XML Description for Automata Manipulations[⋆]

José Alves Nelma Moreira Rogério Reis
{sobuy,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 visualization software tool for various types of automata (standard, weighted, pushdown, transducers, Turing machines, etc.). It provides interactive manipulation of diagrams, comprehensive graphic style creation, multiple export/import filters, and a generic *foreign function calls* (FFC) interface with external systems. In this paper we describe GUItar's XML framework and show how it allows for extensibility, modularity and interoperability.

1 Introduction

FAdo [1,2,3] is an ongoing project which aims to provide a set of tools for symbolic manipulation of automata and other models of computation. For diagram graphical visualization and interactive manipulation the GUItar application [2] is being developed in Python [4] and using the wxPython [5] graphical toolkit. GUItar provides assisted drawing, interactive manipulation of diagrams, comprehensive graphic style creation and manipulation facilities, multiple export/import filters and extensibility, mainly through a generic *foreign function call* (FFC) mechanism. Figure 1 shows the GUItar architecture. The basic frame of its interactive diagram editor has a notebook that manipulates multiple pages, which one containing a canvas for diagram drawing and manipulation.

In this paper we describe GUItar's XML framework and show how it allows for extensibility, modularity and interoperability. In Section 2 we present GUItarXML, an XML format for the description of diagrams that allow several information layers. This format is the base for the export/import methods described in Section 3, where a generic mechanism for add new methods is also presented. Section 4 presents the FFC's configuration and manipulation mechanisms. These allow the interoperability with external software tools. As an example we consider the interface with the FAdo toolkit [2]. Finally in Section 5 we briefly present the *object library* which will allow the dynamic construction of an automata database.

