José Daniel da Silva Alves

An Interactive System for Automata Manipulations

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2010
José Daniel da Silva Alves

An Interactive System for Automata Manipulations

Dissertação submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Mestre em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2010
To my parents
Acknowledgments

I would like to thank professors Nelma Moreira and Rogério Reis for their help and their patience throughout my academic formation, especially during the execution of this work.

I would also like to thank my friend and colleague André Almeida for his friendship and positive criticism.

I would like to thank my parents, José Alves and Ernestina Alves, for their constant support and encouragement.

Finally, I would like to thank my brother, André Alves, for being a great brother and also for helping me to stay sane throughout my academic formation.
Abstract

Automata are fundamental computation models with many practical applications in Computer Science. For this reason, many applications and libraries have been developed for their study, symbolic manipulation and visualization.

Some applications and libraries focus only on providing an efficient platform for testing and developing algorithms, having little or no means of graphical visualization and manipulation. Graphical applications exist, but are generally more limited on the kinds of manipulations that can perform, and are usually more adapted for didactic purposes. GUltar is a graphical interface for the manipulation of automata diagrams, providing assisted drawing features that facilitate the drawing of diagrams. GUltar also provides style editors for nodes and transitions that allow creation of graphical styles to cope with many kinds of applications. A generic diagram description language, GUltarXML, was developed for GUltar. GUltarXML is expressive enough to be used as an intermediate format for conversion into other diagram representation formats such as GraphML, dot and VauCanSon-G. A generic extension mechanism, called the Foreign Function Calls (FFC) allows GUltar to interface with external diagram manipulation tools like, for instance, the FAdo engine.

The FFC mechanism provides Object Creators that can handle foreign objects, objects that are not native to GUltar. The Object Creators ensure that GUltar is not limited to the kinds of objects it can manipulate. The FFC mechanism provides an Object Library that stores objects created during the execution of FFCs that is able to graphically represent the relations between them. GUltar also has scripting capabilities and a
console that is able to manipulate the user interface, giving guitar the possibility of both a mouse-driven and a text-based interaction.

This work presents the GUItar application and the features designed to enhance its extensibility and interoperability, in particular:

- The GUItarXML language;
- The import and export filters;
- The Foreign Function Calls;
- The Object Creators, that are used to handle Foreign objects;
- The Object Library that is used to track objects created during the execution of FFCs;
- The scripting framework and the console interface.
Contents

Abstract 5

List of Figures 13

1 Introduction 15

2 Applications and Languages 17

   2.1 Introduction ...................................................... 17
   2.2 Applications ...................................................... 18
      2.2.1 AMoRE ......................................................... 18
      2.2.2 Vaucanson ..................................................... 19
      2.2.3 FAdo .......................................................... 19
      2.2.4 Graphviz ...................................................... 20
      2.2.5 JFLAP ........................................................ 20
      2.2.6 yFiles and yEd .............................................. 21
      2.2.7 Visual Automata Simulator .................................. 22
   2.3 Graph and Automata Representation Languages .................. 23
2.3.1 Non-XML Languages ........................................ 23
  2.3.1.1 GML ........................................ 23
  2.3.1.2 FAdo ........................................ 24
  2.3.1.3 dot ........................................ 24

2.3.2 XML Based Languages ................................. 26
  2.3.2.1 GraphML ..................................... 27
  2.3.2.2 FSMXML ..................................... 29
  2.3.2.3 XGMML ..................................... 31
  2.3.2.4 SVG ........................................ 31

2.3.3 LATEX ...................................................... 32
  2.3.3.1 VauCanSon-G ................................. 33
  2.3.3.2 GasTeX ....................................... 35

2.4 Scripting .................................................... 36

3 GUItar ....................................................... 39
  3.1 Introduction ............................................ 39
  3.2 GUItar’s Architecture ................................. 40
    3.2.1 The Drawgraph .................................. 42
  3.3 Features ............................................... 42
    3.3.1 Style Managers .................................. 42
    3.3.2 Graph Classifier ................................. 45
    3.3.3 Semaphores ..................................... 45
3.3.4 Import and Export ........................................... 46
3.3.5 Foreign Function Calls ................................. 46
3.3.6 Scripting and Console ............................... 47

4 GUItarXML ........................................ 49

4.1 Introduction .............................................. 49
4.2 Structure of a GUItarXML Document ............. 49
  4.2.1 Nodes and Edges .................................. 50
    4.2.1.1 diagram_data ................................ 51
    4.2.1.2 draw_data ................................... 51
    4.2.1.3 label ......................................... 54
    4.2.1.4 automata_data ............................... 55
  4.2.2 Graph’s automata_data ............................ 55
  4.2.3 Style Data ........................................ 57

5 Format conversions ..................................... 61

5.1 Introduction .............................................. 61
5.2 VauCanSon-G Export .................................. 62
5.3 XPort .................................................. 63

6 Foreign Function Calls ................................. 67

6.1 Introduction .............................................. 67
6.2 XML Specification ..................................... 68
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.1 Top Level</td>
<td>68</td>
</tr>
<tr>
<td>6.2.2 Methods</td>
<td>69</td>
</tr>
<tr>
<td>6.2.3 Menus</td>
<td>71</td>
</tr>
<tr>
<td>6.3 Object Creators</td>
<td>73</td>
</tr>
<tr>
<td>6.3.1 XML Specification</td>
<td>74</td>
</tr>
<tr>
<td>6.4 Object Library</td>
<td>75</td>
</tr>
<tr>
<td>7 Scripting</td>
<td>77</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>77</td>
</tr>
<tr>
<td>7.2 GUitarSimpleAPI</td>
<td>78</td>
</tr>
<tr>
<td>7.3 Console</td>
<td>80</td>
</tr>
<tr>
<td>8 Conclusions</td>
<td>83</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Automaton example ........................................... 18
2.2 AMoRE interface .............................................. 19
2.3 JFLAP ........................................................... 20
2.4 yED example ................................................... 21
2.5 Visual Automata Simulator example ............................ 22
2.6 GML file example .............................................. 23
2.7 FAdo example ................................................... 24
2.8 A dot file ....................................................... 25
2.9 Example of Figure 2.8 rendered by the dot application .... 25
2.10 XML document example ....................................... 26
2.11 GraphML Example ............................................ 28
2.12 Part of an FSMXML document ................................. 29
2.13 Example of a regular expression in FSMXML ................ 30
2.14 Example of an XGMML document .............................. 32
2.15 Example of a VauCanSon-G document ......................... 34
2.16 Rendering of the VauCanSon-G example in Figure 2.15 .... 34
Chapter 1

Introduction

Automata are fundamental computation models with many practical applications in Computer Science, such as compilers, voice and image recognition, model checking, bioinformatics or computer networks. Many applications and libraries have been developed for the study, symbolic manipulation and visualization of automata. However, while some applications provide a platform for testing and developing new algorithms, they often do not have any kind of graphical visualization capabilities. Also, those that provide visualization features often are not extensible and only allow limited interaction. In this context, the GUItar application is being developed. GUItar is a graphical tool for the drawing and the manipulation of many kinds of diagrams, with special focus on automata. This application provides interesting automata drawing capabilities, including assisted drawing and visualization features and complex styling tools.

This thesis presents the GUItar application, while focusing on its extensibility and interoperability features. A generic XML language for the representation of automata, called GUItarXML, is presented. GUItarXML is used as the default specification to export GUItar diagrams and is also used as the base to perform conversions to other formats of automata and graph representation. The generic extension mechanism for GUItar, the Foreign Function Calls (FFC) is also presented. The FFCs provide GUItar
with a framework for calling methods from external modules or objects. The FFCs also provide an Object Library that can store objects created during the execution of a Foreign Function Call and that is able to graphically represent the relationships between those objects. GUItar’s scripting framework, that provides GUItar with automation and scripting features, and the console interface will also be presented.

