The Vaucanson TAF-Kit 1.0 User's Manual

The Vaucanson Group

July 28th, 2006

Contents

Introduction

The VAUCANSON software platform is dedicated to the computation with finite state automata. Here, 'finite state automata' is to be understood in the broadest sense: *weighted* automata on a free monoid — that is, automata that not only accept, or recognize, *words* but compute for every word a *multiplicity* which is taken a priori in *an arbitrary semiring* — and even weighted automata on *non free monoids*. The latter become far too general objects. As for now, are implemented in VAUCANSON only the (weighted) automata on (direct) products of free monoids, machines that are often called *transducers* — that is automata that realize (weighted) relations between words[1].

When designing VAUCANSON, we had three main goals in mind: we wanted

1. a *general purpose* software,
2. a software that allows a programming style natural to computer scientists who work with automata and transducers,
3. an open and free software.

This is the reason why we implemented so to say *on top* of the VAUCANSON platform a library that allows to apply a number of functions on automata, and even to define and edit automata, without having to bother with subtleties of C++ programming. The drawback of this is obviously that the user is given a *fixed* set of functions that apply to *already typed* automata. This library of functions does not allow to write new algorithms on automata but permits to combine or compose without much difficulties nor efforts a rather large set of commands. We call it TAF-KIT, standing for *Typed Automata Function Kit*, as these commands take as input, and output, automata whose type is fixed. TAF-KIT is presented in Chapter 2.

[1] When the relation is "weighted" the multiplicity has to be taken in a *commutative* semiring.

Chapter 1

Installation

1.1 Getting Vaucanson

The latest stable version of the VAUCANSON platform can be downloaded from http://vaucanson.lrde.epita.fr/. The current development version can be retrieved from its Subversion[1] repository as follows:

```
# svn checkout https://svn.lrde.epita.fr/svn/vaucanson/trunk vaucanson
```

1.2 Building Vaucanson

The following commands build and install the platform:

```
# cd vaucanson-1.0
```

Then:

```
# ./configure
...
# make
...
# sudo make install
...
```

More detailed information is provided in the files ‘INSTALL’, which is generic to all packages using the GNU Build System, and ‘README’ which details VAUCANSON’s specific build process.

[1]Subversion can be found at http://subversion.tigris.org/.

Chapter 2

The Vaucanson toolkit

This chapter presents a simple interface to VAUCANSON: a set of programs tailored to be used from a traditional shell. Since they exchange *typed* XML files, there is one program per automaton type. Each program supports a set of operations which depends on the type of the automaton.

Many users of automata consider only automata whose transitions are labeled by letters taken in an alphabet, which we call, roughly speaking, *classical* automata or *Boolean* automata. The first program of the TAF-KIT, `vcsn-b`, allows to compute with classical automata and is described in Section 2.1.

Section 2.2 describes the program `vcsn-tdc` which allows to compute with transducers, that is, automata whose transitions are labeled by pair of words, which are elements of a *product of free monoids*, hence the name.

In Section 2.3 we consider the programs of the TAF-KIT that compute with automata over a free monoid and with multiplicity, or *weight* taken in the set of integers equipped with the usual operations of addition and multiplication, that is, the semiring $\mathbb{Z}$.

It is planned that a forthcoming version will include also:

vcsn-zmin for automata over a free monoid with multiplicity in the semiring $(\mathbb{Z}, \min, +)$

vcsn-zmax for automata over a free monoid with multiplicity in the semiring $(\mathbb{Z}, \max, +)$

vcsn-rw-tdc for transducers viewed as automata over a free monoid with multiplicity in the semiring of rational sets (or series) over (another) free monoid.

2.1 Boolean automata

This section focuses on the program `vcsn-b`, the TAF-Kit component dedicated to Boolean automata.

2.1.1 First Contacts

`vcsn-b` and its peer components of TAF-Kit all share the same simple interface:

```
# vcsn-b function automaton arguments...
```

The ***function*** is the name of the operation to perform on the ***automaton***, specified as an XML file. Some functions, such as evaluation, will require additional arguments, such as the word to evaluate. Some other functions, such as '`exp-to-aut`' do not have an ***automaton*** argument.

TAF-Kit is made to work with Unix *pipes*, that is to say, chains of commands which feed each other. Therefore, all the functions produce a result on the standard output, and if an ***automaton*** is '`-`', then the standard input is used.

A typical line of commands from the TAF-Kit reads as follows:

```
# vcsn-b determinize a1.xml > a1det.xml
```

and should be understood, or analyzed, as follows.

1. `vcsn-b` is the call to a `shell` command that will launch a Vaucanson function. `vcsn-b` has 2 arguments, the first one being the ***function*** which will be launched, the second being the ***automaton*** that is the input argument of the ***function***.
2. `determinize` is, as just said, a Vaucanson function. And as it can easily be guessed, `determinize` takes an ***automaton*** as argument, performs the subset construction on it and outputs the result on the standard output.
3. '`a1.xml`' is the description of an automaton — of the automaton of **??** indeed — in an XML format that is understood[1] by Vaucanson. Which means that this file must exist before the line is executed. The '`data/automata`' directory provides a number of XML files for examples of automata, a number of programs that produce the XML files for automata whose definition depend upon some variables and the TAF-Kit itself allow to define automata and thus to produce the corresponding XML files (cf. below).
4. `>` '`a1det.xml`' puts the result of `determinize` '`a1.xml`', that is, the XML file which describes the determinized automaton of $\mathcal{A}_1$ into the file '`a1det.xml`'.

As a more elaborate example, consider the following command

```
# vcsn-b dump-automaton a1 | vcsn-b determinize - | vcsn-b minimize - | vcsn-b info -
States: 3
Transitions: 6
Initial states: 1
Final states: 1
```

It fetches the automaton `a1` from the automaton library, determinizes it, minimizes the result, and finally displays information about the resulting automaton.

