FAdo and GUItar: Tools for Automata Manipulation and Visualization*

André Almeida, Marco Almeida**, José Alves, Nelma Moreira, and Rogério Reis

DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
{bernarduh,sobuy,mfa,nam,rvr}@ncc.up.pt

Abstract. FAdo is an ongoing project which aims to provide a set of tools for symbolic manipulation of formal languages. To allow high-level programming with complex data structures, easy prototyping of algorithms, and portability (to use in computer grid systems for example), are its main features. Our main motivation is the theoretical and experimental research, but we have also in mind the construction of a pedagogical tool for teaching automata theory and formal languages. For the graphical visualization and interactive manipulation a new interface application, **GUItar**, is being developed. In this paper, we describe the main components of the **FAdo** system as well as the basics of the graphical interface and editor, the export/import filters and its generic interface with external systems, such as **FAdo**.

1 Introduction

The **FAdo** [pro08] project aims to provide an open source extensible high-performance software library for the symbolic manipulation of automata and other models of computation. A first implementation currently includes most standard operations for the manipulation of regular languages [MR05], a Turing machine simulator and parsing tools for context-free languages. An automata random generator package was released, based on previous theoretical work on enumeration and generation of initially connected deterministic finite automata (ICDFA) [AMR07]. Although there are several software packages for the symbolic manipulation of formal languages they either are not open source, have restricted purposes, or are no longer being maintained. Examples include: *Grail+* [RW94, Yu09], *Automate* [CH91], *Amore* [JPTW90], *Fire Station* [FW09] and *OpenFst* [Ril09]. An exception to this is the *Vaucanson* package [LRGS04] whose basic structures, due to its orientation to more algebraic applications of automata, are too heavy for the combinatorial and algorithmic simulations we think useful for complexity studies of formal languages. *JFLAP* [RF06] is

* This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI, and by project ASA (PTDC/MAT/65481/2006).
** Marco Almeida is funded by FCT grant SFRH/BD/27726/2006.

S. Maneth (Ed.): CIAA 2009, LNCS 5642, pp. 65–74, 2009.

a specialized pedagogical tool with an extensive coverage of formal language topics taught in undergraduate computer science courses. The possibility of interactively experimenting with the construction proofs is a major feature of this system. The **FAdo** system was first developed for pedagogical purposes. However the necessity of easily prototyping new algorithms, testing algorithm performance with large datasets, and the combinatorial nature of formal languages representations led us to continue **FAdo** development. The use of *Python*, a high-level object-oriented language with high-level data types and dynamic typing, ensures a system which is modular, extensible, clearly and easily implemented, and portable. Specialized and optimized data structures and performance critical algorithms may be written in a low-level language like *C*, and easily interfaced with *Python*, via the *Cython* language extension [BB09]. Here, we will describe the main components of the **FAdo** system for regular languages manipulation.

GUItar is a visualization software tool for various types of automata (standard, weighted, pushdown, transducers, Turing machines, etc.). Its purposes include automatic and assisted diagram drawing, algorithm animation, interactive editing and export/import filters. Automatic graph drawing has been a very active research area and several commercial software packages are now available for general and specific applications (database design, information systems, bioinformatics, social networks, etc.) [BERT99, Gra08, JM04]. In contrast, automata diagrams (labelled multi-digraphs) require additional aesthetics and graphical constraints: left-to-right reading, initial states on the left and final states on the right, edge shapes and label placements, etc. We intend to design and implement tools for automatic drawing of automata diagrams according to common accepted aesthetics principles. As a first step, in this paper, we describe the basic **GUItar** framework that includes assisted diagram drawing, interactive editing, and export/import filters.