Chapter 2 presents a review of some graph and automata drawing applications and descriptive languages. The chapter includes a brief overview of (visual) scripting languages. Chapter 3 presents an overview of the GUItar application, showing the GUItar architecture and some of its more interesting features. In Chapter 4, the GUItarXML language is presented and in Chapter 5 the method used to convert GUItarXML to other format is presented. The FFCs are presented in Chapter 6. Chapter 6 presents the XML specification used by FFCs, the Object Creators and their specification, and the Object Library. In Chapter 7 GUItar’s scripting features and the console interface are presented. Chapter 8 concludes this thesis and proposes some future work.
Chapter 2

Applications and Descriptive Languages for Graphs and Automata

2.1 Introduction

Many applications and libraries for the study, symbolic manipulation and visualization of automata have been developed. Some implementations focus only on providing a platform to test existing algorithms or implement new ones, having little or no graphical display or drawing capabilities. Graphical tools for drawing and manipulation also exist, but they are, generally, much more limited on the kinds of manipulations they can perform and are often geared more towards didactic purposes. In Section 2.2, some automata manipulation applications are presented. Since automata share some graphical properties with graphs, some graph drawing applications can be adopted for drawing or visualization of automata diagrams, therefore, some examples of graph drawing applications will also be considered. The usefulness of the applications would be limited, however, if they did not have the means of storing the results of their manipulations or exchange them with other applications. For that reason,
specification languages to describe them also had to be developed. Some of these languages were designed mainly to represent graphs, but they can easily be extended to include properties of automata. Some of the languages are reviewed in Section 2.3. An advantage that some of the console based applications and libraries have is the capability for scripting. Scripting automates repetitive operations to make them both faster to execute and more resistant to human error. For this reason, scripting technologies are presented in Section 2.4. Several of the examples presented in this chapter will be based on the automaton in Figure 2.1.

![Automaton example](image)

Figure 2.1: Automaton example

### 2.2 Applications

#### 2.2.1 AMoRE

AMoRE [MMP+95] (Automata, MOnoids and Regular Expressions) is an open-source, console based application for the symbolic manipulation of formal languages. It implements the classical automata algorithms like minimization, intersection or union of two automata, and computes the syntactic monoid of the language of a automaton. Figure 2.1 shows the AMoRE interface.
2.2. Applications

Figure 2.2: AMoRE interface

2.2.2 Vaucanson

The Vaucanson project [CLC+05] aims to provide a platform for the manipulation of finite state machines, with focus on transducers and weighted automata. It consists of a C++ library that includes a few example programs implemented on that library. Those programs can be used to test some of the features of the library. The library itself implements many algorithms and uses the FSMXML [Gro10c] format as the default export and communication format.

2.2.3 FAdo

FAdo [AAA+09] is an ongoing project that aims to provide a set of tools for the symbolic manipulation of formal languages. The FAdo engine is written in Python and provides a set of classes for the manipulation of finite automata and regular expressions. It implements several of the most common algorithms related to automata and regular expressions, such as converting regular expressions to automata (and vice-
versa), minimization, determinization, and intersection and union of automata.

2.2.4 Graphviz

Graphviz [Res10b] is a suite of applications and a library for drawing graphs. Graphviz uses the dot language [Res10a] as the means of specifying graphs. The Graphviz applications (dot, neato, fdp, etc...) receive a file in the dot format and output an image with the graph drawn according to the algorithm the application implements and the drawing constraints given in the file.

2.2.5 JFLAP

JFLAP [RF06] is a didactic graphical tool for the visualization and manipulation of formal languages that has support for various types of automata, Turing machines,
2.2. APPLICATIONS

grammars and regular expressions. The automata interface provides basic drawing capabilities and a few algorithms like minimization, determinization and conversion to grammars (that can then be manipulated inside the application). The grammar interface allows defining new grammars and has features such as building parse tables, parsing of strings (while interactively building the parse trees), and conversion to automata. Figure 2.3 shows part of the JFLAP automata interface.

2.2.6 yFiles and yEd

![yFiles and yEd](Image)

Figure 2.4: yED example

yFiles [yG10b] is a comprehensive commercial library that provides the means for developing applications for the visualization and edition of graphs and diagrams. yEd [yG10a] is a free (but not open-source) diagram editor built with the yFiles library as
a technological demonstration of the library. yEd has many options for customizing
the graphical properties of the diagrams and implements a few automatic layout
algorithms. yEd can also import and export to many formats such as GML, XGMML
and GraphML. Figure 2.4 shows the yEd interface.

2.2.7 Visual Automata Simulator

Visual Automata Simulator [Bov10] is an automata and Turing machines simulation
application written in Java. It only has basic drawing capabilities but can simulate
checking if a word is accepted by an automaton or a Turing machine, and can show
that process step-by-step by highlighting states on the automaton and showing the
state of the tape on a Turing machine, as symbols are consumed. Figure 2.5 shows
the Visual Automata Simulator interface.

![Visual Automata Simulator example](image)

Figure 2.5: Visual Automata Simulator example
2.3 Graph and Automata Representation Languages

2.3.1 Non-XML Languages

2.3.1.1 GML

GML [Him] is a graph representation language that is intended to be simple, portable.
and flexible. It is an attempt at establishing a common data format for graph manipulation applications. **GML** documents are sets of keys followed by values or a list of values. Lists of values are surrounded by square brackets. Users and applications are free to use any non-standard keys, however, recognizing those keys depends on the implementation of the applications that are reading the documents. Figure 2.6 shows an example of a GML file.

### 2.3.1.2 FAdo

| @DFA 0 0 b 0 1 b 1 2 b 2 0 a 1 1 a 2 2 a 0 | @NFA 3 0 a 0 0 b 1 0 b 2 1 a 1 1 b 3 2 a 2 2 b 3 |

Figure 2.7: FAdo example

The **FAdo** engine uses a simple language for storing automata. It can contain multiple automata, each starting with either @DFA or @NFA (depending on the type of the automaton), followed by the identifiers of the list of final states. The following lines contain the transitions (one per line), and are composed of three elements: the source state identifier, the label of the transition and the target state identifier. Figure 2.7 shows two automata: a *DFA*, with three states, on the left and an *NFA*, with four states, on the right.

### 2.3.1.3 dot

The **dot** language is a graph specification language used by the **Graphviz** graph visualization tools. The **dot** language allows the specification of directed and undirected
graphs, including their graphical attributes, and some drawing constraints used by the Graphviz applications. A dot document may contain multiple graphs or digraphs. Transitions of the form “node id1->node id2” are for digraphs, and “node id1 – node id2” are for graphs. Attributes of nodes and transitions are given inside square brackets. Figure 2.8 shows an example of a graph specified in the dot language.

```
digraph A1{
    rankdir=LR;
    s0 [label="s0", shape=doublecircle];
    s1 [label="s1", shape=circle];
    s2 [label="s2", shape=circle];
    null [shape = plaintext label="" ];
    null -> s0;
    s0 -> s0 [label="b"]; 
    s1 -> s1 [label="b"]; 
    s2 -> s2 [label="b"]; 
    s0 -> s1 [label="a"]; 
    s1 -> s2 [label="a"]; 
    s2 -> s0 [label="a"]; 
}
```

Figure 2.8: A dot file

![Figure 2.9: Example of Figure 2.8 rendered by the dot application](image)

The rankdir statement indicates that the diagram is to be drawn from left to right. Nodes and edges have a label attribute that contains their label, and nodes have the shape attribute that indicates what shape they will use. Since Graphviz has no built-in shape with an incoming arrow (for initial states), an invisible node (called null) was
placed next to the initial state with a transition to the initial state. Figure 2.9 shows this diagram rendered by the dot application.

### 2.3.2 XML Based Languages

XML [Con10b] (eXtensible Markup Language) is a formalism used to describe a family of languages that are widely used for storing, exchanging, and representing information. The basic building blocks of XML are the elements, identified by a tag. Elements can have attributes and other elements nested within them. Elements can also have text containers. XML documents must have a root element. Figure 2.10 shows an example of a XML document. The root element has the tag “xml_example”. The root element has an element called “example”, which in turn has the attributes “att1” and “att2”, and a sub-element “other”. The “other” sub-element contains the text “Text example”.

```xml
<xml_example>
  <example att1="1" att2="2">
    <other>
      Text example
    </other>
  </example>
</xml_example>
```

Figure 2.10: XML document example

XML does not define what elements are used or their valid contents. It is up to the user to define them, usually by means of a schema. Common schemas are XSD [Con10c], DTD [Con10a], RelaxNG [vdV03], or RelaxNG-Compact [vdV03]. They allow defining a particular XML language and make it possible to validate documents of the language against the schema. The XML definition requires that all XML documents are well-formed. This means, for example, that there must be only one root element and all tags must be properly closed. This is different from validation: a document
can be well-formed, but invalid, meaning that it does not conform to the schema. XSL [Con10d] (eXtensible Stylesheet Language) is a language that allows to describe how to render an XML document. Although CSS (Cascading Style Sheets) may be used for the same purpose, the W3C [Con09b] recommends XSL as the default styling language for XML. Besides rendering properties, XSL documents allow performing complex transformations to XML documents. This is done by matching parts of the original XML document to templates defined in XSL and then using the rules defined in those templates to write the corresponding result to the output file. This is commonly used to make the conversion between XML dialects or to convert XML documents into HTML documents.

