Proposal: an XML representation for automata

The Vaucanson Group <vaucanson@lrde.epita.fr>

November 15, 2004

Abstract

This paper presents the XML formalism that was introduced at the CIAA 2004 conference to represent automata. This formalism was created to fulfill the need that was expressed at CIAA 2003 to have a standard for exchanging automata between various applications.

This format allows the user to describe many kinds of automata, including weighted automata and transducers. In order to achieve maximal genericity, a file is mainly divided into two parts: one to express the content of an automaton, and another one to specify the kind of automata the file refers to. Furthermore, extra data may be attached to an automaton, such as layout information. Using this formalism it is also possible to put multiple automata in a unique file.

An implementation of this formalism was created as an experimental feature in the Vaucanson software platform, which is a framework dedicated to automata manipulations.

Introduction

Various computer programs manipulate automata. Among them exist finalized products, experimental software, programs designed to manipulate automata as a final purpose or as intermediary computational tools. Some of them need to store their automata in order to reuse them in a latter execution, while other just need to dump some debugging informations. It is noteworthy that complementary programs designed to manipulate automata need to have a common representation. As an example, a program designed to edit automata must produce an output that is compatible with other programs designed to render automata.

Therefore, there is a strong need to have a standard language designed to represent automata. Although a domain specific language, it must be powerful enough to represents the numerous kinds of automata that were invented. This need was primarily expressed at the CIAA 2003 conference.

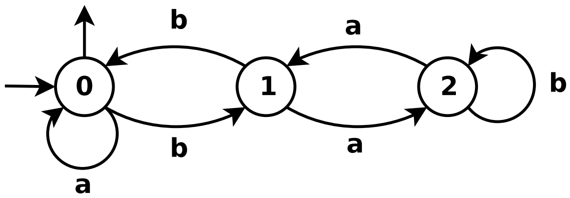
In order to fulfill this need, the Vaucanson developers designed and began to implement an XML formalism to describe automata. Vaucanson is a generic software platform designed to manipulate automata (including weighted automata) and transducers. A proposition of their formalism was introduced at the CIAA 2004 conference. This paper documents this formalism and is a written version of the presentation that was made.

This article is divided in two parts: first the basic and mandatory features that are needed to describe an automaton are presented. Then, a second part introduce some extra features of the formalism, such as layout information and session files.

1 Basic features

As decided at CIAA 2003, XML have been chosen. It brings developers an easier design process and simplifies the implementation task a lot, since many libraries already exist to parse XML. From an end-user point of view, XML is a well known format and should be grokked more easily than any custom format. One may argue that XML is extremely verbose and that, with big automata, huge files may be produced. However, working on compressed XML file is a common practice and should be an easy way of solving this issue. But still there is an overhead due to the time needed to decompress the file.

At top level, the <`automaton`> tag is used to enclose the definition of any automaton. Then the description is divided into two parts: content and type. An example is given in figure 1. The content section refers to the automaton's states, transitions, and related information, whereas the type section refers to the kind of automaton that is described: Boolean or weighted automaton, transducer and so on.




```
<automaton>

  <type> ... </type>

  <content> ... </content>

</automaton>
```

Figure 1: Base structure of an automaton.

First, the information that is expected to be found in the content section will be presented. For this purpose, the Boolean automaton of figure 1 will be used. Then the focus will be made on the type section.

1.1 Content

In order to describe the content of an automaton, the user must provide information about four kinds of objects: states, transitions, initial states and final states. An example is shown in figure 2.

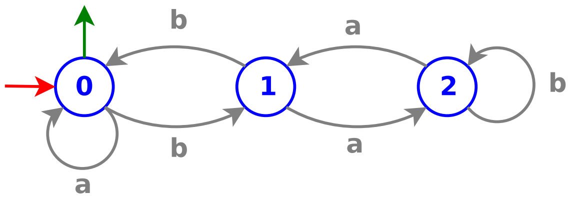
1.1.1 States

The declarations of the states for the automaton of figure 1 is shown in figure 3. It is performed inside a <`states`> tag using <`state`> tags. Each state should be assigned an unique name using the `name` attribute. This name may be any string, and will be used latter to refer to transitions's extremities.

1.1.2 Transitions

The declarations of the transitions for the automaton of figure 1 is shown in figure 4. In a similar way as states, it is done using <`transitions`> and <`transition`> tags. The `src` and `dst` attributes should refer to valid state names inside the states section. Of course, these attributes indicate which are the source and destination of each transition.

