An XML Format Proposal for the Description of Weighted Automata, Transducers and Regular Expressions

Akim DEMAILLE [a], Alexandre DURET-LUTZ [a], Florian LESAINT [a], Sylvain LOMBARDY [b], Jacques SAKAROVITCH [c] and Florent TERRONES [a]

[a] *LRDE, EPITA,* {name}@lrde.epita.fr

[b] *IGM, Université Paris Est Marne-la-Vallée,* lombardy@univ-mlv.fr

[c] *LTCI, CNRS / ENST,* sakarovitch@enst.fr

Abstract. We present an XML format that allows to describe a large class of finite weighted automata and transducers. Our design choices stem from our policy of making the implementation as simple as possible. This format has been tested for the communication between the modules of our automata manipulation platform `Vaucanson`, but this document is less an experiment report than a position paper intended to open the discussion among the community of automata software writers.

Keywords. XML format, finite automata, weighted automata, transducers, regular expressions

Introduction

The aim of an interchange format for automata is to make possible, and hopefully easy, the communication between the various programs that input or output such objects.

There exist many kinds of (finite) automata: automata on finite words or on infinite words, automata on tuples of words (often called transducers), weighted automata where the weights can be taken in very different semirings, timed automata, counter automata, pushdown automata, Petri nets, *etc.* The scope of our proposal is restricted to *weighted automata and transducers on finite words.* These automata already form a large family and cover most of the needs in Finite State Machines that are relevant to Natural Language Processing.

To our knowledge, there does not exist any format representing this class of automata. Many tools have devised their own format for reading and writing automata. For instance Grail [1], FSM [2], OpenFST [3] each have their own textual representations of automata. Such representations, often integer-based, are concise and simple to parse but they are dedicated to one program, and will hardly allow any generalization. What if the weights of our automaton are not integers, or if we want to label the transition of an automaton with rational expressions instead of letters? Other formats, such as GraphML [4], are more generic and allow

to represent any kind of graph, but they do not allow to represent the semantics associated to the automaton: indeed exchanging automata requires some typing information to be conveyed along with the structure.

Most of the design choices for our proposal for an exchange format have been shaped by the policy of making its implementation as simple as possible. This is already true of the option of choosing XML as the language for describing the format. Our proposal, called FSM XML, already covers a large class of automata but should be considered as a skeleton that can be completed to cater for other needs. We believe such a generic interchange format, should be of interest to the community.

FSM XML has been implemented and tested within Vaucanson [5], our automata manipulation platform, where it serves as an exchange format between components such as the core and the command-line interface. But this document is less a report on an experiment than a *position paper* intended to open the discussion.

Because of size constraints in these proceedings, we only give a brief summary of the FSM XML format and of the choices we have taken. We refer interested readers to our web page for more exhaustive information [6].

1. FSM XML Overview

We assume the reader familiar with the terminology of automata theory [7].

1.1. Data to Carry

The complete description of an automaton involves four different types of data: (1) the type of the labels, which amounts to define a semiring of series, a mathematical structure; (2) the automaton structure itself, that is a labeled graph; (3) if the automaton is to be seen on a screen or drawn in a figure, geometric data that tell where the states are located, and possibly the shape of the transitions, the relative location of its label; (4) finally, data which we call drawing data and that tell how the states and transitions are actually drawn, their size, the thickness the lines, the color, *etc.*

The two latter types are relevant only to applications that display the automaton in some way: they have no influence on the structural meaning of the automaton. Their presence is optional and we will focus on the first two items.

1.2. Automaton Description

An FSM XML description of an automaton consists in a tag <automaton/> containing two required children:

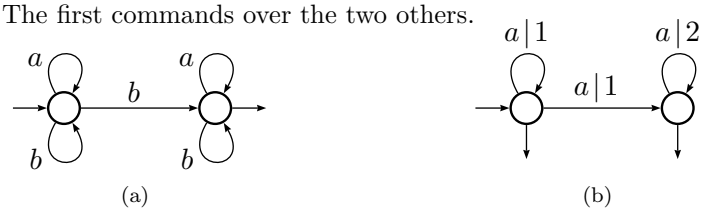
```
<automaton name='Example Automaton' readingDir=right>
   <valueType>...</valueType>
   <automStruct>...</automStruct>
</automaton>
```

The tag <valueType/> specifies the type of the labels of the automaton, and the tag <automStruct/> holds the description of the structure of the automaton (list of states and transitions with their labels).