Please, note the typographic conventions: user input is represented `# like this`, standard output follows `like this`, followed by standard error output `error: like this`, and finally, if different from 0, the exit status is represented `=> like this`. For instance:

[1]This format is not exactly part of the Vaucanson platform. It has been developed for providing a mean of communication between various programs dealing with automata. And then it has been used as a communication tool between the invocation of Vaucanson function by the TAF-Kit. A lay user of the TAF-Kit should not need to know how this format is defined but a rough description of it is provided at **??** of the Appendix.

```
# vcsn-b dump-automaton a1 | vcsn-z info -
error: Bad semiring
=> 1
```

Other than that, the interface of the TAF-Kit components is usual, including options such as '`--version`' and '`--help`':

```
# vcsn-b --help
Usage: vcsn-b [OPTION...] <command> <args...>
VCSN TAF-Kit -- a toolkit for working with automata

  -a, --alphabet=ALPHABET    Set the working alphabet for rational expressions
  -l, --list-commands        List the commands handled by the program
  -v, --verbose              Be more verbose (print boolean results)

 The following alphabets are predefined:
   'ascii': Use all the ascii table as the alphabet, 1 as epsilon
   'a-z': Use [a-z] as the alphabet, 1 as epsilon
   'a-zA-Z': Use [a-zA-Z] as the alphabet, 1 as epsilon
   'ab': Use 'ab' as the alphabet, 1 as epsilon

  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.

Report bugs to <vaucanson-bugs@lrde.epita.fr>.
```

The whole list of supported commands is available via '`--list-commands`'.

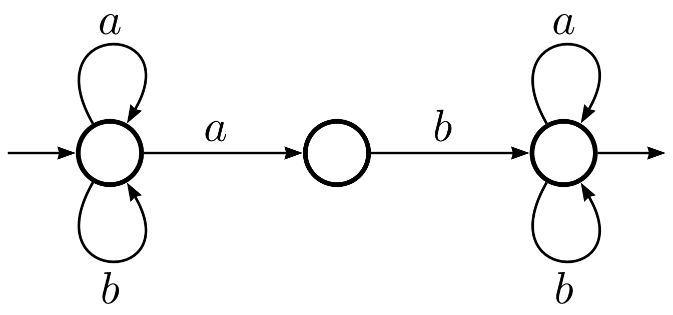
2.1.2 A first example

Vaucanson provides a set of common automata. The function `list-automata` lists them all:

```
# vcsn-b list-automata
The following automata are predefined:
  - a1
  - b1
  - div3base2
  - double-3-1
  - ladybird-6
```

Let's consider the Boolean automaton $\mathcal{A}_1$ (Figure 2.1), part of the standard library. It can be dumped using `dump-automaton`:

```
# vcsn-b dump-automaton a1
<automaton name="a1" xmlns="http://vaucanson.lrde.epita.fr">
  <labelType>
    <monoid generators="letters" type="free">
      <generator value="a"/>
      <generator value="b"/>
    </monoid>
    <semiring operations="numerical" set="B"/>
```



The graphical layout of this automaton was described by hand, using the Vaucanson-G LaTeX package. However, the following figures are generated by TAF-Kit, giving a very nice layout, yet slightly less artistic.

The automaton is taken from **?**, Fig. I.1.1, p. 58.

Figure 2.1: The automaton $\mathcal{A}_1$

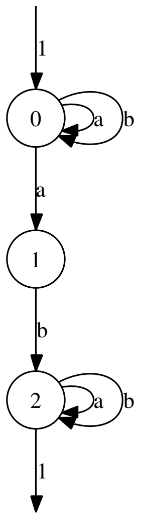
```
  </labelType>
  <content>
    <states>
      <state name="s0"/>
      <state name="s1"/>
      <state name="s2"/>
    </states>
    <transitions>
      <transition src="s0" dst="s0" label="a"/>
      <transition src="s0" dst="s0" label="b"/>
      <transition src="s0" dst="s1" label="a"/>
      <transition src="s1" dst="s2" label="b"/>
      <transition src="s2" dst="s2" label="a"/>
      <transition src="s2" dst="s2" label="b"/>
      <initial state="s0"/>
      <final state="s2"/>
    </transitions>
  </content>
</automaton>
```

Usual shell indirections (‘`|`’, ‘`>`’, and ‘`<`’) can be used to combine TAF-Kit commands. For instance, this is an easy means to bring a local copy of this file:

```
# vcsn-b dump-automaton a1 >a1.xml
```

TAF-Kit uses XML to exchange automata, to get graphical rendering of the automaton, you may either invoke `dot-dump` and then use a Dot compliant program, or use `display` that does both.

```
# vcsn-b dot-dump a1.xml >a1.dot
```



A { 3 states, 6 transitions, #I = 1, #T = 1 }

Determinization of $\mathcal{A}_1$

To determinize a Boolean automaton, call the `determinize` function:

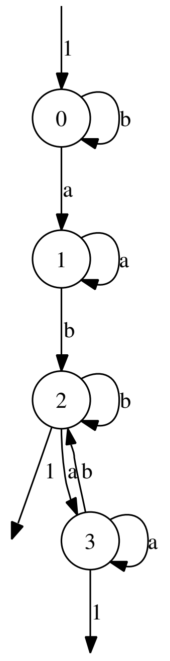
```
# vcsn-b dump-automaton a1 | vcsn-b determinize - >a1det.xml
```

To get information about an automaton, call the `info` function:

```
# vcsn-b info a1det.xml
States: 4
Transitions: 8
Initial states: 1
Final states: 2
```

Or use dotty to visualize it:

```
# vcsn-b dot-dump a1det.xml >a1det.dot
```

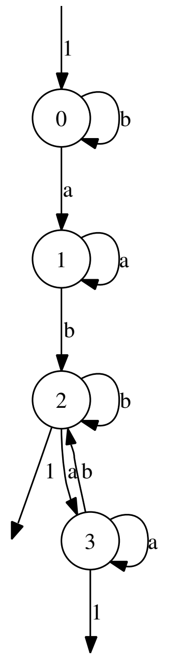

A { 4 states, 8 transitions, #I = 1, #T = 2 }

Minimizing

The minimal automaton can be computed the same way:

```
# vcsn-b minimize a1det.xml >a1detmin.xml
```

```
# vcsn-b dot-dump a1det.xml >a1detmin.dot
```