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

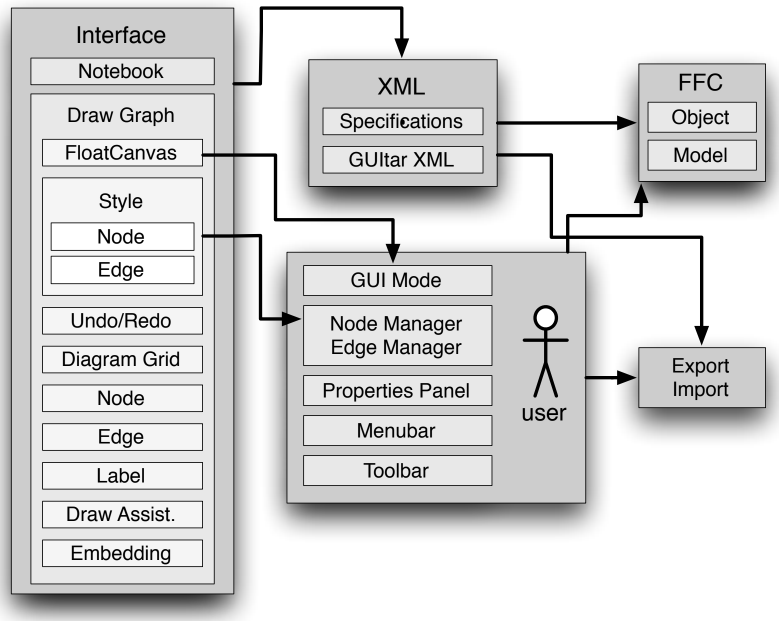


Fig. 1. GUItar architecture

2 GUItarXML

GUItarXML is an XML [6] format for the description of diagrams originally designed for the GUItar application, but that aims to be as generic as possible. It is based on GraphML [7], a simple language to describe the structural properties of graphs. Although GraphML has an extension mechanism based on key/value pairs to encode application specific data, because of efficiency and clarity reasons, we chose not to use it. Instead, we encode that data directly as new elements. GUItarXML describes graphs and digraphs or graph-like diagrams, such as automata, and contains graphical information, style information, and specific automata information. A fragment of the GUItarXML Relax NG schema [8] is presented in Figure 2. Diagrams are composed of a set of nodes connected by a set of edges. They are encoded in `graph` elements and there can be an arbitrary number of diagrams per GUItarXML document. The `graph` element can contain an arbitrary number of `node` and `edge` elements, each of them identified by an `id` attribute that must be unique. Edges have the `source` and `target` attributes that are the `id`s of the endpoints of the edge. Although this information is enough for some applications, some additional data may be required. Automata, for instance, require states and edges to have labels. The nodes and edges have a `label` element. The labels can be either simple or compound. Simple labels are just text strings. Compound labels have fields that can assume user-specified values. This is used,

```
include "styles.rnc"
include "defaults.rnc"

start = element guitarxml {
  attribute version {text},
  graph*,
  style*,
  state_object_group*
}
graph = element graph {
  attribute id {text},
  node*,
  edge*,
  graph_automata?,
  defaults?
}
node = element node{
  attribute id {text},
  node_diag?,
  node_draw?,
  label?,
  node_aut?
}
node_diag = element
    diagram_data {
  attribute x {text},
  attribute y {text}
}
node_draw = element draw_data {
  attribute obgroup {text},
  attribute x {xsd:long},
  attribute y {xsd:long},
  attribute scalex {xsd:long}?,
  attribute scaley {xsd:long}?
}
node_aut = element
  automata_data {
  attribute initial {"1"|"0"}?,
  attribute final {"1"|"0"}?
}
```

```
edge = element edge{
  attribute id {text},
  attribute source {text},
  attribute target {text},
  element diagram_data{...},
  edge_draw?,
  label?
}
edge_draw = element draw_data {
  attribute arrowlinestyle {text}?
  attribute head1style {text}?,
  attribute head2style {text}?,
  attribute numheads {"0"|"1"|"2"}?,
  attribute swapheads {"1"|"0"}?,
  (point*)?,
  ...
}
label = element label {
   attribute type {"simp"|"comp"},
   attribute text {text}?,
   attribute layout {text},
   attribute style {text}?,
   dict*
}
dict = element dict {
    attribute key {text},
    attribute value {text}
}
graph_automata = element
   automata_data{
  element sigma{
   element symbol{
     attribute value {text}
    }*
  }?,
  element classification{
    element class{
      attribute value {text}
    }*
  }?
}
```

Fig. 2. A fragment of the Relax NG schema for diagrams.

for example, with weighted automata, transducers or Turing machines. Figure 3 shows a compound label with two fields: label and weight. These fields have the values a and 0.3, respectively, so the final label value is $a : 0.3$.

```
<label type="Compound" layout="$label : $weight"
    style="style1">
  <dict key="label" value="a"/>
  <dict key="weight" value="0.3"/>
</label>
```

Fig. 3. Example of a compound label.

Embedding and drawing information are described in the `diagram_data` and the `draw_data` elements, respectively. For the nodes, `diagram_data` contains the embedding coordinates. The `draw_data` elements contain graphical data such as the graphical object styles and drawing properties such as the node's draw coordinates and scale. The `automata_data` elements describe specific automata properties, like boolean attributes `final` and `initial` used to indicate, for nodes, if a state is final or initial, respectively. Within `graph` element, general automata information (if applicable) such as the alphabet is specified. The `graph` elements also have the `defaults` element that contains style the default values.

```
<graph>
  ...
<edge ...>
      ...
    <draw_data linestyle="red" head1style="red"
        numberofheads="1" ...>
      ...
</edge>
  ...
</graph>
  ...
<style name="red" basestyle="default" linewidth="2"
    fillstyle="Dot">
    <fillcolor r="255" g="0" b="0"/>
    <linecolor r="255" g="0" b="0"/>
</style>
```

Fig. 4. Style usage example.

