FSMXML and its application in Vaucanson

Florian Lesaint

Technical Report n^o0818, May 2008
revision 1734

Last year, we started to work on a new proposal of an XML automata description format, now called FSMXML. This year we are presenting a final version of our work. It takes the form of an *rfc*. FSMXML mainly includes a full generalized rational expressions support, can describe any kind of automaton and has been made easier to support. We redesigned the VAUCANSON XML parser structure to get rid of a bad management of dependencies. It is updated according to the *rfc*.

Nous avions commencé l'année dernière à travailler sur une nouvelle proposition de format XML de description d'automates, devenu FSMXML. Nous présentons cette année une version aboutie de ce travail sous forme de *rfc*. FSMXML comprend notamment une gestion complète des expressions rationnelles généralisées, il permet de décrire n'importe quel type d'automate et sa gestion est facilitée. Nous avons repensé la structure du parseur XML de VAUCANSON pour nous affranchir d'une mauvaise gestion de dépendances et l'avons mise à jour conformément à la *rfc*.

Keywords
Vaucanson, FSMXML, XML Proposal, Format, Input, SAX, Xerces, XML, XSD

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
florian.lesaint@lrde.epita.fr – http://publis.lrde.epita.fr/200806-Seminar-Lesaint

Copying this document

Contents

Chapter 1

Introduction

Automata are used in lots of programs and projects. Libraries such as OpenFst (Riley et al., 2007) or VAUCANSON (VAUCANSON Group, 2001) are dedicated to them. The need of a description format for storing and exchanging data is natural.

Automaton description formats already exist, however they do not aim at being universal. All of them are dedicated to the needs of a specific program and part of them do not even have open specifications.

Designing an universal automaton exchange format aims at providing the community with a simple communication standard for a good interaction with programs that deal with automata and thus for any kind of them (Transducers, Büchi automata, ...).

FSMXML (VAUCANSON Group, 2004) defines such a description format.

Since FSMXML was written by the VAUCANSON Group, it is also used by the diverse tools of the VAUCANSON Project. The support of FSMXML led to an important update of the input and output system of VAUCANSON. Working on a parser dedicated to FSMXML while specifying it gave us the opportunity to improve the format to avoid as much as possible implementation problems. It also helped us to improve the bad input performances of the library.

This document summarizes the work realized in two years. The first part describes FSMXML and gives explanations of the choices made. The second one is on the implementation of the support of FSMXML within the VAUCANSON library.

Acknowledgements

Jacques Sakarovitch for his help and work on FSMXML. Alexandre Duret-Lutz and Maxime van Noppen for their comments on earlier drafts of this report. The whole VAUCANSON Group since 2002.

Chapter 2

Our proposal: FSMXML

2.1 Presentation

Origin

The VAUCANSON library provides an eXtensible Markup Language (XML) Input/Output system since 2004. The first XML formalism was designed as a proposal for the Conference for Implementation and Application of Automata (CIAA) 2004 (CIAA, 2004) to establish a standard for describing automata. Since then, the VAUCANSON Group has successively improved the format and submitted different papers on the subject.

FSMXML is the result of four years of thoughts and experiences on the subject. My work started two years ago with the rewriting of the proposal presented at CIAA 2005 (CIAA, 2005).

Scope

As said in the introduction, there are different kinds of automata: automata on finite words or on infinite words, automata on tuples of words (often called transducers), automata with multiplicity (usually called weighted automata), timed automata, counter automata, pushdown automata, Petri nets…

FSMXML is currently restricted to *weighted automata and transducers on finite words*, which are the current types supported by the VAUCANSON library. FSMXML also describes standalone rational expressions. FSMXML is in fact only the first step to an universal automaton exchange format.

2.2 Overview

This overview only gives a brief example on how an automaton such as the one in Figure 2.1 is described through an XML file such as the one in Figure 2.2. A more detailed description of the tags and attributes is done in the following sections.

People who are interested in the complete possibilities of FSMXML or more complex examples over weighted automata, transducers,… might want to look at the FSMXML dedicated web page (VAUCANSON Group, 2004).

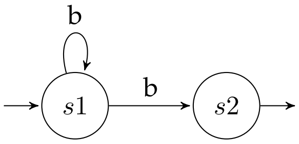


Figure 2.1: Visual representation of A_1, a Boolean automaton on alphabet $\{a, b\}$.

```
<fsmxml xmlns="http://vaucanson.lrde.epita.fr" version="1.0">
  <automaton name="A1"> <!-- description of the automaton -->
    <valueType> <!-- automaton type -->
      <semiring type="numerical" set="B" operations="classical"/>
        <!-- semiring where should be taken weigth values -->
      <monoid type="free"
              genKind="simple" genSort="letter" genDescrip="enum">
        <!-- automaton alphabet -->
        <monGen value="a"/>
        <monGen value="b"/>
      </monoid>
    </valueType>
    <automatonStruct> <!-- automaton structure -->
      <states>
        <state name="s0"/>
        <state name="s1"/>
      </states>
      <transitions>
        <transition source="s0" target="s0">
          <label>
            <monElmt>
              <monGen value="b"/>
            </monElmt>
          </label>
        </transition>
        <transition source="s0" target="s1">
          <label>
            <monElmt>
              <monGen value="b"/>
            </monElmt>
          </label>
        </transition>
        <initial state="s0"/> <!-- s0 is an initial state -->
        <final state="s1"/>
      </transitions>
    </automatonStruct>
  </automaton>
</fsmxml>
```

Figure 2.2: Overview of the FSMXML description of A_1.

