FSM XML

The Vaucanson Group

19 May 2008

FSM XML is an XML format proposal for the description of weighted automata, transducers, and regular expressions. This document gives a pseudo-formal description of the format.

All tags of the format are listed, with their children, and attributes.

1 The root tag

0. `<fsmxml/>`

   The unique possible root of an FSM XML file, which can contain any number of automata and standalone rational expressions.

```
<fsmxml xmlns="" version="">
   <automaton/>      0 or more occ.
   <regExp/>         0 or more occ.
</fsmxml>
```

2 The value type tags

Both `<automaton/>` and `<regExp/>` have a child in common: the `<valueType/>` tag which describes the 'type' of the behaviour of the automaton or of the series denoted by the expression.

1. `<valueType/>`

```
<valueType>
   <semiring/>      req.
   <monoid/>        req.
</valueType>
```

2. `<semiring/>`

   • Pivotal att.: `type` = *numerical*|*series* token, req.

   2.1. `type` = *numerical*

```
<semiring type=numerical set='' operation='' >
   <writingData/>      opt.
</semiring>
```

   • Att: `set` = *B*|*N*|*Z*|*Q*|*R*|*C* token, req.
   `operation` = *classical*| *minPlus*| *maxPlus* token, req.

- Tag `<writingData identitySymbol='' zeroSymbol=''/>`
  - Att.: `identitySymbol` = ' ' string, req.
    `zeroSymbol` = ' ' string, req.

2.2. `type` = *`series`*

```
<semiring type=series>
   <writingData/>      opt.
   <semiring/>         req.
   <monoid/>           req.
</semiring>
```

- Constraint: `<monoid/>` should not be of `type` = *`unit`*

3. `<monoid/>`

- Pivotal att.: `type` = *`unit`*|*`free`*|*`product`* token, req.

3.1. `type` = *`unit`*

This type means 'no monoid' and gives the possibility of describing within the format graphs valued by numerical semirings only.

```
<monoid type=unit/>
```

- Constraint: not allowed in `<semiring type=series/>` (*cf.* 2.2)
  nor in `<monoid type=product/>` (*cf.* 3.3)

3.2. `type` = *`free`*

- Pivotal att.: `genKind` = *`simple`*|*`tuple`* token, req.
- Pivotal att.: `genDescrip` = *`enum`*|*`range`*|*`set`* token, req.

This attribute `genDescrip` is put for further development. No alternative value is described here.

3.2.1. `genKind` = *`simple`*

```
<monoid type=free genKind=simple genDescrip='enum' genSort=''>
   <writingData/>          opt.
   <monGen/>          1+ occ. req.
</monoid>
```

- Att.: `genSort` = *`letter`*|*`digit`*|*`alphanum`*|*`integer`* token, req.
- Tag `<writingData identitySymbol=''/>`
  `identitySymbol` = ' ' string,

Tells how the identity of the monoid should be written when output (*e.g.* in expressions)

3.2.2. `genKind` = *`tuple`*

```
<monoid type=free genKind=tuple genDim='' genDescrip='enum'>
   <writingData/>                    opt.
   <genSort>                         req.
      <genCompSort/>    "genDim"  occ. req.
   </genSort>
   <monGen/>                  1+  occ. req.
</monoid>
```

- Att.: genDim = ' ' integer strictly larger than 1, req.
- Tag <genSort/> holds the *"genDim"* <genCompSort/> tags
- Tag <genCompSort value=''/>
  value = ' ' has the same role as the attribute genSort in 3.2.1 for the corresponding coordinate of the monoid generator and can take the same token values.

3.3. type = *product*

```
<monoid type=product prodDim=''>
   <writingData/>                    opt.
   <monoid/>          "prodDim"  occ. req.
</monoid>
```

- Att.: prodDim = ' ' integer strictly larger than 1, req.

– Constraint: no children <monoid/> can be of type = *unit*

4. <monGen/>

Describes a monoid generator for a free monoid.
Its form will depend on the pivotal attribute genKind.
(The only case considered here is when genDescrip = *enum*.)

4.1. genKind = *simple*

```
<monGen value=''/>
```

- Att.: value must be consistent with genSort req.

4.2. genKind = *tuple*

```
<monGen>
   <monCompGen/>      "genDim" occ. req.
</monGen>
```

- Tag <monCompGen value=''/>

– Constraint: each value must be consistent with the corresponding genCompSort

3 The rational (regular) expressions

5. <regExp/>

```
<regExp name="">
   <valueType/>        req.
   <typedRegExp/>      req.
</regExp>
```

- Att.: name= ' ' string, opt.

– Constraint: the <monoid/> cannot be of type = *unit*

– At this stage, one could think of a <writingData/> tag which would contains writing options for the expressions: a dot or nothing for the product, delimitors for the weight, etc.