The attribute name names the automaton and the attribute readingDir tells whether the automaton reads the word from left to right or from right to left.

1.3. The Labels: That Is the Question

The question of labels has three levels: (1) what are the types of labels that need to be supported? (2) how will these types be represented in the format? (3) how will the label of a given transition in an automaton will be represented in the format? The first commands over the two others.



(a) (b)

Figure 1. How many automata are there?

Let us consider Figure 1. At (a), we see an automaton whose labels are letters a and b and which clearly recognizes the set of words containing at least one b; but we can consider that the same automaton is an automaton with multiplicity in $\mathbb{N}$ where the coefficient of every transition is 1, in which case the automaton realizes the series which associates every word with its number of occurences of b. At (b), we see an automaton which is clearly a weighted automaton, and the weights are (positive) integers, but we do not know without further information which is the *semiring structure* which is applied to these integers: it may be the 'classical structure', in which case the weight of a^n is 2^n, or a 'tropical structure', and the weight of a^n is n if we are in $\langle \mathbb{N}, \mathsf{min}, + \rangle$, or $2n-1$ if we are in $\langle \mathbb{N}, \mathsf{max}, + \rangle$. This example makes clear that the description of an automaton *must* contain the definition of the *semiring of series* to which the behaviour of the automaton belongs. The requirement of such a strong typing is what distinguishes `FSM XML` from more general graph representation formats such as GraphML [4].

1.3.1. Label Types

We represent automata either over *free monoids* or over *products* of free monoids. Products of *arbitrary* number of free monoids allow to represent k-tape automata, that is, generalization of transducers that are 2-tape automata (we thus do not use the name 'transducer' in the description of the format).

The generators of the free monoids are either *simple* letters or *tuples* of simple letters of arbitrary dimension. The simple letters refers to simple types in programming languages such as *characters* (or subsets of them such as *letters* or *digits*) or even *integers*[1]. Tuples of letters consist then in ordered sets of simple letters, not necessarily all of the same type: for instance, pairs of letter and digit will naturally represent indexed letters. Another very useful example: automata over free monoids whose generators are pairs of letters are equivalent to transducers which realize length preserving relations and are also used, modulo some technicalities, to represent *synchronized transducers*.

Within `FSM XML`, we represent *weighted* automata; the weights being taken either in *numerical semirings* or in *series semirings*. By 'numerical semirings', we

[1] Automata used in the study of numeration systems, for instance, make use of labels that are integers.

refer to simple types of numbers, as they are implemented in any programming languages, like integers, or reals, together with conventional operations, or 'unconventional' ones that can be overloaded on the conventional ones. By 'series semirings', we refer to semirings that can be recursively defined by using the already defined numerical semirings and the (products of) free monoids. The reason for opening the possibility in FSM XML is Kleene-Schützenberger Theorem for transducers which states that a (finite) automaton over the product say $A^* \times B^*$ with multiplicity say in $\mathbb{N}$ is equivalent to an automaton over A^* with multiplicity in the semiring of (rational) series over B^* with multiplicity in $\mathbb{N}$.

1.3.2. Label Type Representation

The three main features of label types that we want to represent are then: product of an *arbitrary number* of free monoids, generators that are vectors of *arbitrary dimension*, and *recursive definition* for semiring of multiplicities. These are easily and naturally taken into account in an XML format although it may end rapidly into rather long and apparently complicated description files.

Figure 2 shows three excerpts of FSM XML files that describe label types: the first one for a classical Boolean automaton over a two-letter alphabet, and the two others for the aforementioned two equivalent forms of transducers with multiplicities.

One understands that in both tags `<semiring/>` and `<monoid/>`, we use the attribute `type` to control the *syntax* of the tag: we call such attributes *pivotal*. In `<semiring/>`, `type` = *`numerical`* calls for two other attributes, `set` and `operation` that will be given *token* values, that is, conventional strings that have to be recognized and correctly interpreted by the parser. Whereas `type` = *`series`* calls for another succession of `<semiring/>` and `<monoid/>` tags.

In `<monoid/>`, `type` = *`free`* calls for the attributes `genKind`, `genDescrip` and `genSort`, whereas `type` = *`product`* calls for the attribute `prodDim`, an integer stricly larger than 1 which tells how many children `<monoid/>` this tag `<monoid/>` will have.