A brief example on how standalone rational expressions are described in FSMXML can be seen in Figure 2.3

```
<fsmxml xmlns="http://vaucanson-project.org" version="1.0">
  <regExp name="c1-behaviour"> <!-- description of the expression -->
    <valueType> <!-- rational expression type -->
      <semiring type="numerical" set="N" operations="classical" />
      <monoid type="free"
              genKind="simple" genSort="digit" genDescrip="enum">
        <monGen value="0"/>
        <monGen value="1"/>
      </monoid>
    </valueType>
    <typedRegExp> <!-- rational expression -->
        <star>
          <sum>
            <monElmt>
              <monGen value="0"/>
            </monElmt>
            <one/>
          </sum>
        </star>
    </typedRegExp>
  </regExp>
</fsmxml>
```

Figure 2.3: $|\mathcal{C}_1| = (0 + \epsilon)^*$

2.3 Design principles

Our experience and work on several XML proposals led us to apply some rules in the design of FSMXML. Knowing them might help to understand the choices we made.

Rule 1: XML is not for Human

Even if XML is said to be Human-readable, the format is not intended to be directly used by Humans. Therefore we did not simplify it to make it easily readable but rather made it as easy as possible for a program to parse and use it.

The first consequence of this rule was to remove all the default values of the previous version of the XML proposal. Each default value within a format introduces extra checking for the input parser and an ambiguity within the output system.

Rule 2: Match and regroup concepts

Although FSMXML should not be read by Human, behind a program, there are always developers who will have to implement parsers. Therefore, the format is meant to be logical and to match as much as possible mathematical concepts and the automata theory.

This rule forbids us to have two different representations for a same concept. Allowing it would lead to some problems for the outputs of the format: if we give a file to a program that only converts the XML into an internal representation and then converts it again in XML (identity), the output should be identical.

Rule 3: Distinguish required from optional data

Automata are often described using graphs with states and transitions, and a lot of representation conventions. We tried to separate required information from optional ones.

Rule 4: A coding style

For each part of the format, the question is always the same:
"How should we represent it? With a tag or with an attribute ?"
Most of the time, both answers are correct; the rule followed here was to have the same behaviour for similar representation problems.

Rule 5: Attractiveness

The format is currently used by only one program… VAUCANSON itself. Our objective was to make it more attractive for the community.

The first consequence of this rule was to rename our proposal into FSMXML for Finite State Machine XML. It gives to the community, instead of *XML proposal for universal automaton description*, a name that can easily be identified.

2.4 Automata and rational expressions types

Weighted automata on finite words, transducers on finite words, and generalized rational expressions are represented over a monoid and might take a weight within a semiring. This information is kept within `<valueType>`, it describes in which object can be taken the *value* of the label of a transition.

A significant improvement of FSMXML was to take into account tuples or products of an arbitrary number of items.

Monoids

An automaton is either represented on a *free* monoid or over a *product* of free monoids. To be as generic as possible, we took *k-tape* automata into consideration, therefore a product can take an arbitrary number of free monoids (a transducer can be seen as a 2-tape automaton).

Generators of monoids, which are the symbols of the alphabet can either be simple symbols or tuples of simple symbols of arbitrary dimension. By "symbols", we refer to simple types in programming languages such as *characters*, *digits*, *integer*, or a mix of them.

To enable such a high level of possibilities, tags and attributes like `<monCompGen>`, `<genSort>`, `<genCompSort>`, `genKind`, `genSort` were introduced in the grammar. Examples of their use can be seen in Figure 2.4.

Semirings

A weighted automaton takes its weights within either a *numerical* or a *series* semiring. By "numerical semiring", we refer to simple types of numbers in programming languages such as *integers*, *reals*, ... By “series semiring”, we refer to semirings that can be recursively defined using a numerical semiring and a (product of) free monoid(s).

Kleene-Schützenberger Theorem states that a (finite) automaton over the product $A^* \times B^*$ 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 $\mathbb{N}$. Following our VAUCANSON’s rational expression’s writing conventions, a transition “$\{3\}\,(a, b)$” in a free monoid product transducer would be “$\{\{3\}\,x\}\,b$” in a transducer with rational weights (rw_transducer).

```
<valueType>
  <semiring type="numerical" set="N" operations="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" operations="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="digit">
      <monGen value="0"/>
      <monGen value="1"/>
    </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" operations="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 2.4: Examples of *value* types.

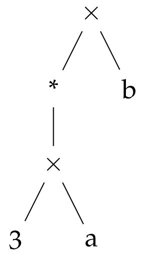
2.5 Generalized rational expressions

By “generalized” we refer to rational expressions that can takes weights in a semiring. In the sequel, rational expressions will be used for “generalized” rational expressions.

The support of standalone rational expressions is natural as the behaviour of a finite automaton *over any monoid* can be denoted by a rational expression and vice versa. Moreover tools like TAF-KIT, which are part of the VAUCANSON project, provide commands to convert automata to rational expressions and vice versa, using FSMXML as their input/output system.

The major part of the work on rational expressions was done by Florent Terrones, followed by Jacques Sakarovitch. The work left was on the best way to represent the leaves of rational expressions.

Rational expressions are commonly represented as strings however they are always implemented as trees (see Figure 2.5). Since XML has a tree structure and given *Rule 1*, we took advantage of it and avoided using strings. We made the choice that the tree should be binary (which matches the implementation of the VAUCANSON library).



