

# Vaucanson XML developement documentation.

Valentin David, valentin@lrde.epita.fr

December 2003

This document describe the Vaucanson XML developement.

## Contents

<b>1</b>	<b>Static mode</b>	<b>1</b>
<b>2</b>	<b>Dynamic mode</b>	<b>2</b>
<b>3</b>	<b>Generic mode</b>	<b>4</b>
<b>4</b>	<b>Error processing</b>	<b>5</b>
<b>5</b>	<b>Geometry</b>	<b>5</b>
5.1	Keeping . . . . .	5
5.2	Accessing . . . . .	6
<b>6</b>	<b>Sessions</b>	<b>7</b>

There is three way to use the Vaucanson XML interface.

- static mode
- dynamic mode
- pseudo-generic mode

## 1 Static mode

This mode use the standard Vaucanson I/O system. An instance of the `vcsn::xml::xml_loader` class has to be passed to the `automaton_loader` and `automaton_saver` functions.

---

```
#include <vaucanson/xml/static.hh>
#include <vaucanson/tools/usual.hh>
#include <iostream>

int main()
{
    using namespace vcsn;
    using namespace vcsn::tools;
    using namespace vcsn::xml;

    XML_BEGIN;
    AUTOMATON_TYPES_EXACT(usual_automaton_t);
```

```

    automaton_t a;

    std::cin >> automaton_loader(a, io::string_out(),
                                   xml::xml_loader());

    /* work on a */

    std::cout << automaton_saver(a, io::string_out(),
                                   xml::xml_loader());

    XML_END;
}

```

---

## 2 Dynamic mode

This mode able to choose the implementation following the type of the automaton described in the XML Document.

The automaton is first loaded in a `xml_automaton_t` typed variable. Then the type can be accessed.

Here is an easy to understand example :

---

```

#include <vaucanson/xml/dynamic.hh>
#include <vaucanson/tools/usual.hh>
#include <iostream>

int main()
{
    xml_automaton_t dynamic_automaton;

    std::cin >> dynamic_automaton;

    if (x.set().semiring_set() == XmlSet::B) {
        AUTOMATON_TYPES_EXACT(usual_automaton_t);
        XMLOF(automaton_t) xml_automaton = dynamic_automaton;
        automaton_t automaton = xml_automaton;

        /* work on automaton */

        xml_automaton = automaton;
        std::cout << xml_automaton;
    }

    if ((x.set().semiring_type() == XmlSet::NUMERICAL)
        && (x.set().semiring_set() == XmlSet::Z)) {
        AUTOMATON_TYPES_EXACT(weighted_automaton_t);
        XMLOF(automaton_t) xml_automaton = dynamic_automaton;
        automaton_t automaton = xml_automaton;

        /* work on automaton */
    }
}

```

```

    xml_automaton = automaton;
    std::cout << xml_automaton;
}

return 0;
}

```

---

But this example can be written better using generic programming.

---

```

#include <vaucanson/xml/dynamic.hh>
#include <vaucanson/tools/usual.hh>
#include <iostream>

template <typename T>
int print_out(typename XMLOF(T) x)
{
    using namespace vcsn;
    using namespace vcsn::tools;
    using namespace vcsn::xml;

    AUTOMATON_TYPES_EXACT(T);

    automaton_t a = x;

    /* work on a */

    x = a;
    std::cout << x;
    return 0;
}

int main()
{
    using namespace vcsn;
    using namespace vcsn::tools;
    using namespace vcsn::xml;

    XML_BEGIN;

    xml_automaton_t x;

    std::cin >> x;

    if ((x.set().semiring_type() == XmlSet::NUMERICAL)
        && (x.set().semiring_set() == XmlSet::Z))
        return print_out<numerical_automaton_t>(x);

    if (x.set().semiring_set() == XmlSet::B)
        return print_out<usual_automaton_t>(x);
}

```

---

```
std::cerr << "Automaton type not dealt" << std::endl;
return -1;
}
```

---

### 3 Generic mode

The generic mode is very long to compile. So, some constants have to be set for activating some parts.