2.3.2.1 GraphML

The GraphML [Gro10a] language is an attempt at setting a standard specification for graph representation based on XML.

The basic GraphML document allows having zero or more graph definitions represented as graph elements. The graph elements can have zero or more node and edge elements that must have an id attribute that must be unique within the entire document. The edges must also have the attributes source and target that are the identifiers of nodes located in the same document. Besides simple graphs, GraphML also provides support for hyper-graphs - graphs where edges (called hyper-edges) can have more than two endpoints. Hyper-edges are declared as hyperedge elements and can have multiple endpoint sub-elements with the identifier of the endpoint nodes. GraphML also provides the possibility of declaring nested graphs. This is achieved by declaring a graph inside a node element. Ports is another feature of GraphML that allows the specification of additional locations where edges and hyper-edges can connect into nodes. Finally, GraphML provides an extension mechanism in the form of key-value pairs called graphml-attributes. This mechanism requires the declaration of a key element at the beginning of the document that defines an identifier, the domain of
Figure 2.11: GraphML Example
the attribute (graph, node, edge, or all), a name for the attribute, and a default value. This special “attribute” can then be used by declaring data elements with the appropriate key inside its proper domain. In Figure 2.11 we have part of an example of a GraphML document, making use of the extension mechanism to add labels to nodes and edges.

2.3.2.2 FSMXML

```xml
<fsxml>
...
<automatonStruct>
  <states>
    <state id="0" name="s0"/>
    <state id="1" name="s1"/>
  </states>
  <transitions>
    <transition source="0" target="1">
      <label>
        <monElmt> <monGen value="a"/> </monElmt>
      </label>
    </transition>
    <transition source="1" target="1">
      <label>
        <monElmt> <monGen value="b"/> </monElmt>
      </label>
    </transition>
  </transitions>
</automatonStruct>
```

Figure 2.12: Part of an FSMXML document

FSMXML [Gro10c] is an XML language for the description of finite state machines (especially transducers and weighted automata), and regular expressions, developed as part of the Vaucanson project [CLC+05].

Automata are described in automaton elements. Their sub-element valueType de-
scribes the algebraic structures associated with transition labels and their sub-element
automatonStruct contains the states and the transitions elements. The states

```xml
<fsxml xmlns="http://vaucanson-project.org" version="1.0">
  <regexp name="example">
    <valueType>
      <semiring type="numerical" set="B" operations="classical"/>
      <moxnoid type="free" genSort="simple" genKind="letter"
        genDescript="enum">
        <monGen value="a"/>
        <monGen value="b"/>
      </moxnoid>
    </valueType>
    <typedRegExp>
      <star>
        <sum>
          <monElmt>
            <monGen value="a"/>
          </monElmt>
          <monElmt>
            <monGen value="b"/>
          </monElmt>
        </sum>
      </star>
    </typedRegExp>
  </regexp>
</fsxml>
```

Figure 2.13: Example of a regular expression in FSMXML

element can have an arbitrary number of state elements, each one representing a state. The state elements must have an id attribute and a name attribute, that is the label of the state. The transitions element has an arbitrary number of transition elements. Each transition element has the source and the target attributes that are the identifiers of the source and target states. Labels of transitions are described
using a regular expression, that is a combination of \texttt{sum}, \texttt{star}, \texttt{product} or \texttt{monElmt} elements. The \texttt{sum} and \texttt{product} elements must have at least two sub-elements, the first one being the left operand of that operation. The \texttt{star} elements have one sub-element. Regular expressions are described in \texttt{regExp} elements. Like automata, they must have a \texttt{valueType} element to describe the algebraic type of the regular expression. The expression’s body is described in \texttt{typedRegExp} elements and has the same structure as automaton transition labels. Figure 2.12 shows an example of an automaton described in FSMXML. Parts of that document were removed due to space constraints. Figure 2.13 shows an example of a regular expression.

\subsection*{2.3.2.3 XGMML}

XGMML \cite{XGM09} (eXtensible Graph Markup and Modeling Language) is an XML language for the description of graphs based on GML. In fact, XGMML could be considered a direct translation of GML into XML. The XGMML specification offers the following simple rules to transform GML documents into XGMML documents:

- A GML key is a name of an XGMML element if its value is a list of key-value pairs;
- A GML key is a name of an XGMML attribute if its value is a number or a string;
- The comment GML tag and the GML lines starting with “#” character must be ignored or translated to XML comments.

Figure 2.14 show an XGMML document.

\subsection*{2.3.2.4 SVG}

SVG \cite{Con09a} (Scalable Vector Graphics) is an XML language for the description of two-dimensional vector graphics. SVG documents describe images by arranging and
compositing basic shapes like rectangles, circles, ellipses, simple lines, or text. **SVG** also provide the “path” feature that allows the creation of complex shapes and curves by using straight or curved lines. Is is possible to style **SVG** elements, i.e., change their color and other graphical properties. This can be achieved by changing style attributes directly for each shape or by using external **CSS** specifications or even inline **CSS** code.

### 2.3.3 \LaTeX

\LaTeX{} [Lam94] is a document preparation system and language for high quality typesetting based on the \TeX{} system [Knu84], commonly used to produce scientific and technical documents. \LaTeX{}’s philosophy is that the user should not have to worry about the layout of the document and should only have to concentrate on writing its content. Therefore, unlike most popular word processors such as Microsoft Word [Cor09] or OpenOffice Writer [Ope09], \LaTeX{} is not an WYSIWYG (What You See Is What You Get) system (although WYSIWYG tools exist for \LaTeX{} such as LyX [Tea09]). Instead, the user creates \LaTeX{} documents using any text editor and then compiles that text using a \LaTeX{} compiler, that will be in charge of typesetting the document. \LaTeX{} is extensible and many packages for working with mathematics,
2.3. GRAPH AND AUTOMATA REPRESENTATION LANGUAGES

chemistry, graphics, and, of course, for automata drawing exist.

2.3.3.1 VauCanSon-G

VauCanSon-G [Gro09] is a widely used \LaTeX{} package that contains a set of macros for drawing automata in \LaTeX{} documents. It is built upon the PSTricks [Gro10b] package, a \LaTeX{} package for plotting graphs and drawing 2D and 3D figures. Automata are drawn inside a VCPicture environment. The user must indicate the dimensions of the environment by indicating the coordinates of the lower left and upper right corners of the “picture”. This is used to create a bi-dimensional coordinate system where states can be laid. States are declared by using the \State command. This command must receive the coordinates of the state and an id for the state. Optionally, it may have a label. Initial states are declared by using the \Initial command with the identifier of the initial state. This command receives an optional argument with the direction of the arrow. Final states can be declared in two ways: similarly to initial states, by using the \Final command or by declaring states as \FinalState. The first method produces final states with an outgoing arrow, while the second produces states with a double circle. Transitions are declared by using either the \EdgeX command or the \ArcX command. The X in the commands stands for either L or R, and indicates the side of the label for edges, or the orientation of of the concavity for arcs. Both must receive the identifiers of the source and target nodes and, the transition label. Loops (transitions where the source and target node are the same) are declared with the \loopX command, where X is a cardinal direction of the loop (for example, NE for North-East). This command receives as an argument only the identifier of the node and the loop label. VauCanSon-G also provides styling options that allow changing some graphical properties of states and transitions. It also has support to call PSTricks macros directly. Figure 2.15 shows an example of an automaton specified in VauCanSon-G. Figure 2.16 shows the resulting automaton, as rendered by \LaTeX{}.
\begin{VCPicture}\{(-4, -4)(4, 1)\}
\FinalState [s0]\{(-3, 0)\}\{0\}
\InitialState [w]\{0\}
\State [s1]\{(3, 0)\}\{1\}
\State [s2]\{(0, -3)\}\{2\}
\EdgeL[0.5]\{0\}\{1\}\{a\}
\EdgeL[0.5]\{1\}\{2\}\{a\}
\EdgeL[0.5]\{2\}\{0\}\{a\}
\LoopN[0.5]\{0\}\{b\}
\LoopN[0.5]\{1\}\{b\}
\LoopS[0.5]\{2\}\{b\}
\end{VCPicture}

