseix et al. [18] and it is needed for the straight-line drawing algorithm.

Most graph drawing algorithms require that the input graph to be biconnected, i.e., a connected graph that remains connected after the removal of any vertex. FAgoo implements algorithms to test a graph biconnectivity, that with a few modifications computes the graph biconnectivity tree (BC-Tree), and the biconnectivity augmentation. There are two types of nodes in a BC-Tree, the

B-Nodes and the C-Nodes. The B-Nodes represent the maximal biconnected subgraphs and the C-Nodes represent the cutvertices. There is an edge between a C-Node and a B-Node if that C-Node belongs to the biconnected component represented by the B-Node. The biconnectivity augmentation algorithm takes a planar embedding of each biconnected component of the graph and its BC-Tree to biconnect the graph while preserving its planarity. This is a simple linear time algorithm [17], which is an adaptation of the one presented by Read [19].

4.2 Drawing Non-Planar Graphs

Finding an embedding for a non-planar graph that minimizes its edge crossing is NP-hard [20]. One possible approach for non-planar graphs is to remove arcs from its Kuratowski subgraph, that can be found in linear time, until the graph is planar. Finding this minimal set of arcs is the hard task. Another approach for non-planar graphs is to find the maximum planar subgraph. Again, this problem is NP-hard [21]. These problems have been researched over the last 20 years and still do not have good solutions for generic graphs. As future work we intend to develop these approaches and implement them in FAgoo.

4.3 Subgraph Drawings

Some times it is better to visualize a portion of the graph instead of the whole graph. One way to do this in a straight-line drawing is to do a two step triangulation. The selected subgraph is first triangulated and then the rest of the graph is added and triangulated. This makes the selected subgraph to be drawn disregarding the rest of the graph. But this may not always be possible because of planarity constraints. In these cases another technique can be used to pop the subgraph from the whole graph. The subgraph is separately drawn and softer color tones are used in the rest of the graph.

4.4 Multi-arcs

As mentioned before FAgoo specially focuses in finite automata diagrams. These type of graphs often use multi-arcs in their representation. However general graph drawing algorithms do not support multi-graphs and simplify them in a early stage. This may have bad repercussions during the recovery the original graph. When the original graph is recovered multiple arcs may override other nodes or even arcs. A way to overcome this problem is to replace every multiple arc by a new node with arcs to the original arc source and target. The Fig. 5 illustrates this step. Then when recovering the original graph the created nodes are used as control points for the arcs splines. This way no multiple arcs will override other nodes or arcs. The graph planarity is also obviously not affected.

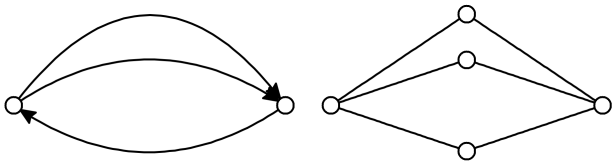

Fig. 5. Multi-arcs step illustration.

4.5 Force Directed Model

An interesting approach to the automatic graph drawing problem as been the simulation of forces. Increasingly force directed algorithms have been adopted by graph drawing libraries. The model used in FAgoo replaces the arcs with springs and the nodes with spheres. Between these spheres we introduce a repulsion force. The spheres are spread in a plane and the simulation stops when a equilibrium state is reached.

Reading directionality is achieved by fixing the initial states to the plane that is then lent to the right and gravity does the rest. This cause the other states to fall into to the right side of the initial states.

A graphical interface for this model is being developed. The idea of this interface is to allow the user to interact in real time with the ongoing simulation. The user can pause and resume the simulation to manually adjusting some components of the graph. For example, the user may want to fix the position of one or more nodes during a simulation or set specific strength values for some nodes and springs.

5 Conclusion

In this paper we presented GUItar as a tool for the visualization and editing of finite automata diagrams that combined with FAdo provides a potential graphical environment for automaton manipulation. We also presented the FFC mechanism that allows GUItar's expansion broaden the application scope to other type of graphs. We also presented FAgoo: a graph drawing library specialized in finite automata diagrams. FAgoo is integrated with GUItar using the FFC mechanism. Both GUItar and FAgoo are ongoing projects. FAgoo still needs improvements in some of the presented algorithms as well the development and implementation of many others.

References

1. FAdo Project: FAdo: tools for formal languages manipulation. `http://www.ncc.up.pt/FAdo` (Access date:12.06.2010)
2. aiSee Graph Layout Software: aiSee. `http://www.aisee.com/` (Access date:12.06.2010)

3. yWorks GmbH: yWorks. http://www.yworks.com/ (Access date:12.06.2010)
4. Chair of Algorithm Engineering, Juniorprofessorship of Algorithm Engineering, C.o.P.J., oreas GmbH: Open Graph Drawing Framework. http://www.ogdf.net/doku.php (Access date:12.06.2010)
5. Labs, A.R.: Graphviz - Graph Visualization Software. http://www.graphviz.org/ (Access date:12.06.2010)
6. Rodger, S.H., Finley, T.W.: JFLAP: An interactive formal languages and automata package. Jones & Bartlett Publishers (2006)
7. Smart, J., Roebling, R., Zeitlin, V., Dunn, R.: wxWidgets 2.6.3: A portable C++ and Python GUI toolkit. (2006)
8. WWW Consortium: XML specification WWW page. http://www.w3.org/TR/xml (Access date:12.06.2010)
9. Alves, J., Moreira, N., Reis, R.: XML description for automata manipulations. In Simões, A., Cruz, D., Ramalho, J.C., eds.: Actas XATA 2010, XML: aplicações e tecnologias associadas, ESEIG, Vila do Conde (2010) 77–88
10. GraphML Working Group: The GraphML file format. http://graphml.graphdrawing.org (Access date:12.06.2010)
11. Graph Visualization Software: The dot language. http://www.graphviz.org (Access date:12.06.2010)
12. Lombardy, S., Sakarovitch, J.: Vaucanson-G. http://igm.univ-mlv.fr/~lombardy (Access date:1.12.2009)
13. WWW Consortium: XSLT specification WWW page. http://www.w3.org/TR/xslt (Access date:12.06.2010)
14. Purchase, H.C., Cohen, R.F., James, M.I.: Validating graph drawing aesthetics. In: Graph Drawing. (1995) 435–446
15. Hopcroft, J.E., Tarjan, R.E.: Efficient planarity testing. J. ACM **21**(4) (1974) 549–568
16. Mehlhorn, K., Mutzel, P., Naher, S.: An implementation of the Hopcroft and Tarjan planarity test and embedding algorithm. Technical report, Research Report MPI-I-93-151, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 (1993)
17. Kant, G.: Algorithms for Drawing Planar Graphs. PhD thesis, Universiteit Utrecht, Faculteit Wiskunde en Informatica (1993)
18. de Fraysseix, H., Pach, J., Pollack, R.: How to draw a planar graph on a grid. Combinatorica **10**(1) (1990) 41–51
19. Read, R.C.: A new method for drawing a planar graph given the cyclic order of the edges at each vertex. Congressus Numerantium (56) (1987) 31–44
20. Garey, M.R., Johnson, D.S.: Crossing number is NP-complete. SIAM J. Algebraic Discrete Methods (4(3)) (1983) 312–316
21. Garey, M.R., Johnson, D.S.: Computers and intractability: A guide to the theory of NP-completeness (1979)