2 FAdo: Tools for Regular Languages Manipulation

Regular languages can be represented by regular expressions (r.e.) or finite automata, among other formalisms. Finite automata may be deterministic (DFA) or non-deterministic (NFA). In **FAdo** these representations are implemented as *Python* classes, as presented in Figure 1. The class *FA* implements the basic structure of a finite automaton shared by DFAs and NFAs. This class also provides methods for manipulating these structures. The class *DFA* and *NFA* implements DFAs and NFAs, respectively. The class *EFA* implements generalized NFAs that are used in the conversion between finite automata and r.e. There are two representations for r.e.: the class *RE* implements, in a object-oriented manner, the usual inductive definition (it is elegant, but not efficient) and the class *ACIRE* implements irreducible regular expressions modulo *ACIA*, i.e., *associativity* of the concatenation and disjunction, *commutativity* of the disjunction, and *idempotence* of both disjunction and Kleene star operations. Disjunctions are represented as sets, which are efficiently implemented in *Python*. Concatenated r.e. are kept in an ordered list. The idempotence of the Kleene star

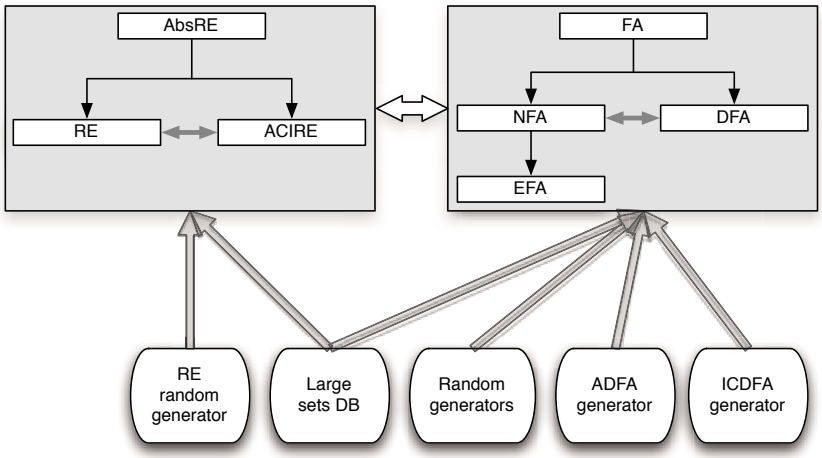

Fig. 1. FAdo classes for regular languages

is assured by not allowing double stared r.e. Whether or not a r.e. accepts the empty word is tabulated as a *ACIRE* attribute, to avoid unnecessary recursive calls. Elementary regular languages operations as union, intersection, concatenation, complementation and reverse are implemented for each class. Several conversions between these representations are implemented: $NFA \rightarrow DFA$: subset construction; $NFA \rightarrow RE$: recursive method; $EFA \rightarrow RE$: state elimination, with possible choice of state orderings; $RE \rightarrow NFA$: Thompson method, Glushkov method, follow, Brzozowski, and partial derivatives.

For DFAs several minimization algorithms are available (some with C implementations): Moore, Hopcroft, incremental algorithms of Watson and Daciuk. Brzozowski minimization is available for NFAs. Language equivalence of two DFAs can be determined by reducing their correspondent minimal DFA to a canonical form [AMR07], or by the Hopcroft and Karp algorithm. Language equivalence of two r.e. is implemented in the *ACIRE* class using variants of a rewrite system [AMR08a]. The class *ACIRE* has also several simplification methods for r.e.

2.1 Generators and Random Samples

We have designed and implemented several exact and random generators for some classes of automata and regular expressions. An exact and a uniform random generator are available for ICDFAs [AMR07]. Based on new canonical forms we also developed exact generators for acyclic (trim) deterministic finite automata (ADFA)[AMR08b], and for minimal ADFA (MADFA) [AMR08c]. For the uniform generation of random r.e. we implemented the method described by Mairson [Mai94] for the generation of context-free languages. Random (non-uniform) generators for NFAs that allow to generate initially connected NFAs (with one initial state) and to control the transition density are also implemented.

For a given number of states and symbols, the number of DFAs grows in a way that experimental tests over the complete universe quickly become impractical [AMR07]. For statistical analysis (or experimental results), a subset of manageable size from which we can make inferences or extrapolations to the whole universe may be used.