```
<typedRegExp>
  <product>
    <star>
      <leftExtMul>
        <weight value="3"/>
        <monElmt>
          <monGen value="a"/>
        </monElmt>
      </leftExtMul>
    </star>
    <monElmt>
      <monGen value="b"/>
    </monEltmt>
  </product>
</typedRegExp>
```

Figure 2.5: Tree representation of $(\{3\}\, a)^* \times b$ and its FSMXML equivalent

A standalone rational expression is described within `<regExp>` which takes a `<valueType>` describing the alphabet and the semiring the expression is based on. It takes a `<typedRegExp>` which describes the tree structure of the expression. Refer to Figure 2.3 for a complete example.

Weights

`<weight>` describes a weight value taken in the semiring defined in `<valueType>`.

Depending of the type of semiring *numerical* or *series*, `<weight>` takes either a simple attribute `value` when dealing with simple symbols or a complete rational expression.

Internal nodes: operations

sum, product

`<sum>` and `<product>` stands respectively for the sum and the product of two rational expressions.

star

<star> starifies the rational expression.

leftExtMul, rightExtMul

Since our tree is binary and weights cannot be attributes (since they can be rational expressions), we dedicated two operations <leftExtMul> and <rightExtMul> to multiply a rational expressions with a weight respectively on the left and on the right.

Moreover allowing optional <weight>s in each internal nodes and leafs was rejected given the *Rules 1 and 2*. An optional child requires extra checking from any tool.

Leaves: values

zero

<zero> is the only way to describe the *null series*. Although the tag exists in the grammar, it should not be used. Since the output of FSMXML will not display a transition if its label is the *null series* and sums or product with zero are always simplified.

one

<one> is the only way to describe the *identity series*.

monElmt

<monElmt> describes a *monoid element*, equivalent of a *word*.

For a *free monoid*, <monElmt> is a concatenation of generators of the monoid: in the FSMXML formalism, a list of <monGen> as described in the monoid.

For a *k-product monoid*, <monElmt> is a concatenation of *k* monoid elements taken in the free monoids of the product.

However, it should be possible to the value taken in one of the free monoids to be the identity, therefore <one> is also able to describe the monoid identity of a free monoid.

No other tag like <monId> is provided, since it would have introduced two possibilities to describe an equivalent concept, and would go against the *Rule 2*.

```
<monElmt>
  <monId/>            <one/>
</monElmt>
```

Equivalence when <monId> is seen as the *identity* generator, within a *free* monoid.

```
<star>                <star>
  <monId/>              <one/>
</star>               </star>
```

Equivalence when <monId> is seen as the *identity* monoid element of a *free* monoid.

2.6 Automata

Automata are represented as graphs with states and transitions labelled with rational expressions.

An automaton is described in FSMXML within <automaton> which takes a <valueType> describing on which alphabet and with which weights are based the rational expressions labelling the transitions. It also takes a <automatonStruct> which holds the automaton representation. Refer to Figure 2.2 for a complete example.

States

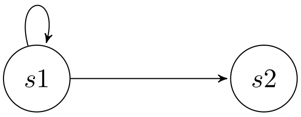
<states> lists all the states of an automaton.

```
...
<automatonStruct>
  <states>
    <state id="1" name="s1"/>
    <state id="2" name="s2"/>
  </states>
```

Transitions

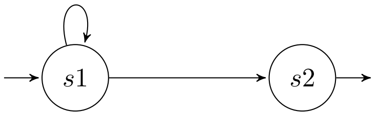
<transitions> lists all the transitions of an automaton...

```
  <transitions>
    <transition source="1" target="1">
      <label>
       ...
      </label>
    </transition>
    <transition source="1" target="2">
      <label>
       ...
      </label>
    </transition>
    ...
```



...and also the initial and final properties of the states, since states with initial and final properties can have a label, they are often seen as transitions with only one source or destination.

```
    ...
    </transition>
    <intial state="1"/>
    <final state="1"/>
  </transtions>
</automatonStruct>
...
```



Labels on transitions, final states, and initial states are rational expressions with the same behaviour as the one describes within <typedRegExp> tags.

2.7 Extra information

Drawing and geometry data

Dealing with automata is often done through visual GUI, representing them as graphs. Therefore, there is a need of storing visual representation data on the automata.

Drawing and geometry data are stored in <drawingData> and <geometricData>, they are not necessary to the description of an automaton, therefore these tags are optional. An exhaustive list of the possible children and attributes can be found on our dedicated web page.

Even if reducing size of the XML files was not our priority, avoiding duplicated data when possible is always better, even more when this duplicated information might take a significant amount of space. A proposal done last year, but yet to be added to FSMXML, are the *Drawing classes*.

Providing a `<drawingClass>`, that can regroup similar visual data within one place and a `drawing` attribute to elements such as `<state>` or `<transition>` to bind them with a class, reduces the size of the file and avoids redundancy. An example of its use can be found in Figure 2.6.

An extreme example would be to describe an automaton with half of its states red and the rest green. Given a certain amount of states, the size of the two equivalent files can be reduced by half.

```
<automaton name="a1">
  <valueType>
    ...
  </valueType>
  <automatonStruct>
    <states>
      <state id="0">
        <drawing stateFillColor="blue">
      </state>
      <state id="1">
        <drawing stateFillColor="red">
      </state>
      ...
    </states>
    ...
  </automatonStruct>
</automaton>
```

default version.

```
<automaton name="a1">
  <drawing>
    <stateclass name="default"
      fillColor="red"/>
    <stateclass name="blue"
      fillColor="blue"/>
  </drawing>
  <valueType>
    ...
  </valueType>
  <automatonStruct>
    <states>
      <state id="s0" drawing="blue/>
      <state id="s1" drawing="red"/>
      ...
    </states>
    ...
  </automatonStruct>
</automaton>
```

with drawing classes.

Figure 2.6: Drawing class presentation.

If *Drawing classes* are not par of FSMXML yet, it is because we are still discussing about the problems that could be linked with the scattering of information.

`readingDir` and `genDescrip`

`readingDir` is an attribute of `<automaton>` that has been introduced to underline the proposal aspect of FSMXML and its possible extension with new types. However nothing has been defined on how should be described an automaton when `readingDir` is set to `right`.

`genDescrip` is an attribute of `<monoid>` which should enable the extension of format with alternatives to numbering all the generators of a monoid. FSMXML only specifies how to describe a monoid when `genDescrip` is set to `enum`.