Figure 2.15: Example of a VauCanSon-G document

Figure 2.16: Rendering of the VauCanSon-G example in Figure 2.15
2.3. GRAPH AND AUTOMATA REPRESENTATION LANGUAGES

2.3.3.2 GasTeX

GasTeX [Gas09] is a LATEX package that adds macros to the LATEX picture environment to make it simpler to draw graphs, automata, and other kinds of diagrams.

\begin{picture}(30,30)
  \node[Nmarks=ir](0)(0,0){s0}
  \node(1)(60,0){s1}
  \node(2)(30,-30){s2}
  \drawedge(0,1){a}
  \drawedge(1,2){a}
  \drawedge(2,0){a}
  \drawloop(0){b}
  \drawloop(1){b}
  \drawloop[loopangle=270](2){b}
\end{picture}

Figure 2.17: Example of a GasTeX document

Diagrams are drawn inside a picture environment. A \gasset command may be used to set global drawing properties for nodes and edges. Nodes are declared with the \node command. This command receives three arguments: the identifier of the node, its coordinates and its label. Optional parameters may be passed that override the global parameters defined in \gasset. Edges are declared with the \drawedge command. The command receives two arguments: the identifier of source and target nodes, separated by a comma and the label of the transition. For drawing loops, the \drawloop command is used. Arguments are the same as for the edges, except that instead of a pair of identifiers, it receives only one. Figure 2.17 shows an example of an automaton defined in GasTeX. Figure 2.18 shows that automaton as rendered by LATEX.
2.4 Scripting

Scripting languages allow automatization of repetitive tasks. In graphical environments, they allow expressing in an easy way sequences of actions that could require several mouse actions. One of the oldest examples of scripting languages still in use are the Unix shell languages (Bourne shell, C shell, or Bourne-Again shell, for example). Shell scripts are often used in the automated installation and configuration of software, compilation of programs, or by users to automate tasks. AppleScript [Coo07] is a scripting language developed by Apple for the MacOS operating system that was designed to be easy to use. For that purpose, the AppleScript syntax is similar to a “natural language”. For example, the instruction ‘‘tell application X to quit’’ can be used to close applications (by replacing X with the name of an application). AppleScript also allows high-level interaction with applications, where scripts may be able to directly manipulate application components such as, for example, individual cells, rows, or columns on a spreadsheet application. Some programming languages such as Perl [SPbdf08] (developed in 1987), Tcl [Ous94] (1988), Python [Fou09] (1991), Lua [IdFC06] (1993) or Ruby [Lan09] (1995) are also scripting languages.

Visual programming is a programming paradigm where, in order to build a program,
graphical objects are manipulated instead of writing the corresponding expressions or commands in text form. Examples of visual programming languages are some dataflow languages like CODE [NB92], PROGRAPH [CP88] or SAC [Sch03]. An example of a visual scripting application is the MacOS Automator [Inc09]. Automator allows the creation of AppleScript scripts visually by means of a script recorder or by manually adding a set of actions to an execution queue.
Chapter 3

GUItar

3.1 Introduction

GUItar [AAA⁺09, Pro09] (Figure 3.1) is an application for the drawing and the manipulation of diagrams. GUItar allows the drawing and the manipulation of several
kinds of graph and automata diagrams, but it is especially focused on finite automata. 

GUItar was developed as a visualization tool for FAdo and it is, currently, still under development. GUItar provides the usual facilities for the assisted drawing of graph diagrams. GUItar also has complex style managers that allow the user to create new graphical styles for nodes and transitions or edit existing ones. GUItar has options to restrict the types of diagrams that can be drawn (Semaphores) and a graph classifier that is able to determine the type of diagram currently being drawn. GUItar allows multiple import/export filters, a mechanism that relies on the GUItarXML specification language as an intermediate format for conversion. GUItar provides a Foreign Function Call (FFC) mechanism for extensibility and interoperability with external diagram manipulation tools such as the FAdo engine. GUItar also has scripting capabilities and a basic script recorder that is able to generate scripts by recording the user’s actions. GUItar also provides a console that can be used to command GUItar.

This chapter presents an overview of the functionalities of the GUItar application and internal architecture. The next chapters describe the import/export mechanism, the Foreign Function Call mechanism and GUItar’s scripting features in more detail.

3.2 GUItar’s Architecture

GUItar is implemented in Python, and its graphical interface is implemented using the wxPython [wxP09] graphical toolkit. Figure 3.2 shows an overview of GUItar’s architecture. The GUItar user interface has a frame that contains a menubar, a toolbar and a notebook, which is a type of widget that can have multiple pages. The menubar and the toolbar are built from XML specifications on startup, and are contextual. This means that they react with the contents of the canvas, enabling or disabling menus or toolbar buttons as required. Each notebook page contains a main working area, called the Canvas, that is where the diagrams are drawn and edited. Each page also contains a properties panel that is hidden by default. The properties panel is used to change the properties of the selected objects. The interface also contains a
Python console that is hidden by default. The console can be used to run Python commands and interact with GUltar objects. The interface is mostly mouse-driven: the user chooses the type of action to be performed from the toolbar (node actions, edge actions, select or move canvas), and uses the mouse on the canvas to add nodes or transitions, move them or edit them. All of these actions are managed by the Drawgraph class and its components. Exporting and importing are handled by the Export and Import classes respectively. These classes are responsible for validating input files, exporting diagrams and, performing conversion between different types of documents. GUltar’s extension mechanism, the Foreign Function Call (FFC), are handled by the FFCManager class. This class is in charge of setting up, calling foreign functions and track FFC history through its Object Library.
3.2.1 The Drawgraph

The Drawgraph class is the class that manages all actions related to the interaction with the Canvas. The Drawgraph controls the Canvas class, that is the widget where diagrams are drawn, and receives commands from the GUImodes. The Drawgraph is responsible for maintaining the internal logic of everything related to the diagram’s structure, such as the internal identifiers of objects and their attributes. The Canvas class is an extension of the FloatCanvas [Bar09], an wxPython class for drawing 2D graphics. The most significant modification that was made to FloatCanvas were the addition of the arrow head class (that allows drawing customized arrow heads) and the creation of a generic spline class that can have multiple control points. The Grid class controls the positioning of elements in the canvas, making sure, for example, that nodes do not overlap. The mouse and keyboard actions are interpreted by the GUImode classes. These classes are responsible for detecting mouse and keyboard events, and calling the corresponding action in the Drawgraph. Switching edition mode in the toolbar effectively switches the GUImode that is currently active.

3.3 Features

GUITar provides a few interesting and unique features. They will be described in the following subsections.

3.3.1 Style Managers

The style managers allow the creation of custom graphical styles for nodes and transitions. They allow changing attributes such as colors, text fonts, line widths, fill, or style. Figure 3.3 shows the edge style manager.

Node styles are the most complex. A node object can be made of a composition of several graphical objects. The available graphic objects are:
3.3. FEATURES

Figure 3.3: Edge style manager

- Ellipse;

- Rectangle;

- Floating Label: A static floating label;

- Arrow: An arrow. The line can have an arbitrary number of control points;

The proportions and distances of the objects can be modified, and even set to auto-adjust according to the size of the node’s label. Figure 3.4 shows the node style manager. Figure 3.5 shows a few style examples. On the left, it shows styles for transitions, and on the right, it shows styles for nodes.
CHAPTER 3. GUITAR

Figure 3.4: Node style manager

Figure 3.5: Examples of edge and node styles
3.3. FEATURES

3.3.2 Graph Classifier

The graph classifier is responsible for determining what type of diagram is currently on the Canvas. Every time it is called the graph classifier runs a few test functions to determine things like, if there are any initial states and final states, or if every edge is directed or undirected. Diagram classes are identified by the result they expect from each test: must verify, meaning that the function must return True, can’t verify that means that the function must return False, or ignore, meaning that the result is ignored (default behavior). If every test verifies the required conditions, then the diagram belongs to that class. The graph classifier is used by the menubar to manage the contextual menus. Figure 3.6 shows part of the graph classifier interface.