2.1 Styles

GUItar has a rich and powerful style set of facilities that allow the creation and manipulation of the graphical representation of nodes and edges. GUItarXML styles are similar in concept to *cascading style sheets* (CSS) [9]. CSS provide a way for markup languages to describe their visual properties and formatting. GUItar styles only focus on visual properties and allow the definition of style classes. The `style` elements have the `name` attribute that can be used when the style is to be applied to an object. An example is shown in Figure 4. The `basestyle` attribute is the name of the base style for the style. Styles inherit their properties from their base style. Besides those attributes, the `style` can have the actual styling attributes such as line color, fill color, line width, etc.

```
<graph>
  ...
<node ...>
    ...
    <draw_data obgroup="final" ...>
    ...
</node>
  ...
</graph>
  ...
<state_object_group name="final">
    <ellipse style="default" primary="True">
        <size x="35" y="15"/>
    </ellipse>
    <ellipse style="default">
        <size x="40" y="20"/>
    </ellipse>
</state_object_group>
```

Fig. 5. A node object group for final states.

Node styles are more complex than the edges styles. A node can be composed of several sub-objects. Consider the classic representation of a final state pre-

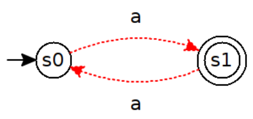


Fig. 6. A GUItar diagram with `red` style edges and nodes with an `initial` and a `final` state object group.

sented in Figure 5. That state representation is composed of two sub-objects that are two concentric circles (or ellipses). These complex graphical objects are defined in `state_object_group` elements. These elements contain some state style specific options and one or more shape elements. Available shapes are: `ellipse`, `rectangle`, `floatingtext` (a static label), and `arrowspline` (a multiple control point spline arrow). Figure 6 shows an example of an automaton that makes use of these styles.

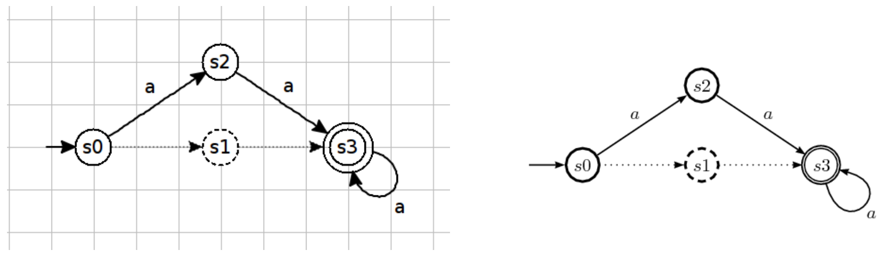


Fig. 7. A GUItar diagram and the Vaucanson-G LaTeX output.

3 Format Conversions

GUItarXML generic format is used for the conversion to, and from, other formats. Currently GUItar implements conversion methods to export to the following formats: GraphML, dot [10], Vaucanson-G [11] and FAdo. It has, also, methods to import from all those formats except from Vaucanson-G. Conversion to GraphML is simple, since GUItarXML is based on GraphML. An XSL transformation is used to remove the styles and extra elements in nodes, edges, and graphs. The result is a basic GraphML file with the basic topological data. The dot is a language for the specification of graphs that is part of Graphviz, a package of graph visualization tools [10]. GUItar exports to dot and currently dot graphs will retain all data except for style data. The Vaucanson-G is a LaTeXpackage that allows the inclusion of automata-like diagrams in LaTeX documents. When GUItar exports to Vaucanson-G it creates a document with one `VCDraw` environment containing the automata data. GUItar styles are converted to LaTeX macros (see Figure 7). Whenever an exact conversion is not possible, a reasonable approximation is used. For example, Vaucanson-G doesn't support drawing of complex states, so the primary object is used as a base for the Vaucanson-G state.

FAdo's format is used for the representation of deterministic finite automata (DFA) and nondeterministic finite automata (NFA). GUItar uses its Xport mechanism to handle the export and import from that format.

Conversions to SVG (Scalable Vector Graphics) [12] and FSMXML [13] are currently being implemented. The first format is a generic XML language for describing two-dimensional graphics. The second one is an XML format proposal for the description of weighted automata, where the graphical information is based on the Vaucanson-G styling and is mainly oriented for algebraic applications of automata.

3.1 Xport Mechanism

The Xport mechanism allows an easy way to add new export and import methods to GUItar, either coded in Python or as XSL transformations. This mechanism is configured using an XML specification (see Figure 8).

