Vaucanson – a tutorial

June 20, 2003

Contents

1 Overview **2**
- 1.1 Presentation . . . 2
- 1.2 Distribution, installation, basic usage . . . 2
- 1.3 Directory tree . . . 2
- 1.4 Aim of this document . . . 3
- 1.5 A simple example . . . 3

2 Preliminaries **4**

3 Fundamental: enriched c++ **4**
- 3.1 Element/Set design pattern . . . 4
  - 3.1.1 Basics . . . 4
  - 3.1.2 A short example . . . 5
  - 3.1.3 Inheritance between Element . . . 5
  - 3.1.4 Orthogonal specialization . . . 5
  - 3.1.5 Dynamic/Static properties . . . 5

4 Dealing with algebraic structures **5**
- 4.1 Alphabet . . . 5
- 4.2 Free monoid . . . 7
- 4.3 Semiring . . . 9
- 4.4 Series . . . 10
- 4.5 Rational expression . . . 11

5 Automaton **12**
- 5.1 Instantiating an automaton . . . 12
  - 5.1.1 Structure operations . . . 12
  - 5.1.2 Delta function . . . 13
- 5.2 Standard services . . . 13
- 5.3 Standard macros . . . 13
- 5.4 Usage example . . . 13

6 Algorithms **14**
- 6.1 General form . . . 14
- 6.2 Manipulation . . . 14
- 6.3 Determinisation . . . 14
- 6.4 Minimisation . . . 14

7 Grammar inference **14**

8 Your 'grep' 14
8.1 From a regular expression to an automaton 14
8.2 Speed consideration 14
8.3 Multiplicity in action 14

9 Play with multiplicity 14

10 Advanced use 14
10.1 Extending Vaucanson 14
10.2 Glossary 14

1 Overview

1.1 Presentation

Vaucanson is a C++ library for finite state machine manipulation.
Vaucanson is:

- **generic**: a general algorithm is written once and is instantiated for the good parameters at use ;
- **algorithm oriented**: the system is meant to provide primitive services to write algorithms ;
- **meta**: the C++ is enriched to obtain a flexible framework.

1.2 Distribution, installation, basic usage

The tarball can be found at `http://www.lrde.epita.fr/twiki/VaucansonLib`.
The installation is the classical:

```
./configure
make
make install (as root)
```

If you are not root on your system, you can install Vaucanson on your home directory by typing:

```
./configure --with-prefix=$HOME/include/
make
make install
```

The only difference is that you must specify a -I $(HOME)/include in your compilation flags.

1.3 Directory tree

The Vaucanson library is organized as follows:

```
'-- vaucanson
    |-- algebra
    |   |-- concept
    |   '-- concrete
    |       |-- alphabets
    |       |-- free_monoid
    |       |-- letter
    |       |-- semiring
```

```
|       '-- series
|           '-- rat
|-- algorithms
|-- automata
|   |-- concept
|   '-- concrete
|-- config
|-- fundamental
|-- internal
|-- misc
'-- tools
```

- **fundamental**: the core of the library, its goal is to enriched C++. (see section **??**) ;
- **config**: the internal configuration system ;
- **internal**: the internal C++ headers ;
- **algebra**: the algebra module ;
- **automata**: the automata module ;
- **algorithms**: the algorithms set ;
- **misc**: some tools to interact with external tools ;
- **tools**: some useful tools for daily work.

1.4 Aim of this document

The goal of this document is to present the philosophy of the library and to demonstrate some features of the system in a practical manner.

1.5 A simple example

At the end of this tutorial, you will probably be able to write a generic algorithm that complete a finite state machine like this one:

A simple example

```
template <class T>
void complete(Element<Automata, T>& auto)
{
  typedef Element<Automata, T> automaton_t;
  AUTOMATON_TYPES(automaton_t);

  const alphabet_t& alphabet = auto.series().monoid().alphabet();
  std::set<hedge_t> delta_ret;
  hstate_t          hole_state = auto.add_state();

  for_each_state(s, auto)
    for_each_letter(e, alphabet)
    {
      delta_ret.clear();
      a.letter_deltac(delta_ret, *s, *e);
      if (delta_ret.size() == 0)
        auto.add_letter_edge(*s, hole_state, *e);
```

```
    }
}
```

---

2 Preliminaries

3 Fundamental: enriched c++

The fundamental module provides some sugar to build the system genericity. It can be seen as the core of the system and, then, non-expert users do not have to understand it. Yet, a so-called 'Element' design pattern is used in every lines of Vaucanson to enable genericity: one must be aware of it before coding with the Vaucanson library.