The `label` attribute is the label of the transition, in a textual representation. This representation may vary with the kind of automaton which is described. As an example, a "classical" Boolean letter automaton will have only one-letter labels whereas a transducer may have labels of the form `"a|b"`. Transitions upon an empty word (*i.e.* epsilon transitions) may be represented using an empty string, the symbol `"1"`, or by not specifying the `label` attribute.




```
<automaton>

  <type> ... </type>

  <content>

    <states> ... </states>
    <transitions> ... </transitions>
    <initials> ... </initials>
    <finals> ... </finals>

  </content>

</automaton>
```

Figure 2: The content tag.

```
<states>
  <state name="0" />
  <state name="1" />
  <state name="2" />
</states>
```

Figure 3: States declaration.

```
<transitions>
  <transition src="0" label="a" dst="0" />
  <transition src="0" label="b" dst="1" />
  <transition src="1" label="a" dst="2" />
  <transition src="1" label="b" dst="0" />
  <transition src="2" label="a" dst="1" />
  <transition src="2" label="b" dst="2" />
</transitions>
```

Figure 4: Transitions declaration.

```
<initials>
  <initial state="0"/>
</initials>

<finals>
  <final state="0"/>
</finals>
```

Figure 5: Initials and final states declaration.

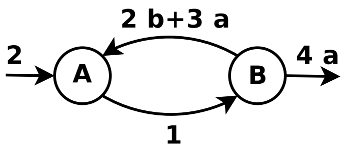
1.1.3 Initial and final states

Initial and final states are declared both in a very similar fashion. As for states and transitions the user must create appropriate sections using the <initials> and <finals> tags. Inside those sections, he may declare initial and final states using the <initial> and <final> tags. An example for the automaton of figure 1 is given in figure 5.

Each <initial> or <final> tag must be accompanied of a name attribute containing a valid and declared state name. This will be the state the tag refers to.

Using special tags to indicate the initial or final status of a state allow the user to give extra information about this status, and is more consistent than just putting some extra attributes to the <state> tag. As an example, when working with weighted automata or transducers, it is more natural to think of initial states as state which are the destination a special kind of transition, with no source, but which may have an arbitrarily complex label. Furthermore, having four sections for states, transitions, initial and final states makes the XML formalism closer to the mathematical definition of an automaton.

1.1.4 Complex example




```
<states>
  <state name="A" />
  <state name="B" />
</states>

<transitions>
  <transition src="A" dst="B" />
  <transition src="B" label="2 b+3 a" dst="A" />
</transitions>

<initials><initial label="2" state="A"/></initials>

<finals><final label="4 a" state="B"/></finals>
```

Figure 6: Advanced usage of contents.

Figure 6 shows a more complex example for different tags that may occur inside the content section. The first thing to be noticed is the name of the states, which are not integers. Then, an empty transition is declared just by not specifying any `label` attribute, and another transition has a rational expression as label. Finally, since the example is a weighted automata, initial and final states my have weights, and even be labeled with rational expressions.

As this example shows, the format for the content section remains rather simple, even for non-trivial structures. It has a strong expressiveness and allows the user to specify different kinds of automata.

1.2 Type

Since the formalism proposed here may be used to describe various types of automata, there is a need to describe the context within which the content information is relevant. In other words, an automaton is not just a content, but also has a type. The aim of the type section is therefore to specify the algebraic context of the described automaton.

First the formalism will be indicated for "usual" Boolean automata, and then some more complex examples will be given.

1.2.1 Specifying the monoid.

When specifying a Boolean automaton, the only relevant information that cannot be guessed by the computer is the monoid the automaton is declared on. Therefore, the type section for such an automaton just contain a <`monoid`> tag, as shown in figure 7.

```
<type>
  <monoid>
    <generator value="a"/>
    <generator value="b"/>
  </monoid>
</type>
```

Figure 7: A basic type section for Boolean automata over $\{a, b\}$.

There should always be a monoid section inside a type section. When no attributes are specified, a monoid is considered to be a free monoid. Thus, <`generator`> tags may be used inside the monoid section to specify the monoid generators which, in case of free monoids, are letters. The textual representation of those letters is indicated using the `value` attribute.

1.2.2 Using special semirings.

When working on weighted automata, a semiring information should be added to the monoid one. An example of type declaration for such an automaton is given in figure 8.

```
<type>
  <monoid>
    <generator value="a" />
    <generator value="b" />
  </monoid>
  <semiring set="Z" operations="numerical" />
</type>
```

Figure 8: A type section for weighted automata.

Since a semiring mathematically consists of a set equipped with two special operations, the XML definition of a semiring consist of declaring a set and some operations. The set is declared using the set attribute of a <semiring> tag, whereas the operations are made using an operations attribute.

Valid sets are: B, Z, R,... Valid operations are: boolean, numerical, tropicalMax, tropicalMin,... The default values for these attributes are B and boolean.

1.2.3 Transducers

As there are two ways of formalizing transducers, there are two way of declaring a transducer in the type section of an XML automaton.

```
<type>
  <monoid type="product">

    <monoid>
      <generator value="a" />
      <generator value="b" />
    </monoid>

    <monoid>
      <generator value="x" />
      <generator value="y" />
    </monoid>

  </monoid>
<type>
```

Figure 9: Transducers using products of free monoids.