The specification allows for multiple Xport to be defined. Each one has a `name`, that is the string that will appear in the GUItar interface export/import menus. The `wildcard` attribute can be a file wildcard used in the wxPython's file dialog wildcard argument.

Depending on what type of Xport it is, additional attributes are required. XSL Xport require the attributes `expfile` (export) and `impfile` (import) that are file `paths` of the XSL transformations. For regular Xport, the attribute `import` must exist and contains the Python import statement for the module that contains the conversion methods. The element `export` and the element `import` indicate the methods used to perform the conversions.

4 Foreign Function Calls

The *foreign function call* (FFC) mechanism provides GUItar with a generic interface to external Python libraries, or programs and mechanisms to interact with foreign objects. In the first case, the FFC mechanism calls a function directly from an external Python module (Module FFC). In the second case, it creates a foreign object and then calls methods of that object (Object FFC). Both cases require an XML configuration file that specifies, among other things, the names of the available methods, their arguments, and their return values. Figure 9 presents the general FFC mechanism.

```
<xport_data>
  <xport name="FAdo" import="GF" wildcard="FAdo files
      (*.fa)|*.fa">
    <import method="read_o"/>
    <export method="save_o"/>
  </xport>
  <xslxport name="GraphML" expfile="guitar-graphml.xsl"
      impfile="graphml-guitar.xsl" wildcard="xml files
      (*.xml)|*.xml"/>
 </xslxport>
<xport_data>
```

Fig. 8. Xport configuration for FAdo and for GraphML files.

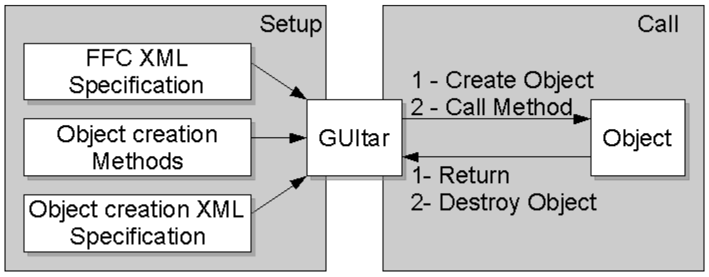


Fig. 9. Module and Object FFC.

```
<foreign_function_call>
 <depends import=...>
   ...
 <module import=...> or <object creatorname=...>
  <method>
    ...
  <menu_data>
     ...
  </menu_data>
    ...
 </module> or </object>
   ...
</foreign_function_call>
```

Fig. 10. An FFC configuration file.

4.1 FFC Configuration Specification

FFC configuration files can contain several FFC definitions (module or object). Module FFC's must indicate the module they import (the `import` attribute that uses Python's import syntax), while object FFC's must indicate the name of the object creator they will use to create the object (`creatorname` attribute). A more user friendly name and a description of the FFC method may be given in the optional attributes `name` and `description`, respectively.

Each module or object can contain multiple method definitions. For each method, a `name`, an `id`, it's `arguments`, and it's `return` values are needed (see Figure 11). Arguments require the `type` attribute, that states its type. An optional `default_value` attribute can be given with the default value for this argument. Return values only require the `type` attribute. A FFC method can have multiple arguments and multiple return values with an order that must agree with their appearing order in the definition. The types for arguments and return values can be `Int`, `Float`, `Boolean`, `String`, and also, one of the following:

`File:` a file. It has two additional attributes: `dialogmode` and `filemode`. The `dialogmode` can be either `save` or `load`, and indicates the type of dialog to

```
<method name=... id=...>
  <argument type=... default_value=.../>
     ...
  <return_value type=.../>
     ...
</method>
```

Fig. 11. Structure of `method` elements.

```
<menu_data>
 <menu title="FAdo">
  <menu title="DFA">
    <menu_entry descr1="Minimal" descr2="Minimize automata"
        action="minimal"/>
    <menu_entry descr1="Union" descr2="Returns union of two
        DFAs" action="union"/>
  </menu>
 </menu>
</menu_data>
```

Fig. 12. A menu data example. Nested menu elements will create sub-menus.

show. The `filemode` can be either `path` or `file`, and indicates if the value is expected to be the path to the file (string) or a Python file object.