6. <typedRegExp/>

```
<typedRegExp>
   {Body::typedRegExp}
</typedRegExp>
```

plays the role of a non terminal in a grammar.

```
{Body::typedRegExp}= <sum/>|<product/>|<star/>|
                     <rightExtMul/>|<leftExtMul/>|
                     <zero/>|<one/>|<monElmt/>
```

7. <sum/>

```
<sum>
   {Body::typedRegExp}
   {Body::typedRegExp}
</sum>
```

8. <product/>

```
<product>
   {Body::typedRegExp}
   {Body::typedRegExp}
</product>
```

9. <star/>

```
<star>
   {Body::typedRegExp}
</star>
```

10. <rightExtMul/> or <leftExtMul/>

```
<xxxExtMul>
   <weight/>
   {Body::typedRegExp}
</xxxExtMul>
```

11. <zero/> and <one/> "final" tags

12. <monElmt/>

Depends on the pivotal attribute type of the <monoid/> in the <valueType/>.

12.1. type = *free*

```
<monElmt>
   <monGen/>
</monElmt>
```

1+ occ. req.

12.2. type = *product*

```
<monElmt>
   <one/>|<monElmt/>
</monElmt>
```

"prodDim" occ. req.

13. `<weight/>`

    Depends on the pivotal attribute `type` of the `<semiring/>` in the `<valueType/>`.

    13.1. `type` = *numerical*

    ```
    <weight value=''/>
    ```

    • Att.: `value`= ' ' string, that will be interpreted according to the attribute `set`

    – If `set` =*Q* , one can think of having 2 integers values.

    13.2. `type` = *series*

    ```
    <weight>
       {Body::typedRegExp}      of the <valueType/> defined by <semiring/>
    </weight>
    ```

4 The automata

14. `<automaton/>`

```
<automaton name='' readingDir=''>
   <geometricData/>          opt.
   <drawingData/>            opt.
   <valueType/>              req.
   <automatonStruct/>        req.
</automaton>
```

• Att.: `name` = ' ' string, opt.
`readingDir` = *left*|*right* token,
• Tag `<geometricData x='' y=''/>` gives relative origin
• Tag `<drawingData drawingClass='' />` or something more complicated

15. `<automStruct/>`

```
<automStruct>
   <states/>            req.
   <transitions/>       req.
</automStruct>
```

16. `<states/>`

```
<states>
   <state/>       0 or more occ.
</states>
```

17. `<state/>`

```
<state id='' name='' key='' >
   <geometricData/>       opt.
   <drawingData/>         opt.
</state>
```

- Att.: `id` = ' ' string, req. must be unique in the whole automaton.
  `name` = ' ' string, opt.
  `key` = ' ' integer opt. may be used to pass an ordering on the states.
- Tag `<geometricData x='' y=''/>` coordinates of the state
- Tag `<drawingData drawingClass='' />` or something more complicated

18. `<transitions/>`

```
<transitions>
   <transition/>      0 or more occ.
   <initial/>         0 or more occ.
   <final/>           0 or more occ.
</transitions>
```

19. `<transition/>`

```
<transition source='' target=''>
   <geometricData/>     opt.
   <drawingData/>       opt.
   <label/>             req.
</transition>
```

- Att.:`source` = ' ' string, req. must be a valid `id`
  `target` = ' ' string, req. must be a valid `id`
- Tag `<geometricData transitionType='' labelPos='' labelDist='' loopDir=''/>`
  - Pivotal att.: `transitionType` = *EdgeL*|*EdgeR*|*ArcL*|*ArcR* token, req.
    `source` must be different from `target`
    `transitionType` = *Loop* `source` must be equal to `target`
  - Att.: `loopDir` = *N*|*S*|*E*|*W*|*NE*|*NW*|*SE*|*SW* token, req.
    or integer between 0 and 360 (*E* 0)
    but only if `transitionType` = *Loop*
    `labelPos` = ' ' float opt.
    `labelDist` = ' ' float opt.
- Tag `<drawingData drawingClass='' />` or something more complicated

20. `<initial/>`

```
<initial state=''>
   <geometricData/>     opt.
   <drawingData/>       opt.
   <label/>             req.
</initial>
```

- Att.: `state` = ' ' string, req. must be a valid `id`
- Tag `<geometricData initialDir='' labelPos='' labelDist=''/>`
  (*cf.* 19)
- Tag `<drawingData drawingClass='' />` or something more complicated

It is assumed that initial states are marked with an incoming arrow.

21. `<final/>`

```
<final state=''>
   <geometricData/>      opt.
   <drawingData/>        opt.
   <label/>              req.
</final>
```

- Att.: `state` = ' ' string, req. must be a valid `id`
- Tag `<geometricData finalMod='' finalDir='' labelPos='' labelDist=''/>`
  - Pivotal att.: `finalMod` = *`circle`*|*`arrow`* token, req.
  - Other att: *cf.* 19.
- Tag `<drawingData drawingClass='' />` or something more complicated

22. `<label/>`

```
<label>
   {Body::typedRegExp}
</label>
```