```
<monoid type="free" genKind="simple" genDescrip="range" genSort="letter">
  <monGen value="a-z"/>
  <monGen value="A-Z"/>
</monoid>
```

Figure 2.7: A plausible extension of FSMXML with `genDescrip` as `range`.

Representation data

In the previous version of the format, `<monoid>` and `<semiring>` were carrying attributes such as `identitySymbol` and `zeroSymbol`. Their values could be used to write rational expressions such as $(a + \varepsilon)$ within the `label` attribute of `<transition>` (this attribute is not allowed any longer).

As you might have noticed, with `<one>` and `<zero>`, these symbols become useless. Moreover, they can lead to two different representations of a same concept, if these symbols can be used in the `value` attribute of a `<monGen>`.

However, these symbols are still useful when we want to convert a FSMXML rational expression within a *string* rational expression into a program. But they are representation data, and definitely optional.

We created an optional `<writingData>` child of `<semiring>` and `<monoid>` that stores this information. An example can be seen in Figure 2.8.

```
<semiring type="numerical" set="N" operations="classical"/>
<monoid type="free" genKind="simple" genDescrip="enum" genSort="letter">
  <writingData identitySymbol="1"/>
  <monGen value="a"/>
</monoid>
...
    <sum>
      <monElmt>
        <monGen value="a"/>
      </monElmt>
      <one/>
    </sum>
```

Figure 2.8: How to represent "(a+1)" where 1 stands for the identity of the monoid.

2.8 Related works

FSMXML *request for comments*

The last year, we spent some time trying to write a paper on our format before realizing we had a problem with the grammar and the definition of the objects described such as monoids and semirings. Therefore we finally ended starting to write a *request for comments* (*rfc*) (Lesaint, 2008).

The *rfc* describes in a very formal way each tag of FSMXML, its attributes and children with the concepts they are representing. It also provides a set of complete examples of FSMXML descriptions.

XSD **file**

An XML Schema Definition (XSD) file has been written to match as much as possible the FSMXML format, and is available on the Vaucanson Group homepage (Vaucanson Group, 2001).

It is important to notice that neither XSD files nor Document Type Definition (DTD) files, the main way to describe an XML formalism, are able to describe the complex dependencies and constraints introduced in FSMXML. Therefore one should not base a parser only on the XSD file provided, but rather on the *rfc*.

A first example is that there is no way (without extra tools such as XPath) to ensure that the values used within `<monGen>` tags in a `<monElmt>` are previously defined in a `<monGen>` within the `<monoid>`.

A second example is that there is no way (without extra tools) to change the behaviour of a tag like `<monoid>` depending on the value given in the `type` attribute. An example is given in Figure 2.9

```
<monoid type="free" genKind="simple" genSort="letter">
  <monoid type="free" genKind="simple" genSort="letter">
    <monGen value="a"/>
  </monoid>
  <monoid type="free" genKind="simple" genSort="digit">
    <monGen value="1"/>
  </monoid>
</monoid>
```

A `<monoid>` with `type=free` cannot have `<monoid>`s tags as children, however the XSD will accept this XML description.

Figure 2.9: XSD limits on FSMXML constraints

2.9 Further work

The work done so far this year has been to finalize the *rfc* and the XSD file, update the description of the `<valueType>` by adding arbitrary dimensions, the specifications of the `<monElmt>` leaf, and to fix the nomenclature of the tags and attributes.

The next step of this work will be to introduce FSMXML to the community. We have already started by submitting a paper to Finite-State Methods and Natural Language Processing (FSMNLP) (FSMNLP, 2008). We might try to provide other extensions or tools that would allow to use FSMXML within tools of alternative projects such as OpenFst.

We also expect a lot of comments and extensions from the community. It might be soon possible to describe new types of automata within the format since the LRDE now supports Spot (MoVe team, 2004), an object-oriented model checking library which works with Büchi automata.

Chapter 3

Implementation in VAUCANSON

As said in the introduction, while working on FSMXML, we redesigned the XML input/output system of VAUCANSON. The first year of our work was dedicated to the conversion of our previous Document Object Model (DOM) implementation of the input parser into a Simple API for XML (SAX) one. The second year was spent on the optimisation of both input and output systems.

3.1 Vaucanson I/O system

Before all, here is a brief recall on how works the I/O system of VAUCANSON. Inputs and outputs are done via a unique interface,

```
template<typename Auto, typename TransConverter, typename Format>
tools::automaton_saver_<Auto, TransConverter, Format>
automaton_saver(const Auto& a,
                const TransConverter& e = TransConverter(),
                const Format& f = Format());

template<typename Auto, typename TransConverter, typename Format>
tools::automaton_loader_<Auto, TransConverter, Format>
automaton_loader(Auto& a,
                 const TransConverter& e = TransConverter(),
                 const Format& f = Format(),
                 bool merge_states = false);
```

where *a* is the automaton we want to load or save, *e* is a tool that formats a transition as wished and *format* is the format in which we want to read/load or write/save. VAUCANSON currently supports XML, FSM and dot formats.

The following code expects to read on the standard input source an FSMXML Boolean automaton description:

```
vcsn::boolean_automaton::alphabet_t alpha;
vcsn::boolean_automaton::automaton_t aut = make_automaton(alpha);
std::cin >> automaton_loader(aut, string_out(), xml::XML());
```

For each format, a simple inclusion of the associated file (*XML.hh, dot_format.hh, simple_format.hh,* . . .) activates the use of it.

3.2 From DOM to SAX

Various mechanisms are possible to parse XML files, the main ones are DOM and SAX, described as below:

DOM

The principle of DOM (W3C, 2000a) is to parse the document in a single pass and to build an associated tree, which can then be traversed as many times as needed, bottom-up or top-down.

The interest of a DOM implementation is the easiness with which we can retrieve information in the associated tree. It also comes with an important documentation and since the mechanism is really mature, lots of good tools are available to deal with it.

DOM has of course some drawbacks, which in our case were quite problematic. Using an associated tree wastes a huge space of memory as soon as the automaton parsed is big. VAUCANSON was known to swap on some automata (determinized ladybirds for example).

SAX

Unlike DOM, SAX (Megginson, 2001) describes how to parse step by step the document and build at the same time our own objects. This also means that there is no way to browse the XML data once more without parsing it again from the beginning.

The interest of a SAX implementation is the lack of associated tree in the process, therefore no space is wasted with an intermediate structure. It made us change our previous DOM implementation to a SAX one for the inputs to significantly reduce the opportunities of swapping of the VAUCANSON library. A system built on a SAX implementation is also easier to maintain compared to a DOM one.

DOM is still used for the output of the VAUCANSON library, since there is no real alternative mechanism to DOM for writing XML. A solution would have been to write our own system…

3.3 SAX Parser implementation

The VAUCANSON project is using the Xerces library (The XML Apache Project, 1999), reasons of this choice are avaiable in my previous report (Lesaint, 2007b).

3.3.1 Terminology

For a better understanding of this section, some terms should be defined.

Entering tags

This term describes a node which is represented in the XML file like this: `<tagName ...>`. It contains the attributes of the tag.

Closing tags

This term describes the end of a node which is represented in the file like this: `</tagName>`. Sometimes, when the tag content is empty, you will find a unique tag like `<tagName .../>`. It stands for `<tagName ...>` directly followed by `</tagName>`.

Tag body

It stands for the content located between an entering tag and a closing tag. In FSMXML, a tag body is always a list of (zero or more) tags.

```
<localname attributes> <!-- entrance tag -->
  <child1> <!-- tag body -->
  <child2> <!-- tag body -->
  ... <!-- tag body -->
</localname> <!-- closing tag -->
```

XMLString

In Xerces, strings have their specific type: *XMLCh**, which are not C++ strings.

3.3.2 Implementation

SAX works with a principle of callback functions: The kernel reads the file step by step and each time an event occurs, it calls the associated function of our handler. There are two main events:

- An entering tag is found. The kernel calls the *startElement* function of the current handler.

  ```
  void
  startElement (const XMLCh* const uri,
        const XMLCh* const localname,
        const XMLCh* const qname,
        const xercesc::Attributes& attrs);
  ```

- A closing tag is found, the kernel calls the *endElement* function of the current handler.