<table>
<thead>
<tr>
<th>Class Result</th>
<th>Graph Classification</th>
<th>NFA</th>
<th>Digraph</th>
<th>Graph</th>
<th>Labelled Digraph</th>
<th>DFA</th>
<th>Multidigraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is only one transition</td>
<td>No</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>between a pair of states</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All arrows have at least 1 head</td>
<td>Yes</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All states have a label</td>
<td>Yes</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Has only one initial state</td>
<td>Yes</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All arrows have a label</td>
<td>Yes</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.6: Graph classifier

3.3.3 Semaphores

Semaphores define a set of constraints for diagram drawing. When semaphores are enabled, a semaphore is shown in the bottom left corner of the canvas. The light is green if none of the constraints are violated, otherwise, the light switches to red. The semaphore can be either locked or unlocked. Unlocked is the default behavior, where the light only turns red to warn the user that the constraints are not being preserved. When the semaphore is locked, it does not allow the user to perform any actions that
3.3.4 Import and Export

The GUItarXML format is the default format for importing and exporting in GUItar. However, GUItar can export to various different formats such as GraphML, dot and VauCanSon-G. This is mainly achieved by first exporting to GUItarXML and then using conversion methods to convert GUItarXML to the desired format. The import method is analogous. The GUItarXML format will be described in more depth in Chapter 4. Format conversions will be described in Chapter 5.

3.3.5 Foreign Function Calls

The Foreign Function Call (FFC) mechanism is the extension mechanism of GUItar. It allows calling external methods from GUItar through a Python API. This mechanism
is configured by XML specifications that contain information about the methods, such as their arguments and return values. FFC methods can also have GUItar menus associated to them, and these menus can be context-sensitive. This means that some menus might only be enabled when, for example, DFA diagrams are present. This mechanism will be described in more detail in Chapter 6.

### 3.3.6 Scripting and Console

GUItar has some basic scripting capabilities. GUItar scripts are Python scripts that can control GUItar’s objects. GUItar’s console also allows the user to run Python commands and have access to GUItar’s internal objects. The console has basic auto-complete features and context-sensitive help. Figure 3.8 shows the GUItar console.

```python
>>> o=CreateObject([x], creatorname="FAdo")
>>> o2 = o.minimal()
>>> o2
DFA({{set([0]), set([2]), set([1])}, set([a', 'b']), 0, set([0]), {0: {a': 1, 't
2, 'b': 1}, 2: {'a': 0, 'b': 2)}})
>>> |
```

Figure 3.8: GUItar console

GUItar’s scripting features will be described in more detail in Chapter 7.
Chapter 4

GUItarXML

4.1 Introduction

GUItarXML is an XML format for the description of diagrams and is based on the GraphML format. GUItarXML can be used to describe graphs, digraphs or any other kind of graph-like diagram, such as automata. GUItarXML can not only represent the structural data of the diagram, but it can also represent its graphical information and styling information. Despite GraphML’s key/value extension mechanism, it was chosen not to use that mechanism to include the additional data that GUItar required. For efficiency and clarity reasons, that data is encoded directly as new elements.

4.2 Structure of a GUItarXML Document

A GUItarXML document can contain an arbitrary number of graph elements. The graph elements contain the diagram’s structure (the nodes and the edges), and may contain some automata specific data. Styling data is encoded in style elements and state_object_group elements. Each GUItarXML document may have an arbitrary number of each of them. The style elements contain style data. Styles include
Figure 4.1: Top level structure of a GUItarXML document

colors, text fonts, line widths, or arrow head shapes that can be applied to graphical objects. The state object group elements contain the structure of a node graphical object, and rely on style elements for their styling. Figure 4.1 shows the RNC schema of the structure of the top level of a GUItarXML document.

4.2.1 Nodes and Edges

A node element represents a node and an edge element represents a transition. They both must have an id attribute that must be an integer, unique within the graph. Edges must additionally contain source and target attributes, that are the identifiers
of the source and the target node, respectively. Nodes and edges also have the sub-elements diagram_data, draw_data, label and automata_data, which are all optional. Figure 4.2 shows the RNC schema for nodes and edges.

4.2.1.1 diagram_data

```xml
node_diag = element diagram_data{
  attribute x {text},
  attribute y {text}
}
```

Figure 4.3: Node diagram_data

The diagram_data element contains diagram specific data. For nodes, it contains the diagram (abstract) coordinates of the node (attributes x and y). This element is currently empty for edges. Figure 4.3 shows the RNC schema for the node’s diagram_data.

4.2.1.2 draw_data

The draw_data element contains graphical data. For nodes, it contains the “world” coordinates of the node (in pixels), the scale of the node’s graphical components, and the name of the state_object_group to apply to it. A state_object_group element with that name must exist in the document, or else the default value is assumed, instead. Figure 4.4 shows the RNC schema for the node’s draw_data. For edges, draw_data has the following attributes:

- **arrowlinestyle**: style to apply to the arrow’s line;
- **head1style**: style to apply to the arrow’s first head (head on the target side);
- **head2style**: style to apply to the arrow’s second head (head on the source side);
node_draw = element draw_data {
    attribute x {text},
    attribute y {text},
    attribute scalex {text}?,
    attribute scaley {text}?,
    attribute obgroup {text}?
}

Figure 4.4: Node draw_data

- **numberofheads**: number of heads the edge has; can be 0, 1 or 2;

- **labelside**: side of the label; if positive, the label is placed on the left side of the edge;

- **labelperc**: label position along the edge, given as a percentage; an 0 places the label next to the source node; an 1 places label close to the target node;

- **defaultdist**: when there are multiple edges stacked on top of each other, this is the distance, in pixels, to keep between them;

- **snapposition**: index of the position the label is snapped to;

- **middlepoint**: index of the point considered the “middle” point;

- **snapped**: if True, the edge is “snapped”;

- **loopangle**: orientation of the loop in radians; only used in loops;

- **loopradius**: distance from the center of the loop to the “middle” point; only used in loops;

- **loopintersection**: distance from the center of the loop to the center of the node, in pixels; only used in loops.

For each of the styles (head or line), a style element with that name must exist in the document, or the default value is assumed. The edge’s draw_data element must
4.2. *STRUCTURE OF A GUITARXML DOCUMENT*

edge_draw = element draw_data {
    attribute arrowlinestyle {text}?,
    attribute head1style {text}?,
    attribute head2style {text}?,
    attribute numberofheads {"0" | "1" | "2"}?,
    attribute labelside {text}?,
    attribute labelperc {text}?,
    attribute defaultdist {text}?,
    attribute midpoint {text}?,
    attribute snaped {"True" | "False"}?,
    attribute straightline {text}?,
    attribute loopangle {text}?,
    attribute loopradius {text}?,
    attribute loopintersection {text}?,
    point*
}

Figure 4.5: Edge draw_data
also contain at least three point sub-elements. These elements have the coordinates (attributes X and Y) of the start point of the line, the control points, and finally, the end point of the line. There must be at least one control point, but there can be an unlimited number of them. Figure 4.5 shows the RNC schema for the edge’s draw_data.