As the probability of any individual member of the universe being selected is exactly the same as any other individual member, a uniform random generator produces a true, unbiased, random sample. In order to have a reasonable sized (enough for statistically significant results), consistent, random sample readily available, we designed and implemented an SQL database to store the uniformly generated DFAs (and r.e.). We used the PostgreSQL open source relational database system [DBM08] to store the random samples of both DFAs and r.e.

Database. The ICDFAs database keeps and makes available random samples of automata with $n \in \{10, 20, \ldots, 90, 100\}$ states, each over an alphabet of $k \in \{2, 3, 4, \ldots, 18, 20, 25, 30, \ldots, 45, 50\}$ symbols. Besides the automaton structure, the database stores some properties such as minimality, being trimmed, acyclic, etc. This allows to obtain, with a simple SQL query, some automata datasets with specific properties. For efficiency reasons, besides its unique string representation [AMR07], the database is used to store the pre-parsed internal **FAdo** representation of each ICDFA. This avoids the need to parse an automaton's description every single time we need to manipulate it. By similar reasons, each automaton's final states set is stored in two different ways: as a comma separated list of integers and as a bitmap.

REs Database. The r.e. database is similar to the ones pertaining to finite automata. Pre-parsed representations of each object is kept in the database, both in the *ACIRE* and *RE* representation, to avoid overhead parsing time in any algorithm process.

3 GUItar: Interactive Visualization

The **GUItar** graphical interface allows the interactive visualization of generic graph diagrams and the execution of external graph manipulation tools. It is implemented with the *wxPython* [SRZD06] graphical toolkit. Figure 2 shows the interactive diagram editor. The basic frame has a *menu bar*, a *tool bar*, and a notebook that manipulates multiple pages. The *menu bar* and the *tool bar* are dynamically built from *XML* [Con08a] configuration files and event handler files, allowing an easy extensibility and modularity. Each notebook page contains a canvas for diagram drawing and manipulation. The canvas is based on the *wxPython*'s *Floatcanvas component* [Bar08] which allows to draw and to interact with graphic objects. It provides zooming, panning and binding mouse clicks on object to callbacks. It allows the addition of new objects and to alter its interactive behavior. To draw graph transitions a new *FloatCanvas* object called *ArrowSpline* was created. This object defines splines with or without arrow heads. It allows the access to the spline interpolation points, which was not

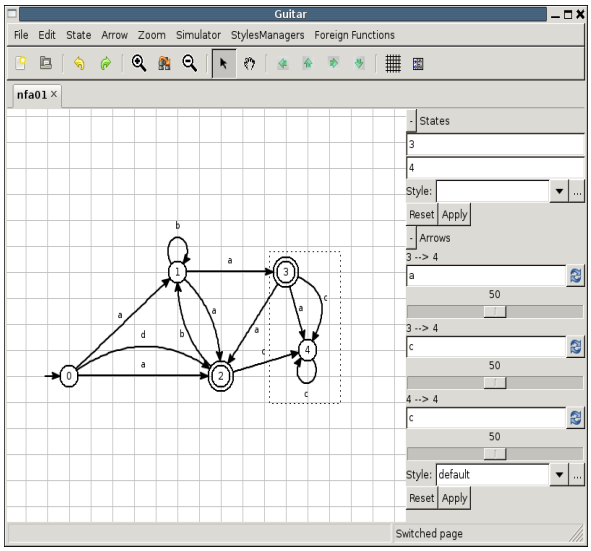

Fig. 2. GUItar graphical interface

possible in the native implementation. The main classes of **GUItar** are presented in Figure 3, and are summarized in the next subsections.

3.1 Drawing a Graph

A graph is defined by a set of nodes and a set of edges. The class *DrawGraph* allows the display and the editing of a graph diagram, and its main components are a canvas, a set of node objects, a set of edge objects and a grid. Nodes and edges can be added, edited, moved or deleted. Node labels can be automatically generated according to a given specification. The grid uses a general coordinate system to manage node positions and prevent objects to overlap. Each object can occupy several grid cells. To assist diagram editing a specialized graphical user interface (GUI) mode, a draw assistant and an undo/redo manager were implemented. Objects properties can be inspected and changed in the *properties panel*.