  ```
  void
  endElement (const XMLCh* const uri,
      const XMLCh* const localname,
      const XMLCh* const qname);
  ```

If we specify “current” handler, it is because handlers can be changed on the fly, a capacity on which is based of our system. Indeed, the easier way would be to have for each tag a specific handler.

A problem of SAX standard implementation (like provided by Xerces) is that there is no possibility of lookahead. There is no way to predict which entering tag will be next. Moreover, since our grammar allows optional tags, we cannot even "assume" which tag will be next. A specific handler for one tag is therefore impossible.

The solution chosen was to split the code dealing with each tag in 2 parts:

- A function, which is called within the current handler, after reading the entering tag and which initializes the good handler.
- A handler, which has to process the tag body, that is to say, compute the attributes and call functions to initialize other handlers. It is also the work of our handler to “restore” the previous one after reading a closing tag.

Figure 3.1 shows a brief example of the mechanism of our implementation. A more detailed explanation can be found in my previous report (Lesaint, 2007b).

init: *StatesHandler* is set.

1 `<state id="1">` is read ⇒ *startElement* is called.

2 (in startElement) *state* is read ⇒ *statefunction* is called.

3 (in statefunction) a state (with id to “1”) is created in the automaton. *StateHandler* is set.

4 `</state>` is read ⇒ *endElement* is called.

5 (in endElement) *state* is read (= end) ⇒ *StatesHandler* is restored.

Figure 3.1: Parsing process example

3.4 Optimisations

3.4.1 Space consumption

A direct result of the SAX implementation was to reduce the space used while loading the automaton. Figure 3.2 shows how much memory we spare with this conversion. This is due to the lack of intermediate tree. Information linked to the structure itself is negligible in comparison to the information directly connected to the automaton therefore the DOM structure should be almost the same size of the internal representation of the automaton, thus it is logical to see that the memory usage has been reduced by half with a SAX implementation.

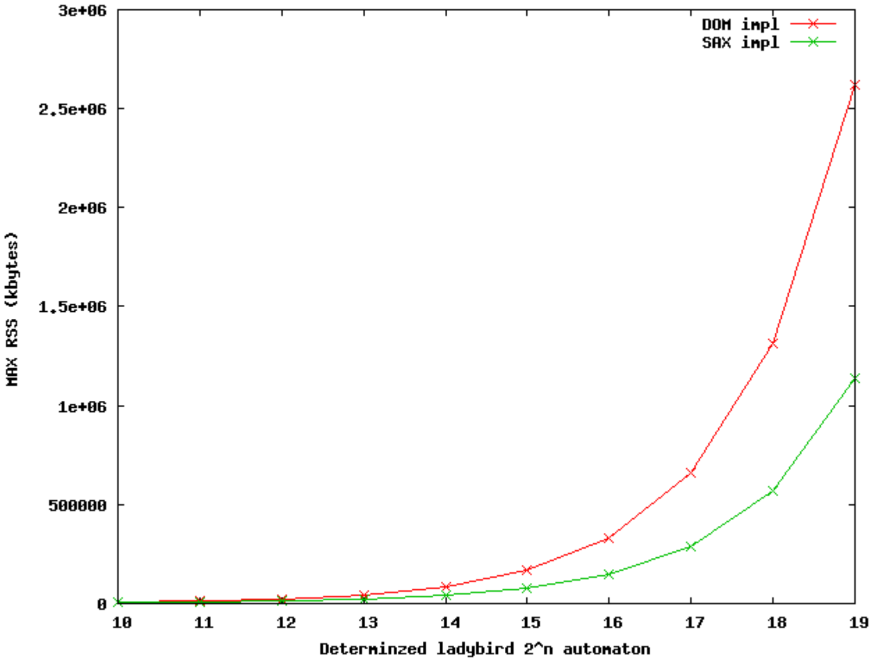


Figure 3.2: Max memory size on both SAX/DOM implementation.

While benchmarking we had a very bad surprise with our DOM implementation. It was leaking a lot, as seen in Figure 3.3. Therefore we do not really know if swapping, which was the problem that lead us to a SAX implementation, was because of DOM itself or our bad implementation.

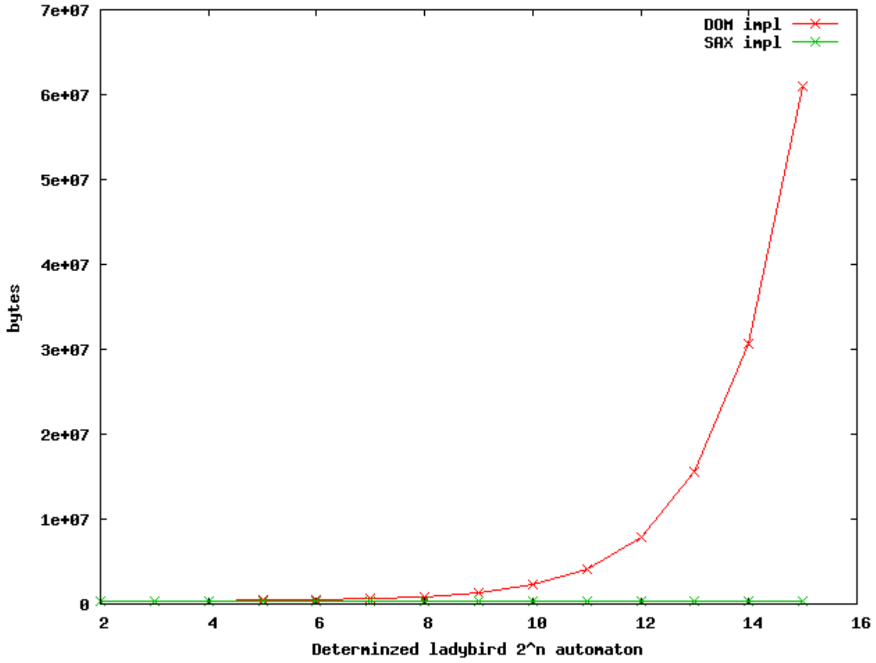


Figure 3.3: Memoryleak on both SAX/DOM implementation.

A determinized ladybird is an automaton with a lot of transitions and states. A ladybird with 6 states when determinized reaches 2^6 states. We based our benchmarks on determinized ladybirds since they are easy to generate and we focused on the size of the automaton.

3.4.2 Time consumption

Our first thought was that on huge automata, the time consumption should be partially reduced since the time required to build an intermediate tree is avoided. Our benchmarks however, dot not confirm our hypothesis (see Figure 3.4).

More advanced benchmarks were done on the SAX implementation, which made us notice that an important part of the time spent in our parser was dedicated to the conversion of the data in XMLStrings into C++ strings.

We did not find a way to improve that conversion so we tried to avoid any conversion that was not required. There are some parts in the implementation where we only check if an XMLString value is equal to a C++ one by converting the XMLString value. We reversed the process, since the conversion of a C++ string into an XMLString is cheaper.

An *XMLEq* class now stores a list of tokens in XMLString that we might need for comparison. The list of tokens is in fact the list of tag names of the FSMXML format. Now “transition” is converted into XMLString only once, instead of each time a transition was parsed.

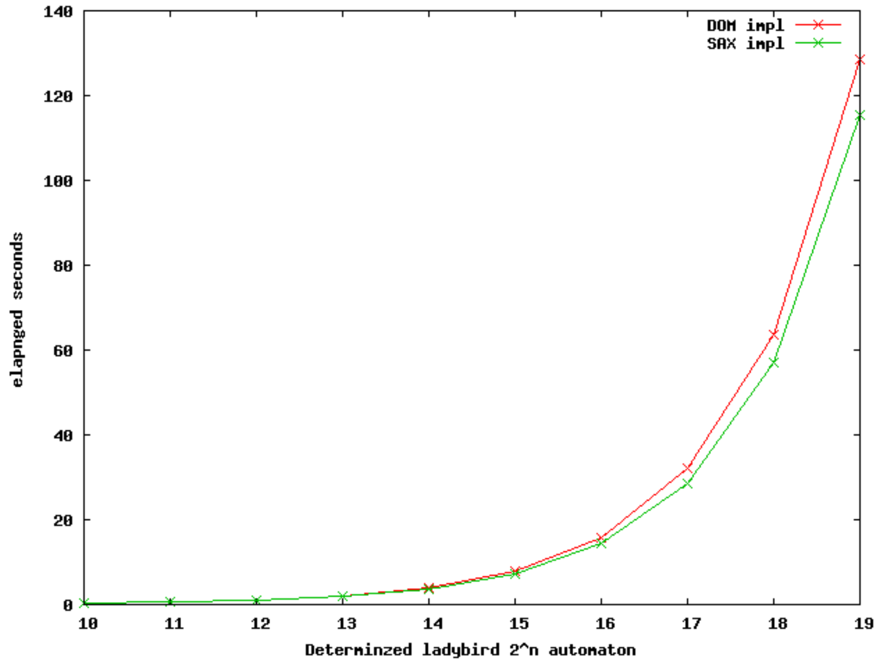


Figure 3.4: Time spent loading an automaton with both SAX/DOM implementation.

No new benchmarks were done this year since we did not focus on time consumption but on useless dependencies, described in the following section.

3.4.3 Useless dependencies

Michaël Cadilhac reported a problem of useless dependencies two years ago (ticket #33). For simplicity reasons the previous versions of the system (including the first SAX implementation) were including all existing contexts. Contexts are all the specialized implementations we have on diverse types such as *tropical semirings* or *free monoids products*.

The result was that a basic program dealing only with *Boolean automata* would include all the necessary tools to deal with *fmp transducers*, which is against logic.

This problem is even more important as we recently introduced shared libraries (*libvcsn-b, libvcsn-z, libvcsn-tdc*) to reduce the compilation time of the project's tools. This becomes useless as soon as we use the old XML support including all the possible contexts in the VAUCANSON library.

Some methods were explored in order to resolve this problem, each of them having its own advantages and drawbacks. The following subsections describe our thoughts on the most plausible ones.

Factories

The Factory design pattern (Alexandrescu, 2001) is generally used when objects need to be dynamically built in a program. Given some dynamical information such as a string "boolean", commonly called *identifier*, the right constructor for creating the corresponding object such as a Boolean automaton should be called.

If you look at the *rfc*, you can see that some attributes are listed as *Pivot*. Those are the exact values that should be required to implement such factories. In our case, different factories are required for monoid generators, monoids, semirings, automata (given a monoid and a semiring), and labels or rational expressions (given the automaton).