4.2.1.3 label

```
label = element label {
  attribute type {"Simple" | "Compound"},
  attribute layout{text},
  attribute style{text},
  dict*
}
dict = element dict {
  attribute key{text},
  attribute value{text}
}
```

Figure 4.6: Label specification

The label element contains the label’s data. Figure 4.6 shows the RNC schema for labels. Labels can be either simple or compound. Simple labels are just strings, while compound labels can have a structure. The layout attribute contains the label’s layout if the label is compound, or the label’s text, if the label is simple. The layout of compound labels may have keys (words starting by $) that must have a corresponding dict element. The value attribute of that element is the value of that key, in the label. Figure 4.7 shows an example of a label on a GUltarXML document 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.
### 4.2. STRUCTURE OF A GUITARXML DOCUMENT

#### 4.2.1.4 automatata_data

The `automata_data` element contains automata specific data. Currently only nodes use it to indicate if the node is initial or final. Figure 4.8 shows the RNC schema for the `automata_data` element of nodes.

```xml
node_automata = element automata_data {
    attribute initial {"0" | "1"},
    attribute final {"0" | "1"}?
}
```

Figure 4.8: Node automata_data

#### 4.2.2 Graph’s automatata_data

The `graph` elements also have an `automata_data` element. This element has the `sigma` element where the alphabet of the automaton can be encoded. The `sigma` element can have multiple `symbol` sub-elements, each having a `value` attribute with one of the members of the alphabet. The `automata_data` elements also has a `classification` element. This element has multiple `class` elements, which have a `value` element. They are used to indicate the type (or types) of the diagram. Figure 4.9 shows the RNC schema of the `automata_data` of graphs.

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

Figure 4.7: Compound label example
CHAPTER 4. GUITARXML

graph_automata = element automata_data{
  element sigma{
    element symbol{
      attribute value {text}
    }*
  }?,
  element classification{
    element class{
      attribute value {text}
    }*
  }?
}

Figure 4.9: Graph’s automata_data

style = element style{
  styledata
}
styledata = (  
  attribute name {text},
  attribute basestyle {text}?,
  attribute wxfillstyle {text}?,
  attribute wxlinestyle {text}?,
  attribute linewidth {text}?,
  attribute arrowangle {text}?,
  attribute arrowsize {text}?,
  attribute cornerssize {text}?,
  fill_color?,
  line_color?,
  font?
)

Figure 4.10: Style data
4.2.3 Style Data

Styles are graphical properties that can be applied to nodes or edges, and are defined in style elements. Figure 4.10 shows the RNC specification for styles. They must have a name attribute that must be unique in the entire document, and it is used to identify the style. Styles can also have the following attributes:

- **wxlinestyle**: line style; can be “Solid”, “Transparent”, “Dot”, “LongDash”, “ShortDash” or “DotDash”;

- **wxfillstyle**: fill style; can be “Solid”, “Transparent”, “BiDiagonalHatch”, “CrossDiagHatch”, “FDiagonalHatch”, “CrossHatch”, “HorizontalHatch” or “VerticalHatch”;

- **linewidth**: width of the line, in pixels;

- **arrowangle**: angle of the arrow’s head opening in radians,

- **arrowsize, cornerssize**: define arrow head properties. see Figure 4.11;

- **fillcolor** element: fill color;

- **linecolor** element: line color;

- **font** element: font data.

![Figure 4.11: Arrow head](image)
CHAPTER 4. GUITARXML

Styles do not need to specify all attributes. The `basestyle` attribute can be used to indicate the name of a style to inherit properties from. If a style has the `basestyle` attribute set to “redstyle”, for example, that style will inherit all properties from the style called “redstyle”. All properties specified on that style will override the values of the base style. Nodes can be a composition of many sub-objects that are defined in `state_object_group` elements. `state_object_group` elements must have a `name` attribute that is used to identify them. `state_object_group` elements can have multiple “shape” sub-elements. These shapes can be `ellipse`, `rectangle`, `floatingtext` or `arrowspline`. “Geometric” shapes (ellipse or rectangle) have the `scalesize` attribute, that indicates if the size of the object is automatically scaled to the size of the label, and a `size` element. The `floatingtext` elements have a `text` attribute, that contains the text of the label. The `arrowspline` has elements for the style of the line and the head. Additionally it has `point` sub-elements, just like `edges`. All shapes may have the following attributes:

- **typename**: the name of the style to apply to the shape;

- **scaleposition**: if “True”, the position of the shape will be scaled depending on the size of the label;

- **scalesize**: if “True”, the size of the shape will be scaled depending on the size of the label;

- **position**: position of the shape, relative to the center of the object;

The `state_object_group` elements may have the attribute `marginobjectindex` that identifies which object the edges attach to. Figure 4.12 shows an example of a node style. The node is composed of two concentric ellipses. The external ellipse uses the style “reddot”, which changes the fill and the line color to red, and changes the line style to “Dot”. Figure 4.13 shows the rendering of a node using that style.
4.2. STRUCTURE OF A GUITARXML DOCUMENT

...<style name="reddot" basestyle="default" linewidth="2" fillstyle="Dot">
   <fillcolor r="255" g="0" b="0"/>
   <linecolor r="255" g="0" b="0"/>
</style>
...

<state_object_group name="finalred/">
   <ellipse style="default" scalesize="True">
      <size x="35" y="15"/>
   </ellipse>
   <ellipse style="default" scalesize="True">
      <size x="40" y="20"/>
   </ellipse>
</state_object_group>
...

Figure 4.12: Example node style

Figure 4.13: Rendering of the style in Figure 4.12
Chapter 5

Format conversions

5.1 Introduction

GUltar uses the GUltarXML format as its main export format, but it is also used as means of converting GUltar diagrams into other formats. This is achieved by first exporting to GUltarXML and then using a conversion method to convert from GUltarXML to the desired format. The importing method is analogous.

Currently GUltar can export to GraphML, dot, FAdo and VauCanSon-G, and can import from all of the previous formats, except VauCanSon-G. Exporting to GraphML is simple since GUltarXML is an extension of it. The conversion is done via an XSL transformation that removes the extra elements present in GUltarXML. Node coordinates and labels, both for nodes and edges, are included using the GraphML key/value mechanism. In the future, this export method will be improved to include all the data present in GUltarXML.

The dot export method uses the pyGraphViz [PyG09] Python package to create a dot document. The dot export is currently in an experimental phase, and only considers the structural data of the diagram. Exporting to FAdo format is done by converting the diagram into a FAdo DFA or NFA object (depending on the diagram type), and
then using FAdo’s internal export method to write the file.

## 5.2 VauCanSon-G Export

The VauCanSon-G export method is the most complex method implemented. It outputs a \LaTeX\ document with a VCPicture environment containing the automaton. The method tries to produce an exact rendering of the drawing present in GUltar, but when an exact conversion cannot be made, a reasonable approximation is done. For example, VauCanSon-G has no native support for rectangular states, so regular round states are used instead. Also, the method does not have full support for all of the VauCanSon-G features. For example, it does not support zigzag edges or parameterized arcs. The method provides a few customization options, as can be seen in Figure 5.1.

![VauCanSon-G export dialog](image)

Figure 5.1: VauCanSon-G export dialog

The options it provides are:

- Default state size: allows choosing the default size for states;
• Loop orientation: allows choosing if loops will use absolute angles or if they will use cardinal directions;

• Global scaling: if scale to page is selected, the method will ensure that the image does not exceed the size of a default A4 \LaTeX article text area (approximately 12cm by 19cm). In the future, an option to scale the image to fit inside a user-specified box may be given;

• State Scaling: if the “use $\text{\textbackslash VarState}$” option is selected, VauCanSon-G $\text{\textbackslash VarState}$ will be used when the size of the state label exceeds a size specified by the user;

• Styles: allows exporting styles.

Styles are exported as \LaTeX macros that set various VauCanSon-G styling properties. When necessary, new \LaTeX colors are also defined and included in the document. These styles are applied by calling the macro before declaring the state or the edge.