Nodes. The *Node* class has an identifier (ID), a position, canvas objects and a style. This class has methods to change node position and to determine borders for docking edges.

Edges. The *Edge* class has an ID, an origin and an target nodes, a canvas *ArrowSpline* object, and a label object (with side and position).This class has methods to edit *ArrowSpline* control points, change nodes dock points and change label location.

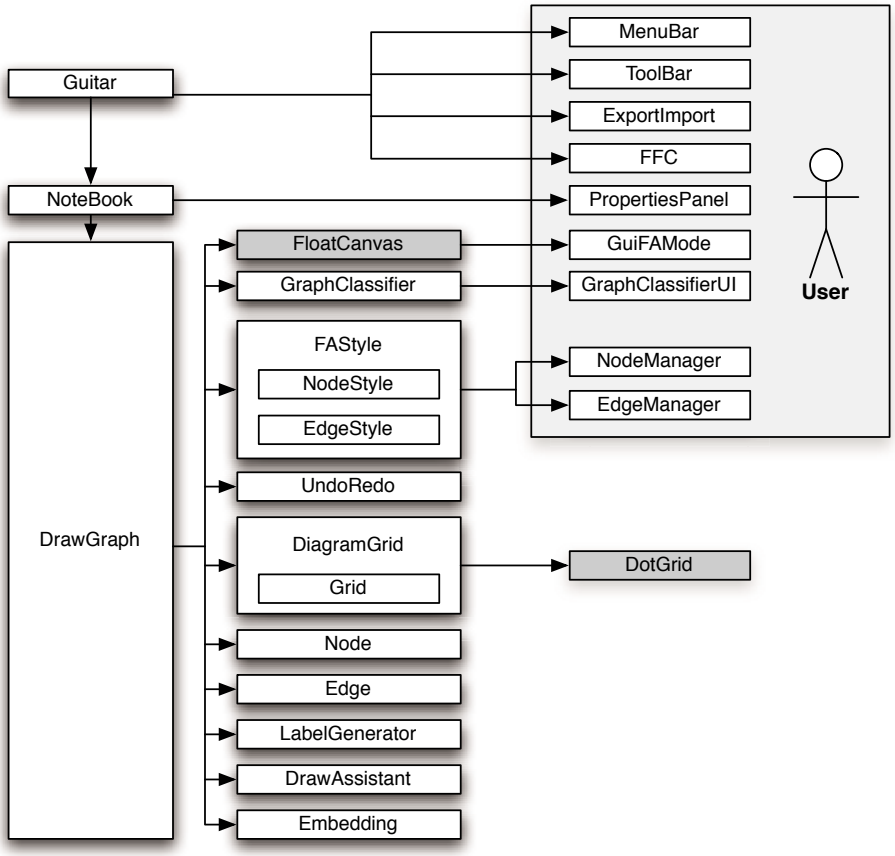

Fig. 3. A **GUItar** overview

Labels. A label can be simple (text string) or composed of several components.

Embeddings. The embedding is the layout of the nodes and edges in the plane. Currently a integer coordinate embedding is provided.

Editing Mode. The *GUIFAMode* class implements an user interface that allows several interactions with the graphical objects, essentially mouse based events, such as addition, deletion, selection, or movement of objects, as well as activation of pop-up menus. It also provides movement in the canvas viewport. The *DrawAssistant* class helps to place the edges and the loops. The edges can be edited by dragging their control points freely or using stepwise movements. To support undo and redo actions, the Undo/Redo manager assigns an ID to each kind of action, a method that handles the undo event, and the ID of the reverse action. The handler method receives as an argument the information needed to undo/redo the action. For each performed action, its ID and the information

that the Undo handler method needs are pushed into the Undo stack. The Undo and the Redo methods pop an action from the stack and call the handler method with the appropriated information.