The attribute `genKind` is another pivotal attribute which controls whether the generators are *`simple`* or *`tuple`*. In the former case, `genSort` is a token which tells which kind of generators are expected: *`letter`*, *`digit`*, *`alphanum`*, or *`integer`*. The latter case is not exemplified in Figure 2 but the mecanism is of the same type as for product of monoids, although it is not recursive.

The attribute `genDescrip` = *`enum`* tells that the generators of the free monoid will be enumerated, by means of the attribute `value` in tags `<monGen/>`. The assignments `genDescrip` = *`range`* and `genDescrip` = *`set`* are meant to give way to the description of large alphabets such as those that are used in NLP. The precise syntax and semantic open by these tokens have still to be defined.

This constrained specification of the type of an automaton makes it easier to extend the format to support new types without redefining the complete semantics of each new automaton type. For instance to support automata over a log-probability semiring we would just have to introduce some new tokens (and their semantics) for the `set` or `operation` attributes.

```
<valueType>
   <semiring type=numerical set='B' operation='classical' />
   <monoid type=free genKind=simple genDescrip='enum' genSort='letter'>
      <monGen value='a'/>
      <monGen value='b'/>
   </monoid>
</valueType>
```

(a) Type for a Boolean automaton over $\{a, b\}^*$.

```
<valueType>
    <semiring type=numerical set='N' operation='classical' />
    <monoid type=product prodDim='2'>
        <monoid type=free genKind=simple genDescrip='enum' genSort='letter'>
            <monGen value='a'/>
            <monGen value='b'/>
            </monoid>
        <monoid type=free genKind=simple genDescrip='enum' genSort='letter'>
            <monGen value='a'/>
            <monGen value='b'/>
        </monoid>
    </monoid>
</valueType>
```

(b) Type for an automaton over $\{a, b\}^* \times \{a, b\}^*$ with multiplicity in $\mathbb{N}$

```
<valueType>
    <semiring type=series>
        <semiring type=numerical set='N' operation='classical' />
        <monoid type=free genKind=simple genDescrip='enum' genSort='letter'>
            <monGen value='a'/>
            <monGen value='b'/>
        </monoid>
    </semiring>
    <monoid type=free genKind=simple genDescrip='enum' genSort='letter'>
        <monGen value='a'/>
        <monGen value='b'/>
    </monoid>
</valueType>
```

(c) Type for an automaton over $\{a, b\}^*$ with multiplicity in $\mathbb{N}\langle\langle A^* \times B^* \rangle\rangle$

Figure 2. FSM XML files for label types

1.3.3. Rational Expressions for Label Representation

It is quite a natural idea to be able to describe rational (that is, regular) expressions within an XML format for automata. The behaviour of a finite automaton over any monoid can be denoted by a rational expression, and most of the automata related software deal with the conversion between automata and expressions, back and forth.

For the large range of automata that we want to be able to describe, the need for rational expression is even more striking. For instance, as a consequence of the Kleene-Schützenberger Theorem we may have a transition labeled by a letter whose *weight* is a regular expression.

Rational expressions are well-formed formulas, that naturally correspond to trees and XML is perfectly fitted to describe trees. There is thus not much to say about the translation of an expression into an XML file. Two points have to be noted though.

The expressions we are interested in denote rational series, of course the same as the automata we are considering realize and, for the same reason, the expressions must begin with the description of the type of the semiring of series to which the series they denote belongs. The expressions thus share with automata

the tag `<valueType/>` (and this is the reason why we have called it *valueType* and not *labelType*).

As they correspond to weighted automata, our expressions are *weighted expressions* and as the weights may be taken in non commutative semirings, there exist *two* external multiplication operators: a *left* and a *right* one.

It is a major, as well as quite logical, feature of FSM XML that it possesses the possibility of describing rational expressions. As bare letters are also rational expressions, and with the idea of giving a uniform treatment to the largest class of entities, all labels (of transitions) are represented in FSM XML using the syntax for rational expressions. As an example, Figure 3 shows the description of a transition connecting two states $s0$ and $s1$, and labeled by $2(a, b)$.

```
<transition source="s0" target="s1">
  <label>
    <leftExtMul>
      <weight value="2"/>
      <monElmt>
        <monElmt><monGen value = "a"/></monElmt>
        <monElmt><monGen value = "b"/></monElmt>
      </monElmt>
    </leftExtMul>
  </label>
</transition>
```

Figure 3. A transition with input label '`a`', output label '`b`' and weight '`2`'

1.3.4. Geometry and Drawing Informations

The tags `<geometricData/>` and `<drawingData/>` are optional child tags of `<automaton/>`, `<state/>`, `<transition/>`, `<inital/>`, and `<final/>`. The former contains coordinates for the automaton and the states, and geometric shapes for transitions. The latter is planned to hold information about the way the automaton and its parts are drawn, but its content is not specified at this stage of our proposal.