`Canvas:` a GUItarXML representation of a diagram in a canvas.

`Object:` a foreign object. The attribute `creatorname` is required to know which *object creator* will be used.

FFC's can, optionally, define their own menus. Those menus will be dynamically created by GUItar on startup, just like GUItar's own native menus. The structure of the `menu_data` element is presented in Figure 12.

Optionally, FFC's can also include multiple `depends` elements that have the `import` attribute and are used to indicate any module dependencies that the FFC has. These dependencies are checked when the FFC's are being loaded and a warning is raised for the user, in case of failure.

4.2 Foreign Objects

A *foreign object* is a Python object which type is not recognized by GUItar. Its GUItar type (see Section 4.1) will be `Object`, and to convert to and from a GUItar object *object creators* are used. Object creators require two components: an XML configuration file and a Python file containing the conversion methods. Object creator configuration files may define multiple object creators, under the condition that they are methods defined in the same module (see Figure 13). The module containing the methods' definition is the value of the `import` attribute.

```
<object_creator_group  import="GF">
  <depends import="FAdo"/>
  <depends import="yappy"/>
  <object_creator name="FAdoDFA" class="DFA">
    <to_method method="GuitarToFA">
      <argument type="Canvas" default_value="Current"/>
    </to_method>
    <from_method method="FAToGuitar">
      <returns type="Canvas"/>
    </from_method>
  </object_creator>
</object_creator_group>
```

Fig. 13. Example object creator configuration file for FAdo DFA objects.

Each object creator has a `name` attribute, which value is used in arguments and return values of FFC methods. The `class` attribute contains the foreign object Python class name. The attributes `to_method` and `from_method` name the methods to be used in the conversions.

4.3 A FFC Example

To illustrate the use of a FFC object, we will use the FAdo DFA minimal method. This method computes the minimal DFA equivalent to a given automaton.

```
<foreign_function_call>
  <depends import="FAdo"/>
  <object creatorname="FAdoDFA">
  <method name="minimal" id="minimal" friendly_name="Minimal"
      description="Returns equivalent minimal DFA">
    <return_value type="Object" creatorname="FAdoDFA"/>
  </method>
    ...
</foreign_function_call>
```

Fig. 14. Fragment of the FAdo DFA interface.

Figure 14 shows part of the definition of the FAdo DFA interface, highlighting the *minimal* method.

Figure 15 shows the GUItar graphical interface and the menu selection of the *Minimal* method. The user will be asked to choose which GUItar object wants to be minimized and the object creator will create the correspondent DFA object. The FAdo DFA minimal method takes no extra arguments, but if there were any arguments, they would also be created using their appropriate creator. After

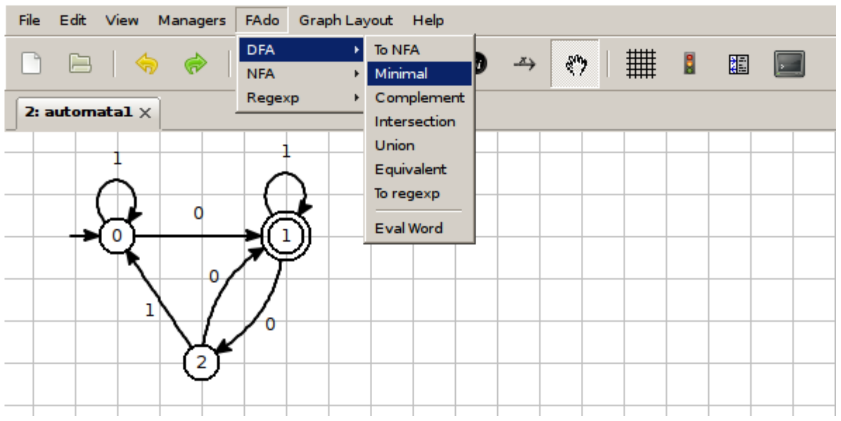


Fig. 15. GUItar graphical interface and the FAdo FFC interaction.