5.3 XPort

The XPort mechanism allows for a simple way of adding new export and import methods to GUItar, coded either as Python methods or XSL transformations. A menu entry for every method will be added under GUItar’s import or export menu. Figure 5.2 shows the XPort RNC specification. The mechanism is configured using an XML specification that allows multiple XPort definitions per document, defined in xport elements for Python methods, or xslxport elements for the XML transformations. xport and xslxport elements both must have a name attribute, that is the string that will appear in the menu, and can, optionally, have a wildcard attribute that is the file wildcard that will be used in the file dialog. Depending on the type of XPort, additional attributes and elements may be required. The xport elements must have the import attribute, that is the Python import statement for the module containing the methods. XPort elements must have export and import sub-elements. These
Figure 5.2: XPort specification
sub-elements have a **method** attribute, that is the name of the method for export or import. **import** and **export** elements can optionally have the **customdialog** attribute that is a name of a custom dialog class to use when activating the method. The export method must be a method that only receives two argument: a **GUItarXML** string and a string with the path to export to. The import method receives only one argument, the path to import from. The **xslxport** elements must have the **expfile** and **impfile** attributes that are the paths for the XSL file containing the export transformation and the import transformation, respectively. Figure 5.3 shows the **XPort** definition used for **VauCanSon-G** and **GraphML** in **GUItar**.

```xml
<xport_data>
  <xslxport name="GraphML" impfile="graphml−guitar.xsl"
           expfile="guitar−graphml.xsl" wildcard="xml files (*.xml)|*.xml"/>
  <xport name="Vaucanson experimental" import="vaucanson" wildcard="tex files (*.tex |*.tex)"
         expfile="guitar−graphml.xsl" wildcard="xml files (*.xml)|*.xml"/>
  <export method="ExportVaucanson" customdialog="VaucansonDialog"/>
</xport>
</xport_data>
```

Figure 5.3: XPort definition for VauCanSon-G and GraphML
Chapter 6

Foreign Function Calls

6.1 Introduction

The Foreign Function Call (FFC) mechanism provides GUltar with a generic interface to external libraries or programs, and mechanisms to interact with foreign objects. In the first case (called Module FFC), the FFC mechanism calls functions directly from external modules. These modules can be any type of module that can be imported into Python. In the second case (Object FFC), GUltar can deal with foreign objects and call their methods, as long as there are methods to create those objects and convert them back into GUltar. This functionality is implemented by the Object Creators. The entire mechanism is configured by an XML specification that specifies things such as the name of the methods, their arguments, and their return values. The FFC mechanism also includes a facility called the Object Library, that is used to track FFC operations.
6.2 XML Specification

6.2.1 Top Level

```xml
ffc = element foreign_function_call{
    attribute silent_dependency_fail {"True" | "False"}?,
    (depends | path)*,
    (mod | ob)*
}
```

Figure 6.1: Top level FFC RNC specification

FFC configuration files can contain several FFC definitions (module or object). Figure 6.1 shows the top level specification of a FFC configuration file. The root element (foreign_function_call) has the attribute silent_dependency_fail, that, if True, means that GUltar should not raise any error if any of the dependencies for this FFC are not met. The root element can have multiple path or depends elements. The path elements have the attribute value that is used to add paths to GUltar in the case that FFC modules are located in non-standard paths. The depends elements are used to indicate the Python modules that the FFC depends on and only have an import attribute that must contain the name of a module. The root element may have multiple module and object elements. The module elements represent one module FFC and have the import attribute that is the statement used to import the module in Python. The module elements also have the name and description attributes that are a “user-friendly” name and description for the module. The module elements can have multiple method sub-elements and one MenuData element that will be described in subsections 6.2.2 and 6.2.3, respectively. The object elements represent Object FFCs and have the creator attribute that is the name of the Object Creator used to create the object that will contain the methods of this FFC. Object creators will be explained in more detail in Section 6.3. Like in module FFCs, object elements also have a name and a description attribute and can have multiple method elements and
6.2. XML SPECIFICATION

6.2.2 Methods

Figure 6.2: Method specification

FFC methods are defined in method elements. Methods have a name attribute, that is the name of the method as it is defined in the module or object. method elements have an id attribute that is an unique identifier for this method and that will be used in the menu definitions. The id attribute allows having different definitions for the same method that, for example, have a different number of arguments. Methods also have a friendly_name attribute and a description attribute that are the name and the description of the method that will appear in the FFC dialog when it is called in GUItar. Methods may have multiple argument and return_value elements. The argument elements contain information about the method’s arguments and the return_value elements contain information about the values returned by the method. Both have the type attribute that can be one of the following:

- Int;
- Float;
• Bool;

• String;

• File: Requires the additional attributes `diagmode` that indicates the type of dialog to use ("Save" or "Load") and the `filemode` attribute, that indicates if the method expects a path or a live file object;

• Canvas: a `GUltarXML` string;

• Object: foreign object that requires the additional attribute `creator` that is the name of the object creator to use;

```
argument = element argument{
    attribute type {text},
    attribute default_value {text}?,
    attribute use_default {"True" | "False"}?,
    attribute requires {text}?,
    (fileargdata | objectargdata)?
}
fileargdata = ( attribute diagmode {"Save" | "Load"}?,
                attribute filemode {"Path" | "File"}? )
objectargdata = (attribute creator {text})
return = element return_value{
    attribute type {text}?,
    (fileargdata | objectargdata)?
}
```

Figure 6.3: Argument and return value specification

Arguments may have the `default_value` attribute, that is the default value for that argument. “Canvas” arguments have special default values that can be:

• Current: use current canvas;
• First: use canvas on first page;
• Last: use canvas on last page;
• Next: use canvas on next page;
• Previous: use canvas on previous page;

If the use\_default attribute is True, the default value is used and the user is not prompted for a value. Figure 6.4 shows an example FFC for FAdo's DFA object Minimization and Intersection methods.

```xml
<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>
    <method name="and" id="and" friendly_name="Intersection"
             description="Returns intersection of two automata">
      <argument type="Object" creatorname="FAdoDFA"/>
      <return_value type="Object" creatorname="FAdoDFA"/>
    </method>
  </object>
</foreign_function_call>

Figure 6.4: Example FFC definition

6.2.3 Menus

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.

Menu\_Data elements may have multiple Menu sub-elements. Each may have a title attribute, that is the name of the menu. If a menu with the same title already exists,
CHAPTER 6. FOREIGN FUNCTION CALLS

menu = element Menu {
  attribute title {text},
  attribute pos {text}?,
  attribute requires {text}?,
  (menu_entry | menu_sep | menu) *
}

menu_entry = element Menu_Entry {
  attribute descr1 {text},
  attribute descr2 {text}?,
  attribute action {text}?,
  attribute type {"normal" | "radio" | "check" | "sep"}?,
  attribute accel {text}?,
  attribute pos {text}?,
  attribute requires {text}?,
}

Figure 6.5: Menu specification

the contents of this menu are appended to it. They may have a pos attribute, that is
the position of the menu on the menu bar and a requires attribute that has a comma
separated list of the names of the classes the currently displayed diagram must belong
to for the menu to be enabled. Each Menu may have multiple Menu_Entry or Menu sub-
elements. Menu_Entry elements are single entries on the menu while Menu elements
are sub-menus. Menu_Entry elements have the following attributes:

- **descr1**: text that will appear in the menu entry;

- **descr2**: help text that will appear in the status bar when the mouse hovers over
  the menu entry;

- **action**: method to execute. The value of this attribute must be the id of a
  method;

- **type**: can be normal, check, radio or sep. A check value makes an entry with a
  checkbox. A radio value makes an entry that is part of a set of mutually exclusive
options (selecting one deselects the others). A sep value makes a separator and all other attributes are ignored;

- **accel**: key combination (accelerator) used to activate the menu;
- **pos**: position of the entry inside the menu;
- **requires**: same as in Menu elements;

### 6.3 Object Creators

Foreign objects are any type of value that is not internal to GUItar. They may be returned by FFC methods or may be required as arguments for an FFC method. Therefore, there are two processes that must be considered when handling foreign objects: creating the objects and converting them back into values that GUItar is able to work with.

The Object Creators were implemented for this purpose. They require a module containing methods for creating foreign objects and methods to convert them back. They are configured using an XML specification. Figure 6.6 shows an example of the Object Creator functionality. The FAdo method nfaT creates an NFA from a regular expression object. GUItar uses the regexp object creator to create a regular expression object.
object. After calling the method, it uses the NFA object creator to convert that automaton into a GUltarXML string that can be interpreted and imported by GUltar.

### 6.3.1 XML Specification

Object Creator configuration files may specify many object creators, as long as all of the methods are present in the same module. The module is specified by the import attribute, that must contain the Python import statement used to import the module.