3.1 Element/Set design pattern

Vaucanson is designed in an algebraic manner. To reinforce this view and to provide a simple way to separate implementation and theoritical behaviour, we use a unified way to structure all the objects used in the library: the Element pattern. The 'Basics' section is essential whereas the other can be skipped for a first reading.

3.1.1 Basics

First, in Vaucanson, everything is an instance of **vcsn::Element<S, T>** where the 'S' parameter is interpreted as the set of the element and 'T' as its implementation. For example, Element<Series, polynom> is an element of the series implemented with the polynom class. What is important is that every Element<S, T> provides the same interface. You can access the implementation by the 'value()' method and the set by the 'set()' one. As a consequence, objects are build incrementally from their set.

Use of the element services.

```
Alphabet a;
FreeMonoid f(a);
Semiring s;
Series s(f, s);
Element<Series, polynom<std::string, int> > s;
Element<FreeMonoid, std::string> m(f);
Element<Semiring, int>   w;
s.set();          // the series set.
s.set().monoid(); // the monoid.
s.value();        // the polynom.
m.set().alphabet() == s.set().monoid().alphabet();
s.assoc(m, w);
```

Second, the particular behaviour of an implementation 'T' viewed as an element of a set 'S' is defined by specialization of the class **vcsn::MetaElement<S, T>**. For instance, the MetaElement<Series, polynom> provides a 'is_stareable()' method that returns true if we can take the star of the current serie. The MetaElement class is not a user class, it is just a way to define the interpretation of the implementation as a particular concept.

3.1.2 A short example

3.1.3 Inheritance between Element

3.1.4 Orthogonal specialization

3.1.5 Dynamic/Static properties

4 Dealing with algebraic structures

Even if classical context are provided (like letter acceptors), Vaucanson enables the user to precisely define the algebraic context to work with.

Next sections show how it can be done for the main algebraic components: alphabet, free monoid, semiring, serie and rational expression.

Thanks to the Element design pattern, these constructions are unified: you first define the set of your element and then one (or several) instance(s) of it.

4.1 Alphabet

concept:	**alphabet, letter**
concerned files:	algebra/concept/alphabets_base.h[h, xx]
	algebra/concrete/alphabet/*

In Vaucanson, the set of all alphabets over a particular type of letter is denoted by the AlphabetSetBase abstract class. Consequently, to get all the services you can expect from an alphabet, you must look at MetaElement specialized w.r.t AlphabetSetBase (in algebra/concept/alphabets_base.hh)

A final class AlphabetSet implements this class in a trivial manner: it is an empty class only for static typing purpose. This class is just parameterized by the type of letters. So, we just have to provide the kind of letter we wish to manipulate inside our future alphabet. You can use the C++-*builtins* types (like char, int, ...) or use more specific types proposed in Vaucanson.

For example, let us define a type alias declaring that we want to work with letter implemented with 'char':

Letter implemented with char

```
typedef char Letter;
```

Sometimes, it is useful to work only on a subset of the 'char' type, that's why you can use the 'static_ranged' class:

Letter implemented with static_ranged

```
typedef static_ranged<char, static_char_interval<'a','z'> >  Letter;
```

If you choose this last option, symbols can only be letters between 'a' and 'z', nothing more. The set of all the alphabets that contain this type of letter is expressed in Vaucanson with:

Type denoting the set of alphabet holding letter of type 'Letter'

```
typedef AlphabetSet<Letter>  Alphabets;
```

An element of the alphabet set (*ie* an alphabet) can be implemented in many ways. For example, we can use std::set or std::list for dynamic alphabets. Lastly, we can create the type of the alphabet we will effectively use, with the Element pattern:

Type of an alphabet implemented with std::set structure

```
typedef Element<Alphabets, std::set<Letter> >  Alphabet;
```

This previous type should be understood like: "an element of the set of alphabets holding 'Letter' letter and implemented by a std::set of 'Letter' ". Actually, this code is already present inside Vaucanson, more exactly inside the file "`algebra/concrete/alphabet/predefs.hh`".