```
     semiring_t*
     AutFactory::create(const std::string& set,
const std::string& operations);
```

Figure 3.5: Example of a factory for semirings.

A implementation using this design pattern should perfectly work, if only we were not using the Element design pattern (Régis-Gianas and Poss, 2003). The Element design pattern radically changes the inheritance our main objects such as semirings are based on.

Therefore our *create* function in any possible Factory would only be able to return an Element<S,T> object, which we would have to cast within an expected object. An even more complicated problem, the semiring_t from boolean_automaton do not have the same interface of a semiring_t from rw_transducer. The code has therefore to be specialized each time we use a semiring.

But the main problem is that the Factory design pattern was not designed to return a templated object. Some work has already been done on the problem (Peleg, 2007), but the solution is still in discussion and was quite hard to implement and to test within the limited time we had.

Preprocessing conditions

A second option was to use a new set of MACROS constants to define which contexts are used in the program and to activate or deactivate part of the code within the handlers files. For example, the file *tropical_semiring.hh* would contain a TROPICAL_SEMIRING constant, and the *handlers.hh* file should have a macro #ifdef condition based on that constant.

The problem is that it requires the user to include files within a specific order in its program. This constraint cannot be checked but only specified within the documentation.

A user would hardly understand why after including *XML.hh* and *boolean_automaton.hh*, the program fails telling him there is no XML support. We really do not want that to happen, since it is one of the easiest way to lose a user.

It would also have implied to define MACRO constants without undefining them, which is a bad way of programming.

Plug and play files

Finally, the chosen way is to request the user to include not only *XML.hh* but also files depending of the automata he wants to use, for example *max_plus.hh*.

This solution is not very good, but is the better we found to solve this problem without using Factories. It requires the user to know that he needs to include more than the *XML.hh* if he wants to deal with complex automata, which can be done using an explicit message when a type is not supported.

```
#include <vaucanson/xml/contexts/max_plus.hh>
#include <vaucanson/xml/XML.hh>
```

Figure 3.6: Code required to load a Boolean automaton on a maxPlus semiring.

A good point compared to preprocessing conditions, is that the code is well separated in different files and directories, which makes it easier to maintain and understand. Each time a new context is added to Vaucanson, a new file (following the same pattern as the already present ones) should be written.

3.5 Annex problems

While updating the I/O system, several problems appeared in the conception designs used within the Vaucanson library, aspects that should be discussed in a near future.

3.5.1 Rational expressions design

The way rational expressions are designed is problematic as soon as we want to write them back in XML. Our representation of rational expressions includes three leaves: one, zero, and atom standing respectively for the series identity, the null series, and a word.

The problem is that contrary to the XML representation where a word is a list of generators, here, the word is a single value (C++ string or integer, or pair of strings) and the Vaucanson monoid interface does not yet provide a way to extract the associated generators from that word. In a letter based monoid, it should be fine, but what about more complicated once based on integers ?

3.5.2 Algebra interface

Since the current conception of the XML parser requires the knowledge of the type of automaton before loading it, parsing `<valueType>` is in fact only a checking step, which underlines the following problem.

The semirings and monoids objects in the Vaucanson library do not carry a string identifier like "MaxPlus" for a MaxPlus semiring. Therefore, the I/O system has to "list" all the possible objects that might be associated with identifiers. By listing, we mean to find a way to distinguish 2 contexts. Figure 3.7, which is the current way to retrieve information about a semiring, underlines how easier it would be if semiring had a function to retrieve such an identifier.

Even more, the current implementation with the design pattern Element does not help to understand on which object we need to specialize a function to retrieve the required information. Given a semiring *s*, its set can be found using the *s.value()* object and its operations using the *s.structure()* object.

3.5.3 Transition design

When parsing a transition, we first get the source and the target in the `<transition>` tag and we then get the series of that transition in the `<label>` tag.