Product of free monoids A first solution, presented in figure 9, is to consider a transducer as a Boolean automaton over a product of free monoids. This could be easily achieved using a type attribute for the <monoid> tag. When this attribute is set to product the monoid is considered to be a product of free monoids, and therefore other monoid declarations are expected to be enclosed inside the monoid section. Note that there may be an arbitrary number of free monoids inside the product, and therefore multi-band transducers may be declared.

Rational series as semiring A Boolean series may be used to denote a language. A Boolean series equipped with union and concatenation also constitute a semiring. Therefore, using Boolean series as weights on a letter automaton is a valid approach to define a transducer.

Such a definition is possible using the proposed XML formalism. An example is given in figure 10. The set attribute of the <semiring> tag may be assigned a ratseries value, thus indicating the referred semiring is a rational series. The properties of the series are enclosed in the <semiring> tag, using "classical" <monoid> and <semiring> tags.

As those examples shows, it is possible to represents complex kinds of automata using the proposed XML formalism. Classical automata, weighted automata and multiple kinds of transducers are supported. There is however default values that are provided for simple types, to avoid the user typing complex XML code for simple objects. As an example, defining a boolean automaton just means defining its alphabet, thereby using the <monoid> tag.

Equipped with those features it is possible to describe the strict minimum that is required to work with an automaton. It allows an user to load an automaton in order to run various algorithms, tests and manipulations on it. However, one could expect more from an XML representation for

```
<type>
  <monoid>
    <generator value="a" />
    <generator value="b" />
  </monoid>

  <semiring set="ratseries">
    <monoid>
      <generator value="x" />
      <generator value="y" />
    </monoid>
    <semiring set="B" operations="boolean" />
  </semiring>
</type>
```

Figure 10: Transducers as weighted automata.

automata: it would be helpful to define multiple automata in one file, or to have layout information in order to display automata.

2 Other features

Basically the proposed formalism provides two extra features: the ability to describe an automaton's geometry, and to make session files. These features are presented in this section.

2.1 Geometry

When required to display an automaton, a layout must be used. There exist some specialized applications that compute such a layout (*e.g. Graphviz*) or sometimes an end user would like to choose a custom layout. More than just a layout, it is often useful to specify information such as edges style, states color, and so on.

```
<automaton>
  <geometry StateFillColor="blue" />
  ...
  <states>
    <state name="s0"><geometry x="10" y="10" /></state>
    ...
  </states>

  <transitions>
    <geometry EdgeLineStyle="dashed" />
    ...
  </transitions>
  ...
</automaton>
```

Figure 11: Geometry.

This can be achieved using the <`geometry`> tag, as shown in figure 11. This tag may be placed anywhere in the document and control the way its surrounding bloc should be displayed. Various attributes are available to change different aspects of the automaton. Layout information may be specified using the `x` and `y` attributes, but there exists other attributes such as `StateFillColor` or `EdgeLineStyle`. The attribute names have been chosen to be those of VAUCANSON-G, a LaTeX package dedicated to automata representation.

2.2 Sessions

It is often desirable to store more than one automaton in a file. One may want to run the same algorithm with various entry and save each results, or to store the intermediary automata that are computed by an algorithm at different iterations.

```
<session>
  <automaton>
    ...
  </automaton>

  <automaton>
    ...
  </automaton>

  <automaton>
    ...
  </automaton>
</session>
```

Figure 12: XML sessions.

This is the purpose of the <`session`> tag, that may be used as a top-level tag to enclose various <`automaton`> tags. The so-defined file is called a session and contains multiple automata. An example is given in figure 12.

Therefore this formalism also contains extra features that provide more than a basic information suitable only for automata computations. Graphical applications designed to represent or edit the content of an automaton have a sufficient expressiveness using this formalism, thanks to the <`geometry`> tag. It is also possible to store sessions in one unique file, for programs that need to do so, using a <`session`> tag.

Conclusion

This paper propose an XML formalism to represent automata. This formalism has been designed to be both simple and powerful. Simple, because the notations are intuitive and simple automata may be described easily. Powerful, because it is possible to describe quite complex kinds of automata. In order to achieve this, an XML automaton file is composed of two parts: one that describe the type of automaton that is defined, and one that describe the content of the automaton. It is also possible to define multiple automata in one unique file, and to store geometry information (such as layout data) using the formalism.

During the design of the formalism, an experimental implementation was developed for the VAUCANSON platform, both to provide an experimental playground and to check the consistency of this formalism.

Finally, this XML format allow an user to describe any weighted automaton, including "standard" Boolean automata. Transducers (including weighted and multi-band transducers) may also be described, using two different algebraic views. Furthermore, the format is easily extensible and other features may be added in a later time. The VAUCANSON developers hope to see this formalism spread among other applications that manipulate automata in the future.