The pre-existing alphabet definitions in Vaucanson

```
namespace small_alpha_letter {

     typedef static_ranged<char, static_char_interval<'a','z'> >  Letter;
     typedef AlphabetSet<Letter>                                  Alphabets;
     typedef Element<Alphabets, std::set<Letter> >                Alphabet;

   } // small_alpha_letter

   namespace char_letter {

     typedef AlphabetSet<char>                                    Alphabets;
     typedef Element<Alphabets, std::set<char> >                  Alphabet;

   } // char_letter

   namespace int_letter {

     typedef AlphabetSet<int>                                     Alphabets;
     typedef Element<Alphabets, std::set<int> >                   Alphabet;

   } // int_letter
```

You only have to choose the right namespace for your work, or create a new.
To finish, here is a short example of alphabet manipulation :

Example of alphabet manipulation

```
using namespace vcsn;
using namespace algebra;
using namespace small_alpha_letter;

using std::cout;
using std::endl;

Alphabet A;
Letter a('a');

A.insert(a);
```

```
  A.insert('b');

  cout << "Size of alphabet : "
       << A.size() << endl;                // return 2

  cout << "Is \'a\' inside alphabet ? (0 or 1) : "
       << A.contains('a') << endl;         //return true

  cout << "Is \'1\' inside alphabet ? (0 or 1) : "
       << A.contains('1') << endl;         //return false

  cout << "element of alphabet are : ";
  for (Alphabet::iterator i = A.begin(); i != A.end(); i++)
    cout << *i << " ";
  cout << endl;

  cout << "random sequence of 10 symbols of the alphabet : ";
  for (unsigned i = 0; i < 10; i++)
    cout << A.choose() << " ";
  cout << endl;
```

4.2 Free monoid

concept:	**Free monoid**
concerned files:	algebra/concept/freemonoid_base.h[h, xx]
	algebra/concrete/freemonoid/*

Alphabet and letter are not enough to express language manipulation, we need the word concept. A word is a sequence of letters.

The algebraic structure denoting the set of words is called a **free monoid**. Such a set needs an alphabet to be characterized. Usually, we write "A^*" for the free monoid generated by an alphabet A.

A free monoid is fully characterized by a concatenation operation, denoted as a product '.', an empty word and an alphabet. In Vaucanson, this concept is managed by the FreeMonoidBase abstract class. For example, we can write such a function:

A function that manipulates a Free Monoid

```
template <class Self>
unsigned free_monoid_alphabet_cardinal(const FreeMonoidBase<Self>& fm)
{
  return fm.alphabet().size();
}
```

A natural definition for a concrete free monoid is the FreeMonoid class which aggregates an instance of the alphabet:

An instance of a free monoid over the alphabet $\{a, b\}$

```
{
  using small_alpha_letter;
  typedef FreeMonoid<Alphabet>  Words;
```

```
  Alphabet     alphabet;
  alphabet.insert('a');
  alphabet.insert('b');
  FreeMonoid<Alphabet> free_monoid(alphabet);
  free_monoid_alphabet_cardinal(free_monoid);
}
```

A predefinition of this class exists in `algebra/concrete/free_monoid/predefs.hh`. It is called 'Words'.

Now, let's work with element of this set. Naturally, a word can be implemented with a `std::basic_string<Letter>`.

Then, its type using the Element class is:

An element of a free monoid implemented with std::string

```
{
  Element<Words, std::basic_string<Letter> > word;
}
```

A predefinition of this class exists in algebra/concrete/free_monoid/predefs.hh, it is called 'Word'.

As a sequence, a word provides iterators and reverse iterators.

Free monoid element iterators

```
  Word str(free_monoid);
  str = "abb";
  for (Word::iterator i = str.begin(); i != str.end(); i++)
    cout << *i << " ";
  cout << endl;
  for (Word::reverse_iterator i = str.rbegin(); i != str.rend(); i++)
    cout << *i << " ";
  cout << endl;
```

Some others services and external functions are proposed:

Free monoid element services

```
  Word str(free_monoid);
  str = "abb";
  // Side effect.
  str.mirror();
  cout << str << endl;
  // Pure functional mirroring.
  cout << mirror(str) << endl;
  cout << str.length() << endl;
```

To obtain all the services of a word, please look at "`algebra/free_monoid/concept/freemonoid_base.hh`

How to get the neutral element of the semiring (the empty word) ? It is not the role of an element to provide it, the set is the only owner of this information. Yet, the set is not aware of the implementation to choose to instantiate the neutral element !