Complex style managers. In general, automata diagrams provide several graphical information on state or transition representations. For instance, an initial state representation can have a side arrow, or a final state representation can have a doubled line border. Instead of having a few special styles built-in, **GUItar** provides a *Node Style Manager* that allows the construction of node styles with complex graphical objects. A node style can have several graphic objects, as components. Two of these are mandatory: the *primary* object and the *primary* label. Primary objects must be ellipses or rectangles, and they ensure that there is always a docking object for the edges. The primary label must be text. For each object, its usual style properties such as line color, line width, line style, fill color, fill style, sizes, fonts, etc. can be defined. A node style can be previewed while it is being defined (or edited), and saved in the **GUItar** internal database. A set of tags (key/value pairs) may also be associated with each node style. The *Edge Style Manager* permits the definition of edge styles. An edge style is characterized by the graphical properties of the edge's *ArrowSpline* canvas object. It is possible to specify the number of heads and their shapes, line style properties, and loop properties.

Graph Classifier. The *GraphClassifer* class allows the definition of graph classes by specifying graphic properties of each object. The *GraphClassificationUI* class provides an user interface to visualize and to create new classes. *Graph*, *digraph*, or *multidigraph* are the default classes.

Automatic graph drawing. A simple layout algorithm for visualizing graphs without any embedding information is implemented. An automatic placement based on physical forces simulation is also available.

3.2 Foreign Function Calls

GUItar provides a generic *foreign function calls* (FFC) interface between the diagram graphical editor and external manipulation tools, as the **FAdo** toolkit. The FFCs have two components: a description on a *XML* configuration file and a *Python* module. The description includes the module path and the methods that will be imported by **GUItar**. Each method must have a *name*, a *return type*, and, for each argument its *type* and a possible *default value*. Each module may have a menu in the main **GUItar**'s frame, or be accessed from a general FFC menu. At startup, **GUItar** loads the FFC configurations and builds the FFC menus.

3.3 Export/Import

Diagram descriptions and embeddings are saved in a *XML* format that was defined as a dialect of the *GraphML* language [Gro08]. *GraphML* is a simple

language to describe the structural properties of a graph and has a flexible extension mechanism to add application-specific data. Extensions are provided by a key/data mechanism that can be added to each graph element. For efficiency reasons, for the **GUItar** internal information our dialect encodes this mechanism directly. A fragment of the **GUItar** *Relax NG* schema, is presented below, where `diag_data` represent the embedding information, and `draw_data` correspond to general drawing information.

```
include "styles.rnc"
guitar = element guitar {
  attribute version {text},
  graph*
}
graph = element graph {
  attribute id {text},
  element node {
    attribute id {text},
    label,
    node_diag,
    node_draw,
    node_automata
}*,
  element edge {
    attribute id {text},
    attribute source {text},
    attribute target {text},
    label,
    edge_diag,
    edge_draw
}*,
  graph_diag,
  graph_class,
  style*
}
node_diag = element diag_data {
    attribute x {text},
    attribute y {text}}
node_draw = element draw_data {
    attribute style {text},
    attribute x {text},
    attribute y {text} }
node_auto = element auto_data {
    attribute initial {1 | 0},
    attribute final {1 | 0} }
edge_draw = element draw_data {
    attribute style {text},
    element point{
        attribute x {text},
        attribute y {text}} * }
label = element label {
   attribute type {"sim"|"com"},
   (dict*|text),
   label_draw }
```

GUItar exports its objects in three other formats: basic *GraphML*, *dot* and *Vaucanson-g* [LS08]. **GUItar** can also import from *GraphML* and **FAdo** automata format. These export/import methods are implemented as *XSLT* transformations [Con08b] from the **GUItar** format. We are developing *XSLT* transformations for the *fsmxml* format [Gro09].

4 Conclusions

The development of a solid and reliable symbolic manipulation package for formal languages is not a simple task. Being written in a high-level programming language and kept in an free software license promotes its usability by the scientific community. Visualization tools, and specially automatic drawing of automata diagrams, are challenging and important for both research and pedagogical purposes.