A { 4 states, 8 transitions, #I = 1, #T = 2 }

The commands can be composed with pipes from the shell, using '-' to denote the standard input.

```
# vcsn_b determinize a1.xml | vcsn_b minimize - > a1_min.xml
```

Evaluation

To evaluate whether a word is accepted:

```
# vcsn-b eval a1.xml 'abab'
1

# vcsn-b eval a1.xml 'bbba'
0
```

where 1 (resp. 0) means that the word is accepted (resp. not accepted) by the automaton.

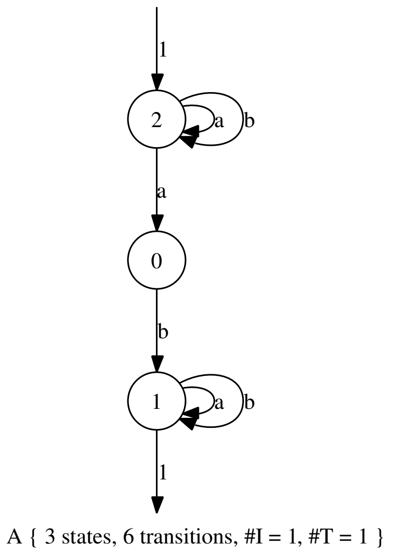
2.1.3 Rational expressions and Boolean automata

Vaucanson provides functions to manipulate rational expressions associated to Boolean automata. For instance, computing the language recognized by a Boolean automaton can be done using `aut-to-exp`:

```
# vcsn-b aut-to-exp a1.xml
(a+b)*.a.b.(a+b)*

# vcsn-b aut-to-exp a1det.xml
b*.a.a*.b.(a.a*.b+b)*.(a.a*+1)
```

Vaucanson provides several algorithms that build an automaton that recognizes a given language. The following sequence computes the minimal automaton of '`(a+b)*ab(a+b)*`'.

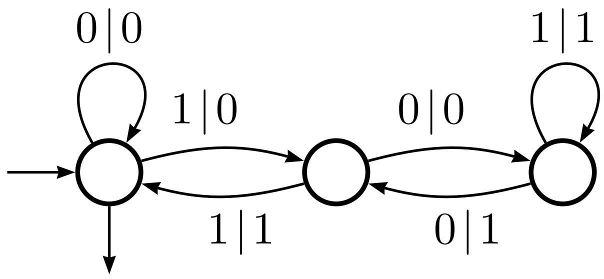
```
# vcsn-b --alphabet=ab standard "(a+b)*a.b.(a+b)*" | vcsn-b minimize - >l1.xml

# vcsn-b dot-dump l1.xml >l1.dot
```



2.1.4 Available functions

The whole list of supported commands is available via '`--list-commands`':

```
# vcsn-b --list-commands
List of available commands:
 * Input/output work:
   - define-automaton file: Define an automaton from scratch.
   - display aut: Display ‘aut’.
   - dot-dump aut: Dump dot output of ‘aut’.
   - dump-automaton file: Dump a predefined automaton.
   - edit-automaton file: Edit an existing automaton.
   - identity aut: Return ‘aut’.
   - info aut: Print useful infos about ‘aut’.
   - list-automata: List predefined automata.
 * Tests and evaluation on automata:
   - are-isomorphic aut1 aut2: Return whether ‘aut1’ and ‘aut2’ are isomorphic.
   - eval aut word: Evaluate ‘word’ on ‘aut’.
   - is-ambiguous aut: Return whether ‘aut’ is ambiguous.
   - is-complete aut: Return whether ‘aut’ is complete.
   - is-deterministic aut: Return whether ‘aut’ is deterministic.
   - is-empty aut: Return whether trimed ‘aut’ is empty.
   - is-realtime aut: Return whether ‘aut’ is realtime.
 * Generic algorithms for automata:
   - accessible aut: Give the maximal accessible subautomaton of ‘aut’.
   - eps-removal aut: Give ‘aut’ closed over epsilon transitions.
   - co-accessible aut: Give the maximal coaccessible subautomaton of ‘aut’.
   - complete aut: Give the complete version of ‘aut’.
   - concatenate aut1 aut2: Concatenate ‘aut1’ and ‘aut2’.
   - power aut n: Give the power of ‘aut’ by ‘n’.
   - product aut1 aut2: Give the product of ‘aut1’ by ‘aut2’.
   - quotient aut: Give the quotient of ‘aut’.
   - realtime aut: Give the realtime version of ‘aut’.
   - sum aut1 aut2: Give the sum of ‘aut1’ and ‘aut2’.
   - transpose aut: Transpose the automaton ‘aut’.
   - trim aut: Trim the automaton ‘aut’.
 * Boolean automaton specific algorithms:
   - complement aut: Complement ‘aut’.
   - determinize aut: Give the determinized automaton of ‘aut’.
   - minimize aut: Give the minimized of ‘aut’ (Hopcroft algorithm).
   - minimize-moore aut: Give the minimized of ‘aut’ (Moore algorithm).
 * Conversion between automata and expressions:
   - aut-to-exp aut: Give the automaton associated to ‘aut’.
   - derived-term exp: Use derivative to compute the automaton of ‘exp’.
   - exp-to-aut exp: Alias of ‘stardard’.
   - expand exp: Expand ‘exp’.
   - standard exp: Give the standard automaton of ‘exp’.
   - thompson exp: Give the Thompson automaton of ‘exp’.
```



The transducer computing the quotient by 3 of a binary number.