The solution is to precise to the set the implementation we want. To do that, we use a tricky typed null pointer:

Get the empty string

```
free_monoid.identity((std::string*)(0));   // horrible but it works.
free_monoid.identity(SELECT(std::string)); // macro to hide it.
identity_as<std::string>::of(free_monoid); // solution using a template function.
```

4.3 Semiring

concept:	**Semiring**
concerned files:	algebra/concept/semiring_base.h[h, xx]
	algebra/concrete/semiring/*

To characterize some words of the language, we usually use element from a semiring. A semiring is a set which provides a multiplication, an addition and neutral elements for each of these operations. Consequently, we can compute the "score" of every words in function of these primitive operations.

The unweighted finite state machines are also called "acceptors". In fact, there are just weighted with boolean. Consequently, we can instantiate the Boolean semiring:

The Boolean semiring

```
BooleanSemiring semiring;
```

If you look at algebra/concept/boolean_semiring.hh, you will see that the BooleanSemiring is an empty class. Indeed, there is no dynamic data associated to it: it is only present for static purpose.

A straightforward implementation of a boolean weight is the builtin "bool". Vaucanson provides such an implementation:

A boolean in action !

```
Element<BooleanSemiring, bool> t = true;
Element<BooleanSemiring, bool> f = false;
std::cout << t * f << std::endl;
std::cout << t + f << std::endl;
```

As with free monoid, particular elements can be retrieved using the set:

Get the identity and the zero of the semiring

```
semiring.identity(SELECT(bool));
semiring.identity((bool*)(0));
zero_as<bool>::of(semiring);
```

Some other semirings are useful for weighted automaton. For example, Vaucanson provides the famous MaxTropical semiring. An element of such a semiring can be an integer encoded as an 'int' builtin:

(Z, Max, +) tropical semiring

```
MaxTropical semiring;
Element<MaxTropical, int> a = 1;
Element<MaxTropical, int> b = zero_as<int>::of(semiring);
Element<MaxTropical, int> c = identity_as<int>::of(semiring);

cout << a << endl;
cout << b << endl;
cout << c << endl;
```

A really important operation is the "star" of a semiring element x that is defined by:

$$x^\star = \sum_{k=0}^{\infty} x^k$$

The star is useful when you want to symbolize a non bounded computing of weight (for example, when you want to associate a weight to a star regular expression). Yet, the star operation is not defined everywhere. Consequently, Vaucanson provides two services star() and is_stareable():

The star of a semiring element

```
Element<MaxTropical, int> a = 1;
Element<MaxTropical, int> b = -1;
Element<MaxTropical, int> z = zero_as<int>::of(semiring);
Element<MaxTropical, int> i = identity_as<int>::of(semiring);
cout << a.is_stareable() << std::endl;
cout << b.is_stareable() << std::endl;
cout << z.is_stareable() << std::endl;
cout << i.is_stareable() << std::endl;
cout << z.star() << std::endl;
```

4.4 Series

concept:	**Series**
concerned files:	algebra/concept/series_base.h[h, xx]
	algebra/concrete/series/*

We want to manipulate the mapping between words and weights as object. This kind of object is called formal series. Vaucanson enables the construction of series element.

The set of series is constructed from a free monoid and a semiring:

The set of series

```
Series<BooleanSemiring, FreeMonoid<Alphabet> > series(freemonoid, semiring);
```

Then, we can implement a finite serie using a finite support map. Vaucanson provides such an implementation through the polynoms class:

A serie

```
Element<BooleanSeries, polynom<std::string, bool> > serie(series);
```

As a map, a serie can associate a weight to a word:

Serie's services

```
serie.assoc(''foo'') = true;
cout << serie.get(''foo'') << endl;
cout << serie.get(''baz'') << endl;
```

The set of series can be seen as a semiring, then a serie inherits all the services of a semiring element:

A serie as a semiring

```
serie.star();
Element<BooleanSeries, polynom<std::string, bool> > serie2(series);
serie2.assoc(''a'') = true;
Element<BooleanSeries, polynom<std::string, bool> > serie3(series);
serie3.assoc(''a'') = true;
cout << serie1 + serie2 << endl;
cout << serie1 + serie3 << endl;
cout << serie2 * serie1 << endl;
cout << serie1.is_stareable() << endl;
```

4.5 Rational expression

The polynom implementation does not manage infinite series. Indeed, we can't denote such serie in an extensional way. Yet, a particular class of infinite series can be expressed using a finite formalism: the rational expressions. The Kleene theorem says that this class of series is exactly what can be recognized by a finite state machine.