2. Design Choices

2.1. Why XML?

While parsing XML is certainly not as efficient as loading a binary file, efficiency is not the first concern when devising a interchange format. The choice of XML simplifies exchanges, manipulations, and future evolutions (adding new tags, attributes, or tokens to support new automata do not invalidate existing files). Frameworks such as DOM [8] or SAX [9] make it easier to build a parser in many languages. An *XML Schema Description* (XSD) document [10] is available on our webpage [6] and many transformations can easily be applied to XML files using languages such as XSLT [11].

2.2. Apparent Verbosity

While XML documents remain human-legible (compared to a binary file at least) this interchange format is meant to be written by computers. We purposely tried to (1) unify the representation of the various automata and (2) refrained from adding any kind of syntactic sugar. In both cases, the intent is to simplify the number of cases an implementation of the format has to deal with.

For instance from the perspective on someone actually typing in an automaton in `FSM XML`, entering the transition as shown in Figure 3 is cumbersome and one could dream about some kind of syntactic sugar like:
`<transition source="s0" target="s1" in="a" out="b" weight="2"/>`.

Our point is that one never writes an XML file representing an automaton by hand (automata are either drawn using a graphical interface, or computed) and from a implementation perspective, the two forms are as easy to input or output. Since the syntax of Figure 3 makes it possible to represent more complex types than the above abbreviation, we have only kept the first: this frees the implementation from having to deal with many special cases. In other words, the verbosity is the result of a simpler grammar, chosen for the sake of simplicity.

2.3. Computable Properties Are Not Part of the Type

Among the 'structural' properties of automata, we can distinguish between properties that are *static*, or could be called a *type property*, such as the input alphabet, or the semiring of weights, and properties that we could call *computable* such as 'being deterministic', or 'unambiguous', or 'trim', or 'functional' (for a transducer).

As it stands, our proposition can specify static properties but makes no provision for the expression of computable properties. We do agree that such kind of attributes are useful (especially if the format is used for the communication between trusted components). The floor is open for the specification of tokens that will describe these computable properties.

The reason we left these properties aside is that we did not want to organize the format around them. For instance it sounds wrong to specify the type of an automaton by first telling whether it is deterministic or not: this kind of property definitively is not part of the type.

3. Conclusion

The experience gained using an XML format in `Vaucanson`, with the constraint of being able to define a large variety of automata, has shaped our choices for the proposal on the level both of design and implementation and there have been significant changes since the format we presented at the CIAA 2005 conference [6].

Even though the class of automata initially supported by `FSM XML` are those targeted by `Vaucanson`, this format is meant to be extended to encompass the needs of other tools from the community. We believe that the strong typing enforced by the format will give the many communities that use automata the necessary tools to ease such extensions.

References

[1] D. Raymond and D. Wood, Grail: Engineering Automata in C++, http://www.csd.uwo.ca/Research/grail/

[2] M. Mohri, F.C.N. Pereira and M.D. Riley, A Rational Design for a Weighted Finite-State Transducer Library, LNCS 1436, 1998, http://www.research.att.com/~fsmtools/fsm/

[3] C. Allauzen, *et al.*, OpenFst: A General and Efficient Weighted Finite-State Transducer Library, *Proc. of CIAA'07*, LNCS 4783, pp. 11–23, 2007, http://www.openfst.org

[4] U. Brandes, M. Eiglsperger and J. Lerne, GraphML - an XML based graph interchange format (2002), http://graphml.graphdrawing.org

[5] The Vaucanson Group, Vaucanson, a generic C++ platform for computing automata and transducers (2003–2008), http://vaucanson.lrde.epita.fr

[6] The Vaucanson Group, XML proposal for Automaton Exchanges (2004–2008), http://vaucanson.lrde.epita.fr/XML

[7] J.E. Hopcroft, R. Motwani and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 2000.

[8] W3C, Document Object Model 2 (2000), http://www.w3c.org/DOM

[9] D. Megginson, Simple API for XML 2 (2001), http://www.saxproject.org

[10] W3C, XML Schema Description (2001), http://www.w3c.org/XML/Schema/

[11] W3C, XSL Transformations (1999), http://www.w3.org/TR/xslt