Figure 2.2: Rational-weight transducer $\mathcal{T}_1$

2.2 Transducers

While the VAUCANSON library supports two views of transducers, currently TAF-KIT only provides one view:

vcsn-tdc considering a transducer as a weighted automaton of a product of free monoid,

In a forthcoming release, TAF-KIT will provide:

vcsn-rw-tdc considering a transducer as a machine that takes a word as input and produce another word as (two-tape automata).

Both views are equivalent and VAUCANSON provides algorithms to pass from a view to the other one.

2.2.1 Example

To experiment with transducers, we will use $\mathcal{T}_1$, described in Figure 2.2, and part of the automaton library (**??**).

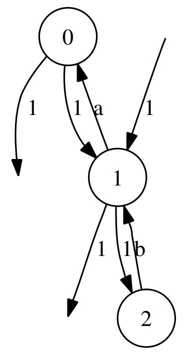
Domain

The transducer T only accepts binary numbers divisible by 3.

```
# vcsn-tdc dump-automaton t1 | vcsn-tdc --alphabet1=ab domain - >div-by-3.xml
```

Now the file '`divisible-by-3.xml`' contains the description of a Boolean automaton that accepts only the numbers divisible by 3:

```
# vcsn-b dot-dump div-by-3.xml >div-by-3.dot
```

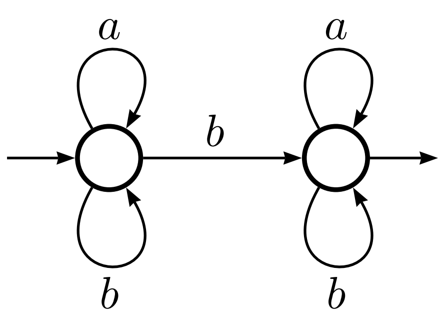


A { 3 states, 4 transitions, #I = 1, #T = 2 }

2.2.2 Available functions

The following functions are available for both `vcsn-rw-tdc` and `vcsn-tdc` programs. To invoke them, run '*`program algorithm-name`* `[`*`arguments`*`]`'.

```
# vcsn-tdc --list-commands
List of available commands:
 * Input/output work:
   - define-automaton file: Define an automaton from scratch.
   - display aut: Display ‘aut’.
   - dot-dump aut: Dump dot output of ‘aut’.
   - dump-automaton file: Dump a predefined automaton.
   - edit-automaton file: Edit an existing automaton.
   - identity aut: Return ‘aut’.
   - info aut: Print useful infos about ‘aut’.
   - list-automata: List predefined automata.
 * Tests and evaluation on transducers:
   - are-isomorphic aut1 aut2: Test if ‘aut1’ and ‘aut2’ are isomorphic.
   - is-empty aut: Test if ‘aut’ realizes the empty relation.
   - is-sub-normalized aut: Test if ‘aut’ is sub-normalized.
 * Generic algorithm for transducers:
   - eps-removal aut: epsilon-removal algorithm.
   - domain aut: Give the automaton that accepts all inputs accepted by ‘aut’.
   - eval aut exp: Give the evaluation of ‘exp’ against ‘aut’.
   - eval-aut aut: Evaluate the language described by the Boolean automaton
        ‘aut2’ on the transducer ‘aut1’.
   - image aut: Give an automaton that accepts all output produced by ‘aut’.
   - trim aut: Trim transducer ‘aut’.
 * Algorithms for transducers:
   - sub-normalize aut: Give the sub-normalized transducer of ‘aut’.
   - composition-cover aut: Outsplitting.
   - composition-co-cover aut: Insplitting.
   - compose aut1 aut2: Compose ‘aut1’ and ‘aut2’, two (sub-)normalized
        transducers.
   - u-compose aut1 aut2: Compose ‘aut1’ and ‘aut2’, two Boolean transducers,
        preserve the number of path.
   - to-rt aut: Give the equivalent realtime transducer of ‘aut’.
   - intersection aut: Transform a Boolean automaton in a fmp transducer by
        creating, for each word, a pair containing twice this word.
```



Considered without weight, $\mathcal{B}_1$ accepts words with a '`b`'. With weights, it counts the number of '`b`'s. Taken from **?**, Fig. III.2.2, p. 434.

Figure 2.3: The automaton $\mathcal{B}_1$

2.3 $\mathbb{Z}$-Automata

This part shows the use of the program `vcsn-z`, but all comments should also stand for the programs `vcsn-z-min-plus` and `vcsn-z-max-plus`.

Again, we will toy with some of the automata provided by `vcsn-z`, see **??**.

2.3.1 Counting '`b`'s

Let's consider $\mathcal{B}_1$ (Figure 2.3), an $\mathbb{N}$-automaton, *i.e.* an automaton whose label's weights are in $\mathbb{N}$. This time the evaluation of the word `w` by the automaton $\mathcal{B}_1$ will produce a number, rather than simply accept or reject `w`. For instance let's evaluate '`abab`' and '`bbab`':

```
# vcsn-z dump-automaton b1 | vcsn-z eval - 'abbb'
3

# vcsn-z dump-automaton b1 | vcsn-z eval - 'abab'
2
```

Indeed, $\mathcal{B}_1$ counts the number of '`b`'s.