- VCSN\_XML\_GENRERIC\_WEIGHTED: Enable "Z" and "R" for semiring.
  - VCSN\_XML\_GENRERIC\_TRANSDUCERS: Enable "ratseries" for semiring (no recursion).
  - VCSN\_XML\_GENRERIC\_CHAR\_PAIRS: Enable "pair" of letters for free monoid.
  - VCSN\_XML\_GENRERIC\_WEIGHTED\_LETTERS: Enable "weighted" letters for free monoid.
  - VCSN\_XML\_GENRERIC\_INT\_LETTERS: Enable "integers" for free monoid.
- 

```
#define VCSN_XML_GENRERIC_WEIGHTED 1

#include <vaucanson/xml/generic.hh>
#include <vaucanson/tools/usual.hh>
#include <iostream>

struct MyData
{
    int argc;
    char **argv;
    MyData(int c, char** v) : argc(c), argv(v) { }
};

template <typename Auto>
struct MyCallBack
{
    int operator()(Auto& a, MyData&)
    {
        using namespace vcsn;
        using namespace vcsn::tools;
        using namespace vcsn::xml;

        AUTOMATON_TYPES(Auto);

        /* work on a */

        return 0;
    }
};

int main(int argc, char *argv[])
{
    using namespace vcsn;
```

---

```

using namespace vcsn::xml;

XML_BEGIN;

MyData data(argc, argv);
return apply<MyCallBack, MyData>(std::cin, data);
}

```

---

## 4 Error processing

There are two possibilities, for error processing.

- The program exit on failure.
- The program raise an exception on failure (default).

To switch this mode a macro has to be set.

- `#define FAIL WITH_EXIT` for exiting.
- `#define FAIL WITH_THROW` for raising exception.

When raising exceptions, the raised exception is mainly a `vcsn::xml::LoaderException` one. This exception provides the method `get_msg()`.

---

```

xml_automaton_t x;

try {
    std::cin >> x;
}
catch (const xml::LoaderException& e) {
    std::cerr << "XML parser error: " << e.get_msg() << std::endl;
    return -1;
}

```

---

## 5 Geometry

### 5.1 Keeping

To keep the geometry in a graph implemented automaton, the type has to be converted to accept attached data.

---

```

template <typename T>
int print_out(typename XMLOF(T) x)
{
    using namespace vcsn;
    using namespace vcsn::tools;
    using namespace vcsn::xml;

```

```

typedef typename ATTACHXMLINFOS(T) my_automaton_t;
AUTOMATON_TYPES(my_automaton_t);

automaton_t a = x;

/* then, here, a contains the geometry */

return 0;
}

```

---

## 5.2 Accessing

The data are in the `Tag`. This tag is `XmlInfosTag` typed. See Vaucanson API documentation for more information.

---

```

// This example align states diagonally with a depth-first traversal.
template<typename I>
void align(I& a)
{
    AUTOMATON_TYPES(I);
    int x = 0;
    std::map<hstate_t, bool> visited;
    std::stack<hstate_t> stack;

    for_each_state(i, a) {
        visited[*i] = false;
        // ensure inaccessible states will be visited
        stack.push(*i);
    }

    for_each_initial_state(i, a)
        stack.push(*i);

    while (!stack.empty()) {
        hstate_t i = stack.top();
        stack.pop();

        if (!visited[i]) {
            visited[i] = true;

            a.tag().states[i]().x = a.tag().states[i]().y = x++;

            std::list<hedge_t> aim;
            a.deltac(aim, i, delta_kind::edges());
            for_all_const_(std::list<hedge_t>, j, aim)
                stack.push(a.aim_of(*j));
        }
    }
}

```

---

## 6 Sessions

Sessions are handled like life by class `xml::XmlSession`. `<<` and `>>` operators are use for popping un pushing.

---

```
#include <vaucanson/xml/session.hh>
#include <vaucanson/tools/usual.hh>
#include <iostream>

int main()
{
    using namespace vcsn;
    using namespace vcsn::tools;
    using namespace vcsn::xml;

    XML_BEGIN;

    typedef XMLOF(usual_automaton_t) my_xml_automaton_t;
    AUTOMATON_TYPES_EXACT(usual_automaton_t);

    automaton_t a;
    my_xml_automaton_t x;
    xml_automaton_t dyn;

    XmlSession session1, session2;
    std::cin >> session1;
    session1 >> dyn;
    a = my_xml_automaton_t(dyn);

    /* work on a */

    session2 << my_xml_automaton_t(a);

    session >> dyn;
    a = my_xml_automaton_t(dyn);

    /* work on a */

    session2 << my_xml_automaton_t(a);

    XML_END;

    return 0;
}
```

---