```
template <typename T>
const char* get_semiring_operations(const T&)
{
  return "undefined operation";
}
...
template <>
const char* get_semiring_operations<
    vcsn::algebra::TropicalSemiring<vcsn::algebra::TropicalMax>
  >(const vcsn::algebra::TropicalSemiring<vcsn::algebra::TropicalMax>&)
{
  return "maxPlus";
}
```

Figure 3.7: How to retrieve the identifiers of operations given a semiring

Our first approach was to create a transition from the source to the target with a null series and then to give the label handler a reference to that series. However, there is no easy way to update a series within a transition.

To update a transition, we currently have to follow this procedure:

1. retrieve source *i* and target *o* of the transition *t*
2. create a new series *s* with the wanted value
3. destroy the transition *t*
4. create a new transition from *i* to *o* with a label *s*.

In our case, the problem is not that important, it only obliges us to store information about the states until the series is build before creating a transition. However, in lots of algorithms like ε-removal, it might significantly reduce and simplify some steps.

3.6 Further work

This year, we finalized the previous incomplete implementation of the input system. We also significantly improved the library especially by reducing these useless dependencies. It also helped us to underline problems with the conception of the VAUCANSON library, about which work should be done.

Some work still remains to be done in the current implementation, such as to factorize rational expressions operations handlers, but it should not be source of problems and is only time consuming.

As further improvements, we should take into considerations the use of Factories and avoid the need of knowing the type of automaton we are expecting to load. It can be a real problem for tools like a GUI that does not know which automaton it will receive.

Chapter 4

Conclusion

This work is not the only one dealing with the I/O system of VAUCANSON, the whole interface of VAUCANSON has been improved this year. Vivien Delmon has been working on the rational expression parser (Delmon, 2008). Florent D'Halluin has been working on a new Graphic interface (D'Halluin, 2008). The objective is definitely to make VAUCANSON more attractive to new users. We are still waiting an answer from the FSMNLP committee about our last paper. A positive answer would be great to remind the VAUCANSON project to the community.

There is still work to be done to optimize the code itself, improve the user documentation on how input/output works in VAUCANSON. We also should spend some time on the different further improvements previously listed in the paper, such as an FSMXML support extension for OpenFst. We should especially take some time to determine if the annex problems listed are slowing down the development of VAUCANSON, and act accordingly.

Appendix A

Bibliography

Alexandrescu, A. (2001). *Mondern C++ Design: Generic Programming and Desgin Patterns Applied*. Addison Wesley.

Brandes, U., Eiglsperger, M., and Lerne, J. (2002). GRAPHML – an XML based graph interchange format. http://graphml.graphdrawing.org/.

CIAA (2004). Conference for implementation and application of automata. http://www.informatik.uni-trier.de/~ley/db/conf/wia/ciaa2004.html.

CIAA (2005). Conference for implementation and application of automata. http://www.i3s.unice.fr/ciaa05/.

Delmon, V. (2008). Rational Expression Parser.

D'Halluin, F. (2008). Yet Another Vaucanson GUI.

FSMNLP (2008). Finite-State Methods and Natural Language Processing. http://langtech.jrc.it/FSMNLP2008/.

Galtier, J. (2008). Improving Vaucanson's transducers composition algorithm.

Gansner, E., Koutsofios, E., and North, S. (2004). Dot and Dotty. http://www.graphviz.org/.

Herman, I. and Marshall, M. (2000). GRAPHXML – equivalent to GRAPHML. http://www.cwi.nl/ftp/CWIreports/INS/INS-R0009.pdf.

Lesaint, F. (2007a). Draft of the XML Proposal. https://svn.lrde.epita.fr/svn/lrde-publis/trunk/lesaint.07.fsmnlp/.

Lesaint, F. (2007b). XML proposal and its application in vaucanson.

Lesaint, F. (2008). FSMXML rfc. http://www.lrde.epita.fr/dload/vaucanson/techrep/fsmxml-0.5.rfc.pdf.

Lombardy, S. and Sakarovitch, J. (2005). How expressions can code for automata.

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

MoVe team (2004). Spot. http://www.lrde.epita.fr/Spot/.

Peleg, G. (2007). Subscribing template classes with object factories in c++. http://www.artima.com/cppsource/subscription_problem.html.

Régis-Gianas, Y. and Poss, R. (2003). On orthogonal specialization in c++. http://www.lrde.epita.fr/dload/papers/poosc03-vaucanson.pdf.

Riley, M., Schalkwyk, J., Skut, W., Allauzen, C., and Mohri, M. (2007). OpenFst library. http://www.openfst.org.

Sakarovitch, J. (2003). *Éléments de théorie des automates*. Vuibert informatique.

VAUCANSON Group (2001). VAUCANSON home page. http://vaucanson.lrde.epita.fr/.

VAUCANSON Group (2004). FSMXML. http://vaucanson.lrde.epita.fr/Vaucanson/XML.

W3C (1998). eXtensible Markup Language – XML. http://www.w3c.org/XML/.

W3C (1999). XML Path Language. http://www.w3c.org/TR/xpath.

W3C (2000a). Document Object Model 2. http://www.w3c.org/DOM/.

W3C (2000b). XML document type definition. http://www.w3c.org/TR/REC-xml/#dt-doctype.

W3C (2001). XML schema description. http://www.w3c.org/XML/Schema.

W3C (2007). XSLT Transformation 2.0. http://www.w3c.org/TR/xslt.

The XML Apache Project (1999). Xalan and Xercesc libraries. http://xalan.apache.org.