Power

Now let's consider the $\mathcal{B}_1^n$, where

$$\mathcal{B}_1^n = \prod_{i=1}^{n} \mathcal{B}_1, n > 0$$

This is implemented by the `power` function:

```
# vcsn-z dump-automaton b1 | vcsn-z power - 4 >b4.xml

# vcsn-z power b1.xml 4 > b4.xml
```

The file '`b4.xml`' now contains the automaton $\mathcal{B}_1^4$. Let's check that the evaluation of the words '`abab`' and '`bbab`' by $\mathcal{B}_1^4$ gives the fourth power of their evaluation by $\mathcal{B}_1$:

```
# vcsn-z eval b4.xml 'abbb'
81

# vcsn-z eval b4.xml 'abab'
16
```

Quotient

Successive products of an automaton create a lot of new states and transitions.

```
# vcsn-z dump-automaton b1 | vcsn-z info -
States: 2
Transitions: 5
Initial states: 1
Final states: 1

# vcsn-z info b4.xml
States: 16
Transitions: 97
Initial states: 1
Final states: 1
```

One way of reducing the size of our automaton is to use the `quotient` algorithm.

```
# vcsn-z quotient b4.xml | vcsn-z info -
States: 5
Transitions: 15
Initial states: 1
Final states: 1
```

2.3.2 Available functions

In this section you will find a brief definition of all functions for manipulating weighted automata. The following functions are available for both. They are called using `vcsn-z`, `vcsn-z-max-plus`, and `vcsn-z-min-plus` run as '*program algorithm-name* [*arguments*]'.

```
# vcsn-z --list-commands
List of available commands:
 * Input/output work:
   - define-automaton file: Define an automaton from scratch.
   - display aut: Display 'aut'.
   - dot-dump aut: Dump dot output of 'aut'.
   - dump-automaton file: Dump a predefined automaton.
   - edit-automaton file: Edit an existing automaton.
   - identity aut: Return 'aut'.
   - info aut: Print useful infos about 'aut'.
   - list-automata: List predefined automata.
 * Tests and evaluation on automata:
   - are-isomorphic aut1 aut2: Return whether 'aut1' and 'aut2' are isomorphic.
   - eval aut word: Evaluate 'word' on 'aut'.
   - is-ambiguous aut: Return whether 'aut' is ambiguous.
   - is-complete aut: Return whether 'aut' is complete.
   - is-empty aut: Return whether trimed 'aut' is empty.
   - is-realtime aut: Return whether 'aut' is realtime.
 * Generic algorithms for automata:
   - accessible aut: Give the maximal accessible subautomaton of 'aut'.
   - eps-removal aut: Give 'aut' closed over epsilon transitions.
   - co-accessible aut: Give the maximal coaccessible subautomaton of 'aut'.
   - complete aut: Give the complete version of 'aut'.
   - concatenate aut1 aut2: Concatenate 'aut1' and 'aut2'.
   - power aut n: Give the power of 'aut' by 'n'.
```

```
  - product aut1 aut2: Give the product of ‘aut1’ by ‘aut2’.
  - quotient aut: Give the quotient of ‘aut’.
  - realtime aut: Give the realtime version of ‘aut’.
  - sum aut1 aut2: Give the sum of ‘aut1’ and ‘aut2’.
  - transpose aut: Transpose the automaton ‘aut’.
  - trim aut: Trim the automaton ‘aut’.
* Conversion between automata and expressions:
  - aut-to-exp aut: Give the automaton associated to ‘aut’.
  - derived-term exp: Use derivative to compute the automaton of ‘exp’.
  - exp-to-aut exp: Alias of ‘stardard’.
  - expand exp: Expand ‘exp’.
  - standard exp: Give the standard automaton of ‘exp’.
  - thompson exp: Give the Thompson automaton of ‘exp’.
```

Chapter 3

Vaucanson as a library

To be written.

Chapter 4

Developer Guide

The chapter is work in progress. It is not meant for user of the VAUCANSON library, but to developer and contributor who wish to include code in VAUCANSON.

4.1 Contributing Code

4.1.1 Directory usage

The VAUCANSON package is organized as follows:

Directory	Usage
`doc`	Documentation.
`doc/css`	CSS style for Doxygen.
`doc/makefiles`	Sample Makefile to reduce compilation time in VAUCANSON.
`doc/manual`	User's (and developer's) manual.
`doc/share`	LRDE share repository.
`doc/xml`	XML proposal.
`include/vaucanson`	Library start point: defines classical entry points such as "boolean_automaton.hh".
`include/vaucanson/algebra/concept`	Algebra concepts, "Structure" part of an Element.
`include/vaucanson/algebra/implementation`	Implementations of algebraic Structures. Some specialized structures too.
`include/vaucanson/algorithms`	Algorithms.
`include/vaucanson/algorithms/internal`	Internal functions of algorithms.
`include/vaucanson/automata/concept`	Structure of an automaton.
`include/vaucanson/automata/implementation`	Its implementation.
`include/vaucanson/config`	Package configuration and system files.
`include/vaucanson/contexts`	Context headers.
`include/vaucanson/design_pattern`	Element design pattern implementation.
`include/vaucanson/misc`	Internal headers of the whole library.
`include/vaucanson/tools`	Tools such as dumper, bencher.
`include/vaucanson/xml`	XML implementation.

Directory	Usage
`argp`	Argp library for TAF-Kit.
`build-aux`	Where Autotools things go.
`data`	Misc data, like Vaucanson's XSD, Emacs files.
`data/b`	Generated Boolean automata.
`data/z`	Generated Automata over Z.
`debian`	Debian packaging.
`src/benchs`	Benches.
`src/demos`	Demos.
`src/tests`	Test suite.
`taf-kit`	Typed Automata Function Kit, binaries to use Vaucanson.
`taf-kit/tests`	Test suite using the TAF-Kit.
`vaucanswig`	Vaucanswig, a SWIG interface for Vaucanson.
`vcs`	Version Control System configuration (VCS Home Page).

4.1.2 Coding Style

Until this is written, please refer to Tiger's Coding Style.

Emacs users should use the indentation style of '`data/vaucanson.el`'.

Includes

Please, never use backward relative paths anywhere. There are very difficult to follow (because several such strings can designate the same spot), they make renaming and moving virtually impossible etc.

Relative paths to sub-directories are welcome, although in many situations they are not the best bet.

In **Makefiles**, please using absolute paths starting from '`$(top_srcdir)`'. Unfortunately, because Automake cannot grok includes with Make macros (except... '`$(top_srcdir)`'), we can't shorten these.

For **header inclusion**, stacking zillions of '`-I`' is not the best solution because

- you have to work to find what file is really included
- you are likely to find unexpected name collisions if two separate directories happens to have (legitimately) two different files share the same name
- etc.