Vaucanson enables us to define such series:

A rational expression

```
Element<BooleanSeries, polynom<std::string, bool> > serie(series);
series = "a";
Element<BooleanSeries, polynom<std::string, bool> > serie2(series);
series2 = "ab";
Element<BooleanSeries, polynom<std::string, bool> > serie3(series);
series3 = "b";
cout << serie1 + serie2 << endl;
cout << serie1 + serie3 << endl;
cout << serie2 * serie1 << endl;
cout << serie1.is_stareable() << endl;
cout << (serie1.star() + serie2) * serie3 << endl;
```

As syntactic expression, rational expresssion can be parsed:

```
parse("(a+b*).ab", exp);
std::cout << exp << std::endl;
```

5 Automaton

FIXME: definition automate.

An automaton is both a theorical and a computational object. Therefore, the services must be both general and atomic as possible.

However, the underlying data structure can be more or less adapted to each of these trends. For example, a graph with letter as label is a great structure for computation but to be seen as series the label must be enriched. At the contrary, a graph with series as label will be a great theorical object but will be heavy in memory and at computation.

A basic data structure must be enough to define the primitive services but some sugar is necessary to help the user. FIXME: exemple !

Finally, automaton are big structures, they must be reference-counted.

5.1 Instantiating an automaton

concept:	**Automaton**
concerned files:	automata/concept/automata_base.h[h, xx]

A set of automaton is characterized by a series set. So, first, we have to define it:

The set of automata

```
Automata automata_set(series);
cout << automata_set.series().monoid().alphabet() << endl;
```

The implementation of an automaton is the composition of 3 three things:

- a kind
- an implementation adapter
- a reference counter

The kind is parameterized by the type of the automaton label and adapt final services consequently. The implementation adapter is responsible of sugar. The reference counter suppress unuseful copy.

5.1.1 Structure operations

As a graph, an automaton is composed of states and edges. Usually, handlers are used to represent nodes and edges. In fact, handlers are comparable to unsigned integer.

The automaton is able to create and remove nodes and edges :

Structural operations

```
hstate_t s1 = a.add_state();
hstate_t s2 = a.add_state();
hedge_t e1 = a.add_letter_edge(s1, s2, 'a');
hedge_t e2 = a.add_letter_edge(s1, s2, 'b');
hstate_t s3 = a.add_state();
hedge_t e3 = a.add_edge(s1, s3, 'a');
hedge_t e3 = a.add_spontaneous(s1, s3);
a.del_state(s3);
a.del_edge(s3);
```

When you remove a state/edge, the other handlers are not updated.

5.1.2 Delta function

The delta function represents the outer transitions associated to a state. However, there are many ways of viewing this set.

what kind of information to retrieve ? The last argument of delta is the type of the retrieve information. It can be hstate_t or hedge_t.

where to store the information ? Their are two kinds of return storing: you can use a standard container or an output iterator. FIXME: other kind of DELTA with two iterators !

what criteria to filter transition ? The simplest way to select transitions is to get them all. Another way is quite natural: you can specify a letter that must matches with the label. The last one is to use an object function that represents the criteria as an operator() with a label as argument.

5.2 Standard services

The set of states and the set of edges are provided to permit to iterate over the states and the edges.

The initial and the final applications are modelised using mapping.

Additional information can be attached to the nodes or to the edges: this is what is called Tag in Vaucanson.

5.3 Standard macros

To help the user when writing automaton algorithms, a set of macros is provided.

5.4 Usage example

FIXME: fusion max !

6 Algorithms

6.1 General form

6.2 Manipulation

6.3 Determinisation

6.4 Minimisation

7 Grammar inference

MAX !

8 Your 'grep'

'grep' is a well known unix tools used to look for pattern in a text. The pattern is described as a rationale expression: if a substring of the line is in the language denoted by the expression, the line is print in the output.

A naive implementation of grep uses an automaton of the form:

8.1 From a regular expression to an automaton

There exists many algorithms to construct an automaton that recognizes the language denoted by an expression. The most popular is the Thompson's one.

A naive grep

```
Ok
```

8.2 Speed consideration

Do not be pessimistic, we can improve our naive implementation of grep.

What implementation ?

Algorithm's refinement ...

8.3 Multiplicity in action

9 Play with multiplicity

Imagine now that we want to determine the shortest length of the pattern that has been matched.

10 Advanced use

10.1 Extending Vaucanson

10.2 Glossary