```xml
start = element object_creator_group{
    attribute import {text},
    attribute silent.depenendency.fail {
        "True" | "False"
    },
    (depends|path)*,
    obc *
}

obc = element object_creator{
    attribute name {text},
    attribute classname {text},
    element to_method{
        attribute method {text},
        argument*
    },
    element from_method{
        attribute method {text},
        return*
    }
}
```

Figure 6.7: Object Creator specification

Object Creator definitions may have the silent.depenendency.fail attribute, path and depends elements, just like in FFC definitions. The root element can have multiple object_creator elements that must have a name attribute. The name attribute is used in FFC method arguments and return values for the creatorname attribute of object

```xml
start = element object_creator_group{
    attribute import {text},
    attribute silent.depenendency.fail {
        "True" | "False"
    },
    (depends|path)*,
    obc *
}

obc = element object_creator{
    attribute name {text},
    attribute classname {text},
    element to_method{
        attribute method {text},
        argument*
    },
    element from_method{
        attribute method {text},
        return*
    }
}
```
types or in the in the creator attribute of Object FFCs. object_creator elements
must have a classname attribute that is the name of the class the Object Creator can
handle. The Object Creator must have a to_method element that contains the name of
the method used to create the object in the method attribute. to_method may have
multiple argument elements that are the arguments of that method. It must also have
a from_method element with the name of the method used to convert the object back
to GUItar and may have multiple return_value elements with the values returned by
the method.

Figure 6.8 shows an example of an Object Creator specification for the FAdo DFA objects.

```xml
<object_creator_group import="GF">
  <object_creator name="FAdoDFA" class="DFA">
    <to_method method="GuitarToFA"/>
      <argument type="Canvas"/>
    </to_method>
    <from_method method="FAToGuitar"/>
      <returns type="Canvas"/>
    </from_method>
  </object_creator>
</object_creator_group>
```

Figure 6.8: Object Creator example

### 6.4 Object Library

The Object Library is a component of the FFC mechanism that stores objects created
and returned during the execution of an FFC method. The objects may be recalled
for future FFC calls. The Object Library allows viewing a graphical representation
of the relationships between objects. Objects are related if one or more objects
originated other objects by applying some function. There are two ways of displaying
this information. The first one is by displaying a tree of objects that originated the
object in the current page. This tree displays the current object in the top and it’s parents below it. The panel on the right shows a string representation of the value of the object and the method that originated it. The second way is to display a graph that shows the relationships between all objects. This graph is drawn using GUltar’s own canvas. In the following examples, the method nfaPD was applied to an object and then the method minimal was applied to the result of the first methods. Figure 6.9 shows the tree of object relationships. Figure 6.10 shows a graph of object relationships.

![Object and operations tree](image)

**Figure 6.9: Object and operations tree**

![Relationship Graph](image)

**Figure 6.10: Relationship Graph**
Chapter 7

Scripting

7.1 Introduction

GUItar provides scripting facilities based in a Python API. Scripts have access to the GUItar interface by means of the GUItar frame object. Scripts can be manually created or created by GUItar’s script recorder.

Figure 7.1: GUItar script recorder controls

Figure 7.1 shows the script recorder controls. When record is pressed, the script manager listens to events generated by GUItar and stores them in an internal format. If record is pressed again, the script can be saved. The pause button pauses the recording until it is pressed again. The stop button stops the recording process and discards all recorded data. Scripts can be called from the Script Manager or can be run on startup by using GUItar’s -s option (for example, “python Guitar.py -s script.py”). The script recorder is still in its initial development phase and currently only detects Add Node and Add Edge events. GUItar also provides a console that is able to interact with the graphical interface using the same API as the scripts. The
console is implemented using wxPython’s py.shell class.

7.2 GUItarSimpleAPI

GUItar scripts can access any GUItar object. A simplified API called GUItarSimpleAPI was developed and provides the following methods:

- **AddNode**: Adds a node; can have the following optional arguments:
  - *coords*: the two coordinates;
  - *id*: node identifier;
  - *label*: node label;
  - *style*: name of the style to apply to the node;
  - *convertcoords*: if True, diagram coordinate units are used; otherwise, coordinates are in pixels;
  - *undo*: if True, this action is added to the undo stack, making it possible to undo;
  - *page*: number of the notebook page to add the node to;

- **AddEdge**: Adds a transition; has two mandatory arguments: the identifier of the source node and the identifier of the target node; also has the following optional arguments (same meaning as in AddNode): *id, label, style, undo*, and *page*;

- **ConvertToXML**: Returns a GUItarXML string of the current diagram;

- **Draw**: Receives a GUItarXML string as argument and draws it on a new canvas;

- **CreateObject**: Creates a foreign object; First argument is the list of arguments required by the Object Creator for that foreign object; Second argument is the name of the Object Creator to use;
• **UncreateObject**: converts a foreign object into a GUltar object; Requires an Object Creator compatible with the object;

• **DrawObject**: Draws a foreign object; If the result of converting the object is a GUltarXML string, this method draws it on a new notebook page; Otherwise it just prints the result; Requires an Object Creator compatible with the object.

Figure 7.2 shows a script that generates a $K_5$ graph, a complete graph with five vertices. Figure 7.3 shows the result of running the script in GUltar.

```python
for x in range(5):
    AddNode(id=x, label="s"+str(x))
for x in range(5):
    for y in range(x+1,5):
        AddEdge(x, y, style="Line")
```

Figure 7.2: Script Example

![Figure 7.2: Script Example](image)

Figure 7.3: Result of running the script in 7.2

![Figure 7.3: Result of running the script in 7.2](image)
7.3 Console

The GUItar console is a Python shell that also allows interaction with GUItar with the same expressiveness as the scripts. For example, the console can be used, to convert the currently drawn diagram into a foreign object (for example, a FAdo DFA object). The object can be manipulated as it would be in a usual Python session. After performing the manipulations, the object can be imported back to GUItar. Figure 7.4 shows an automaton being converted into a FAdo object using the console. The ConvertToXML function is used to retrieve a string with a GUItarXML representation of the currently drawn diagram. The CreateObject function is used to create a FAdo object. Figure 7.5 shows the object being manipulated and then drawn in GUItar. The DFA is first minimized and then inverted. The DrawObject function is used to draw the object in GUItar.
Figure 7.5: Console object being drawn in GUltar
Chapter 8

Conclusions

This work presents the GUltar application, an interactive graphical environment for the visualization and manipulation of automata diagrams. GUltar has tools that can simplify diagram drawing, like the assisted drawing features and semaphores. It also provides powerful style creators that give the users the freedom of creating their own graphical styles to fit their needs.

This work mainly focuses on the mechanisms implemented in GUltar to make it extensible and able to interact with external tools for diagram manipulation. The GUltarXML format is the default export format of GUltar. GUltarXML contains the structural data of the diagram and styling information. GUltarXML is expressive enough to be used as an intermediate format for conversions to other formats. The FFC mechanism allows the integration of GUltar with external diagram manipulation tools, ensuring GUltar’s modularity. Currently, part of the FAgo library is already integrated in GUltar via the FFC mechanism as well as the FAgoo library, that provides some automatic diagram layout algorithms. However, the FFCs still need some user interface improvements. The Object Library still needs to be improved to be able to infer certain properties of diagrams (like if they are minimal or complete) from the methods applied to them and a language must be developed to be possible to represent the object relationships. GUltar’s scripting framework and the console, allow a large
degree of automatization and control over GUltar that would be difficult with the mouse alone. The script recorder, although not yet finished, provides GUltar with an automated tool for script generation.

As for future work, the GUltar application must continue to be enhanced. More export and import methods must be developed. Methods for exporting to SVG and FSMXML are planned. The manual creation of FFC XML configuration files is difficult, especially for modules or objects that contain many methods, therefore, an automated tool for the task of generating FFC configuration files must be developed. The FFC mechanism must also be improved with more functionality. A new kind of event-driven FFC is planned, which would allow GUltar to respond to events originated from an external module or object, or allow the external module to respond to GUltar events. Algorithm animation capabilities and a visual programming environment are also planned for GUltar.
Bibliography


[Cor09] Microsoft Corporation. Microsoft Office. 


http://www.lsv.ens-cachan.fr/ gastin/gastex/gastex.html, 
Access date: 29.6.2009.


[Ope09] OpenOffice.


[PyG09] PyGraphViz.


[Tea09] The LyX Team. LyX.


[wXP09] wxPython.
[XGM09] XGMML.  