So rather, stick to *hierarchies* of include files, and use qualified '`#include`'s. For instance, use '`-I $(top_srcdir)/include -I $(top_srcdir)/src/tests/include`' and '`#include <vaucanson/...>`' falls into the first one ('`$(top_srcdir)/include`' has all its content in '`vaucanson`'), and '`#include <tests/...>`' falls into the latter since '`$(top_srcdir)/src/tests/include`' has all its content in '`vaucanson`').

4.1.3 Use of macros

C preprocessor (`cpp`) is evil, but code duplication is even worse. Macros can be useful, as in the following example:

```
# define PARSER_SET_PROPERTY(prop)                          \
      if (parser->canSetFeature(XMLUni::prop, true))       \
        parser->setFeature(XMLUni::prop, true);

PARSER_SET_PROPERTY(fgDOMValidation);
PARSER_SET_PROPERTY(fgDOMNamespaces);
PARSER_SET_PROPERTY(fgDOMDatatypeNormalization);
PARSER_SET_PROPERTY(fgXercesSchema);
PARSER_SET_PROPERTY(fgXercesUseCachedGrammarInParse);
PARSER_SET_PROPERTY(fgXercesCacheGrammarFromParse);

# undef PARSER_SET_PROPERTY
```

but please, respect the following conventions.

- Use upper case names, unless they are part of the interface such as `for_all_transitions` and so forth.
- Make them live short lives, as above: undefine them as soon as they are no longer needed.
- Respect the nesting structure: if '`foo.hh`' defines a macro, undefine it there too, not in the included '`foo.hxx`'.
- Indent `cpp` directives. The initial dash should always be in the first column, but indent the spaces (one per indentation) between it and the directive. The above code snippet was included in an outer `#if`.
- Each header file ('`.hh`', '`.hxx`', ...) should start with a classic `cpp` guard of the form

  ```
  #ifndef FILE_HH
  # define FILE_HH
    ...
  #endif // !FILE_HH
  ```

  GCC has some optimizations on file parsing when this scheme is seen.
- We often rely on `grep` and tags to search things. Please don't clutter names with `cpp` evilness.

For instance, this is bad style:

```
#define VCSN_choose_semiring(Canarg, Nonarg, Typeret...)                    \
    template <class Self>                                                   \
    template <class T>                                                      \
    Typeret                                                                 \
    SemiringBase<Self>::Canarg ## choose_ ## Nonarg ## starable(             \
    SELECTOR(T)) const                                                      \
    {                                                                       \
      return op_ ## Canarg ## choose_ ## Nonarg ## starable(this->self(),   \
                                                             SELECT(T));    \
    };
    VCSN_choose_semiring(can_, non_, bool)
    VCSN_choose_semiring(,, Element<Self, T>)
    VCSN_choose_semiring(, non_, Element<Self, T>)
```

4.1.4 Variable Names

Using long variable names clutters the code, so please, don't name your variables and arguments like `automaton1` or `alphabet`. Structure members and functions should be descriptive though.

In order to keep the variable names reasonable in size, and understandable, there are variable name conventions: some families of identifiers are reserved for some types of entities. The conventions are listed below; developers must follow it, and users are encouraged to do it too. In the following list, '∗' stands for "nothing, or a number".

al∗, alpha∗, A∗ alphabets

a∗, aut∗ automata (`automaton_t`, etc.)

t∗, tr∗ transitions

p∗, q∗, r∗, s∗ states (`hstate_t`)

Some variables should be consistently used to refer to some "fixed" values.

monoid_identity The neutral for the monoid, the empty word.

```
monoid_elt_t monoid_identity = a.series().monoid().empty_;
```

null_series The null series, the 0, the identity for the sum.

```
series_set_elt_t null_series = a.series().zero_;
```

semiring_elt_zero The zero for the weights.

```
semiring_elt_t semiring_elt_zero = a.series().semiring().wzero_;
```

4.1.5 Commenting Code

Use Doxygen. Besides the usual interface description, the Doxygen documentation must include:

- references to the definitions of the algorithm, e.g., a reference to the "Éléments de la théorie des automates", or even an URL to a mailing-list archive.
- detailed description of the assumptions, or, if you wish, pre- and post-conditions.
- the name of the developer
- use the `@pre` and `@post` tags liberally.

Don't try to outsmart your tool, even though it does not use the words "param" and "arg" as we do, stick to *its* semantics (let alone to generate correct documentation without warnings). This is correct:

```
/**
 * Delete memory associated with a stream upon its destruction.
 *
 * @arg \c T  Type of the pointed element.
 *
 * @param ev  IO event.
 * @param io  Related stream.
 * @param idx Index in the internal extensible array of a pointer to delete.
 *
 * @see iomanip
 * @author Thomas Claveirole <thomas.claveirole@lrde.epita.fr>
 */
template <class T>
void
pword_delete(std::ios_base::event ev, std::ios_base &io, int idx);
```

while this is not:

```
/** ...
 * @param T   Type of the pointed element.
 *
 * @arg ev    IO event.
 * @arg io    Related stream.
 * @arg idx   Index in the internal extensible array of a pointer to delete.
 * ... */
```

4.1.6 Writing Algorithms

There is a number of requirement to be met before including an algorithms into the library:

Document the algorithm See Section 4.1.5.

Comment the code Especially if the code is a bit tricky, or smart, or avoids nasty pitfalls, it *must* be commented.

Bind the algorithm to TAF-Kit

Include tests See Section 4.1.7 for more details. Tests based on TAF-Kit are appreciated. Note that tests require test cases: to exercise an algorithm, not any automaton will do, try to find relevant samples. Again, ETA is a nice source of inspiration.

Complete the documentation The pre- and post-conditions should also be described here.

When submitting a patch, make it complete (i.e., including the aforementioned items), and provide a ChangeLog. See Le Guide du LRDE , section “La maintenance de projets” and especially “Écrire un ChangeLog” for more details.