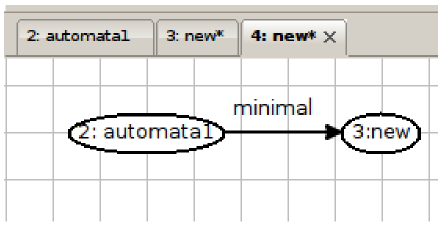
the termination of the FFC execution, the returned values are processed. In this example, a DFA object is returned. That object is converted into a GUItar object (GUItarXML string) which is automatically drawn in a new canvas.

5 Object Library

As seen in the example of Section 4.3, the GUItar framework allows the operation over several automata, using the FFC mechanisms. It is possible to trace the operations (methods) used to generate objects and which objects are arguments for that operations. In this way it is possible to relate the various objects manipulated, and also determine some of their properties. This information can be used for constructing automata databases, and store it in the `automata_data` elements (of the GUItarXML format). Currently we are developing a XML specification for this information. However, we can already automatically display in the GUItar graphical interface the relationships between the several objects, and save them as GUItarXML diagrams. Each diagram has as nodes the created objects. Each arc is labeled with the operation that takes the object that constitutes arc source, to produce the one that constitutes its target. For each operation, there are the same number of arcs (edges) as arguments. In Figure 16 we present two object dependence diagrams. One for an application of the FAdo minimal method and the other for the union of two automata.

6 Conclusions

In this paper we presented an ongoing work for the development of a set of tools for the visualization, conversion and manipulation of automata. The XML framework provides a means of obtaining extensibility and interoperability with external automata manipulation tools. Although GUItar is already a functional prototype, several improvements are needed. More format conversions must be implemented and some of the existent ones must be extended. The FFC configuration must be automated in order to easy the interaction with external systems. The object library must be fully implemented.



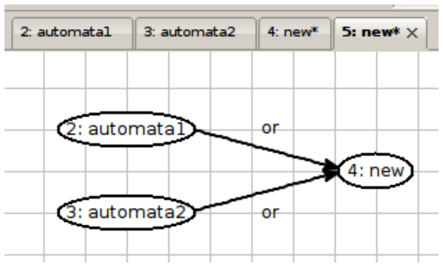


Fig. 16. Examples of object dependence diagrams.

References

1. Moreira, N., Reis, R.: Interactive manipulation of regular objects with FAdo. In: Proceedings of 2005 Innovation and Technology in Computer Science Education (ITiCSE 2005), ACM (2005) 335–339
2. Almeida, A., Almeida, M., Alves, J., Moreira, N., Reis, R.: FAdo and GUItar: tools for automata manipulation and visualization. In Maneth, S., ed.: 14th CIAA'09. Volume 5642 of LNCS., Springer (2009) 65–74
3. FAdo: tools for formal languages manipulation. `http://www.ncc.up.pt/FAdo` (Access date:1.1.2010)
4. Foundation, P.S.: Python language website. `http://python.org` (Access date:1.12.2009)
5. Smart, J., Roebling, R., Zeitlin, V., Dunn, R.: wxWidgets 2.6.3: A portable C++ and Python GUI toolkit. (2006)
6. WWW Consortium: XML specification WWW page. `http://www.w3.org/TR/xml` (Access date:1.12.2008)
7. GraphML Working Group: The GraphML file format. `http://graphml.graphdrawing.org` (Access date: 01.12.2009)
8. van der Vlist, E.: RELAX NG. O'Reilly (2003)
9. WWW Consortium: CSS WWW page. `http://www.w3.org/Style/CSS` (Access date:13.03.2010)
10. Graph Visualization Software: The dot language. `http://www.graphviz.org` (Access date:1.12.2009)
11. Lombardy, S., Sakarovitch, J.: Vaucanson-G. `http://igm.univ-mlv.fr/~lombardy` (Access date:1.12.2009)
12. WWW Consortium: Scalable vector graphics. `http://www.w3.org/Graphics/SVG/` (Access date: 01.12.2009)
13. Vaucanson Group: FSMXML. `http://www.lrde.epita.fr/cgi-bin/twiki/view/Vaucanson/XML` (Access date: 01.12.2009)