References

[AMR07] Almeida, M., Moreira, N., Reis, R.: Enumeration and generation with a string automata representation. Theoret. Comput. Sci. 387(2), 93–102 (2007)

[AMR08a] Almeida, M., Moreira, N., Reis, R.: Antimirov and Mosses's rewrite system revisited. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 46–56. Springer, Heidelberg (2008)

[AMR08b] Almeida, M., Moreira, N., Reis, R.: Exact generation of acyclic deterministic finite automata. In: DCFS 2008, Charlottetown, Canada (2008)

[AMR08c] Almeida, M., Moreira, N., Reis, R.: Exact generation of minimal acyclic deterministic finite automata. I. J. of F. of Com. Sci. 19(4), 751–765 (2008)

[Bar08] Barker, C.: Floatcanvas, `http://morticia.cs.dal.ca/FloatCanvas/` (access date: 1.12.2008)

[BB09] Behnel, S., Bradshaw, R.: Cython: C-extensions for Python, `http://www.cython.org/` (access date: 03.01.2009)

[BERT99] Battista, G., Eades, P., Tamassia, R., Tolli, I.G.: Graph Drawing, Algorithms for the Visualisation of Graphs. Prentice Hall, Englewood Cliffs (1999)

[CH91] Champarnaud, J.M., Hanset, G.: AUTOMATE, a computing package for automata and finite semigroups. J. of Symb. Comput. 12, 197–220 (1991)

[Con08a] World Wide Web Consortium. XML specification WWW page, `http://www.w3.org/TR/xml` (access date: 1.12.2008)

[Con08b] World Wide Web Consortium. XSLT specification WWW page, `http://www.w3.org/TR/xslt` (access date: 1.12.2008)

[DBM08] PostgreSQL DBMS. PostgreSQL website, `http://www.postgressql.org` (access date: 1.12.2008)

[FW09] Frishert, M., Watson, B.W.: Fire Station, `http://www.fastar.org/` (access date: 1.4.2009)

[Gra08] Graphviz — Graph Visualization Software. The dot language, `http://www.graphviz.org/` (access date: 1.12.2008)

[Gro08] GraphML Working Group. Graphml file format, `http://graphml.graphdrawing.org/` (access date: 01.12.2008)

[Gro09] Vaucanson Group. FSMXML format, `http://www.lrde.epita.fr/cgi-bin/twiki/view/Vaucanson/XML` (access date: 1.3.2009)

[JM04] Jünger, M., Mutzel, P. (eds.): Graph Drawing Software. Mathematics and visualization. Springer, Heidelberg (2004)

[JPTW90] Jansen, V., Potthoff, A., Thomas, W., Wermuth, U.: A short guide to the AMoRE system. Aachener informatik-berichte (90) 02, Lehrstuhl fur Informatik II, Universitat Aachen (January 1990)

[LRGS04] Lombardy, S., Régis-Gianas, Y., Sakarovitch, J.: Introducing Vaucanson. Theoret. Comput. Sci. 328, 77–96 (2004)

[LS08] Lombardy, S., Sakarovitch, J., Vaucanson, G.: `http://igm.univ-mlv.fr/~lombardy/` (access date: 1.12.2008)

[Mai94] Mairson, H.G.: Generating words in a context-free language uniformly at random. Information Processing Letters 49, 95–99 (1994)

[MR05] Moreira, N., Reis, R.: Interactive manipulation of regular objects with FAdo. In: ITiCSE 2005, pp. 335–339. ACM, New York (2005)

[pro08] FAdo project. FAdo: tools for formal languages manipulation, http://www.ncc.up.pt/FAdo (access date: 1.12.2008)
[RF06] Rodger, S., Finlea, T.: JFLAP - An Interactive Formal Languages and Automata Package. Jones and Bartlett (2006)
[Ril09] Riley, M.: OpenFst, http://www.openfst.org (access date: 1.4.2009)
[RW94] Raymond, D., Wood, D.: Grail: A C++ Library for automata and expressions. J. Symb. Comp. 17(4), 341–350 (1994)
[SRZD06] Smart, J., Roebling, R., Zeitlin, V., Dunn, R.: wxWidgets 2.6.3: A portable C++ and Python GUI toolkit (2006)
[Yu09] Yu, S.: Grail+, http://www.csd.uwo.ca/Research/grail/ (access date: 1.3.2009)