Because Vaucanson uses Trac, ChangeLog entries should explicit refer to tickets (e.g., “Fixing issue #38: implement is_ambiguous”), and possible previous revisions (e.g., “Fix a bug introduced in [1224]”).

4.1.7 Writing Tests

4.1.8 Mailing Lists

Vaucanson comes with a set of mailing lists:

vaucanson@lrde.epita.fr General discussions, feature request etc.

vaucanson-bugs@lrde.epita.fr To report errors in code, documentation, web pages, etc.

vaucanson-patches@lrde.epita.fr To submitted patches on code, documentation, and so forth.

vaucanson-private@lrde.epita.fr To contact privately the Vaucanson team.

Please, bear in mind that there are these lists have many readers, therefore this is a WORM medium: Write Once, Read Many. As a consequence:

- Be complete.
  One should not strive to understand what you are referring to, so always include proper references: URLs, Ticket numbers *and summary*, etc.

- Be concise.
  Write short, spell checked, understandable sentences. Reread yourself, remove useless words, be proud of what you wrote. Show respect to the reader. Spare us useless messages.

- Be structured.
  Quick and dirty replies with accumulated layers of replies at the bottom of the message is not acceptable. The right ordering is not the one that is the quickest to write, but the easiest to read.

- Be attentive.
  Lists are not write-only: consider the feedback that is given with respect.

As an example of what’s not to be done, avoid answering to yourself to point out you made a spell mistake: we can see that, and that’s a waste of time to read another message for that. Also, there is no hurry, it would probably be better to wait a bit to have a complete, well thought out, message, rather than a thread of 4 messages completing, contradicting, each other. Finally, if you still need to fix your message, supersede it, or even cancel it.

4.2 Vaucanson I/O

January 2005

Here is some information about input and output of automata in Vaucanson.

4.2.1 Introduction

As usual, the structure of the data representing an automaton in a flat file is called the file format.

There are several input and output formats for Vaucanson automata. Obviously:

- input formats are those that can be read from, i.e. from which an automaton can be loaded.
- output formats are those that can be written to, i.e. to which an automaton can be dumped.

Given these definitions, here is the meat:

- Vaucanson supports Graphviz (dot) as an output format. Most kinds of automata can be dumped as dot-files. Through the library this format is simply called `dot`.
- Vaucanson supports XML as an input and output format. Most kinds of automata can be read and written to and from XML streams, which Vaucanson does by using the Xerces-C++ library. Through the library this format is simply called `xml`.
- Vaucanson supports the FSM toolkit I/O format as an input and output format. This allows for basic FSM interaction. Only certain kinds of weighted automata can be meaningfully input and output with this format. Through the library this format is simply called `fsm`.
- Vaucanson supports a simple informative textual format as an input and output format. Most kinds of automata can be read and written to and from this format. Through the library this format is simply called `simple`.

4.2.2 Dot format

This format provides an easy way to produce a graphical representation of an automaton.

Output using this format can be given as input to the Graphviz `dot` command, which can in turn produce graphical representations in Encapsulated PostScript, PNG, JPEG, and many others.

It uses Graphviz' "directed graph" subformat.

If you want to see what it looks like go to the `data/b` subdirectory, build the examples and run them with the "dot" argument.

For Graphviz users:

Each graph generated by Vaucanson can be named with a string that also prefixes each state name. If done so, several automata can be grouped in a single graph by simply concatenating the Vaucanson outputs.

4.2.3 XML format

This format is intended to be an all-purpose strongly typed input and output format for automata.

Using it requires:

- that the Xerces-C++ library is installed and ready to use by the C++ compiler that is used to compile Vaucanson.
- configuring Vaucanson to use XML.
- computer resources and time.

What you gain:

- support for the Greater and Better I/O format. See documentation in the `doc/xml` subdirectory for further information.

If you want to see what it looks like go to the `data/b` subdirectory, build the examples and run them with the `xml` argument.

4.2.4 FSM format

This format is intended to provide a basic level of compatibility with the FSM tool kit. (FIXME: references needed)

Like FSM, support for this format in Vaucanson is limited to deterministic automata. It probably does not work with transducers, either.

It is not meant to be used that much apart from performance comparison with FSM. Some code exists to simulate FSM, in `src/demos/utilities/fsm`.

If you want to see what it looks like go to the `data/b`, build the examples and run them with the `fsm` argument.

4.2.5 Simple format

Initially intended to be a quick and dirty debugging input and output format, this format actually proves to be a useful, compact and efficient textual representation of automata.

Advantages over XML:

- does not require additional 3rd party software,
- simple and efficient (designed to be read and written to streams with very low memory footprint and minimum complexity),
- less bytes in file,
- not strongely typed (can be dumped from one automaton type and loaded to another).

Drawbacks from XML:

- not strongely typed (one cannot know what automaton type to build by only looking at the raw data).
- currently does not (probably) support transducers.

If you want to see what it looks like go to the `data/b`, build the examples and run them with the `simple` argument.

4.2.6 Using input and output

The library provides an infrastructure for generic I/O, which (hopefully) will help supporting more formats in the future.

The basis for this infrastructure is the way a developer C++ using the library will use it:

```
#include <vaucanson/tools/io.hh>

/* to save an automaton */
output_stream << automaton_saver(automaton, converter, format)

/* to load an automaton */
input_stream >> automaton_loader(automaton, converter, format, merge_states)
```

Where:

`automaton` is the automaton undergoing input or output. Note that the object must already be constructed, even to be read into.

`converter` is a helper class that is able to convert automaton transitions to character strings and possibly vice-versa.

`format` is a helper class that is able to convert the automaton to (and possibly from) a character string, using the converter as an argument.

`merge_states` is an optional argument that should be omitted in most cases. For advanced users, it allows loading a single automaton from several different streams that share the same state set.

About converters

The `converter` argument is mandatory. There are several converter types already available in Vaucanson. See below.

An I/O converter is a function object with one or both of the following:

- an operation that takes an automaton, a transition label and converts the transition label to a character string (std::string). This is called the output conversion.
- an operation that takes an automaton, a character string and converts the character string to a transition label. This is called the input conversion.

Vaucanson already provides these converters:

`vcsn::io::string_out`, bundled with `io.hh`. Provides the output conversion only. Uses the C++ operator << to create a textual representation of transition labels. Should work with all label types.

`vcsn::io::usual_converter_exp`, defined in `tools/usual_io.hh`. Provides both input and output conversions. Uses the C++ operator << to create a textual representation of transition labels, but requires also that algebra::parse can read back that representation into a variable of the same type. It is mostly used for generalized automata where transitions are labeled by rational expressions, hence the name.

`vcsn::io::usual_converter_poly<ExpType>`, defined in `tools/usual_io.hh`. Provides both input and output conversions. Converts transition labels to and from ExpType before (after) doing I/O. The implementation is meant to be used when labels are polynoms, and using the generalized (expression) type as ExpType.

Notes about XML and converters When the XML I/O format was implemented, the initial converter system was not used. Instead a specific converter system was re-designed specifically for this format.

(FIXME: explain why!)

(FIXME: why hasn't the generic converter for XML been ported back to fsm and simple formats?)

Because of this, when using XML I/O the "converter" argument is completely ignored by the format processor. Usually you can see `vcsn::io::string_output` mentioned.

(FIXME: this is terrible! it must be patched to use an empty vcsn::io::xml_converter_placeholder or something like it).

About formats

The `format` argument is mandatory. It specifies an instance of the object in charge of the actual input or output.

A format object is a function object that provides one or both the following operations:

- an operation that takes an output stream, the caller `automaton_saver` object, and the `converter` object. This is called the output operation.
- an operation that takes an input stream and the caller `automaton_loader` object. This is called the input operation. Note that this operation does not uses the `converter` object, because it should call back the `automaton_loader` object to actually perform string to transition label conversions.

Format objects may require arguments to be constructed, such as the title of the automaton in the output.

Format objects for a format should be defined in a `tools/xxx_format.hh` file.

Vaucanson provides the following format objects:

`vscn::io::dot(const std::string& digraph_title)`, in `tools/dot_format.hh`. Provides an output operation for the Graphviz `dot` subformat. The title provided when buildint the `dot` object in Vaucanson becomes the title of the graph in the output data and a prefix for state names. Therefore the title must contain only alphanumeric characters or the underscore (_), and no spaces.

`vcsn::io::simple()`, in `tools/simple_format.hh`. Provides both input and output operations for a simple text format.

`vcsn::xml::XML(const std::string& xml_title)`, in `xml/XML.hh`. Provides both input and output operations for the Vaucanson XML I/O format.

(FIXME: why not tools/xml_format.hh with proper includes of headers in xml/?)

(FIXME: really the FSM format should have a format object too.)

4.2.7 Examples

Create a simple dot output for an automaton a1:

```
std::ofstream fout("output.dot");
fout << automaton_saver(a1, vcsn::io::string_output(), vcsn::io::dot("a1"));
fout.close()
```

Output automaton a1 to XML, read it back into another automaton a2 (possibly of another type):

```
std::ofstream fout("file.xml");
fout << automaton_saver(a1, NULL, vcsn::xml::XML());
fout.close()

std::ifstream fin("file.xml");
fin >> automaton_loader(a2, NULL, vcsn::xml::XML());
fin.close()
```

Do the same, but this time using the simple format. The automata are generalized, i.e. labeled by expressions:

```
std::ofstream fout("file.txt");
fout << automaton_saver(a1, vcsn::io::usual_converter_exp(), vcsn::io::simple());
fout.close()
```

```
std::ifstream fin("file.txt");
fin >> automaton_loader(a2, vcsn::io::usual_converter_exp(), vcsn::io::simple());
fin.close()
```

4.2.8 Internal scenario

What happens in Vaucanson when you write:

```
fin >> automaton_loader(a1, c1, f1)
```

?

1. function `automaton_loader` creates an object AL1 of type `automaton_loader_` that memorizes its arguments.
2. `automaton_loader()` returns AL1.
3. `operator>>(fin, AL1)` is called.
4. `operator>>` says to format object f1: "hi, please use fin to load something with AL1".
5. f1 scans input stream fin. Things may happen then:
   - f1 finds a state numbered N. Then it says to AL1: "hey, make a new state into the output automaton, keep its handler s1 for yourself and remember it is associated to N". (callback `AL1.add_state`)
   - f1 finds a transition from state numbered N to state P, labeled with character string S. Then it says to AL1: "hey, create a transition with N, P, and S." (callback `AL1.add_transition`). Then:
     - AL1 remembers handler for state N (s1)
     - AL1 remembers handler for state P (s2)
     - AL1 says to converter c1: "hey, make me a transition label from S"
     - AL1 creates transition from s1 to s2 using converted label into output automaton.
6. When f1 is finished, it returns control to `operator>>` and then calling code.

   Of course since everything is statically compiled using templates there is no performance drawback due to the intensive use of callbacks.

4.2.9 Convenience utilities

For most formats the (relatively) tedious following piece of code:

```
output_stream << automaton_saver(a, CONVERTER(), FORMAT(...))
```

is also available as:

```
FORMAT_dump(output_stream, a, ...)
```

If available, this convenience utility is defined in `tools/XXX_dump.hh`.

Conversely, the following piece of code:

```
input_stream >> automaton_loader(a, CONVERTER(), FORMAT(...))
```

is usually also available as:

```
FORMAT_load(input_stream, a, ...)
```

If available, this convenience utility is defined in `tools/XXX_load.hh`.

(FIXME: move fsm_load away from fsm_dump.hh!)

As of today (2006-03-17) the FSM format is only available using the fsm_load() and fsm_dump() interface.