The Vaucanson TAF-Kit 1.2.95 Manual

The Vaucanson Group

2009-01-22

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-1.2.95
```

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-char-b`, allows to compute with classical automata and is described in Section 2.1. A variant of this program called `vcsn-int-b` handles Boolean automata whose letters are integers.

Section 2.2 describes the program `vcsn-char-fmp-b` 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. A variant of this program called `vcsn-int-fmp-b` handles transducers whose letters are integers.

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 $\mathbb{Z}$. A variant of this program called `vcsn-int-z` is specialized to handle $\mathbb{Z}$-automata whose letters are integers.

It is planned that a forthcoming version will include also:

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

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

vcsn-char-rw 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-char-b`, the TAF-KIT component dedicated to Boolean automata.

2.1.1 First Contacts

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

```
# vcsn-char-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`, require additional arguments, such as the word to evaluate. Some others, 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-char-b determinize-nodep a1.xml > a1det.xml
```

and should be understood, or analyzed, as follows.

1. `vcsn-char-b` is the call to a `shell` command that will launch a VAUCANSON function. `vcsn-char-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 Section A.1.1 indeed — in an XML format that is understood[1] by VAUCANSON. 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 allows to define automata and thus to produce the corresponding XML files (cf. below).

4. '>a1det.xml' puts the result of `determinize` into the file 'a1det.xml', that is, the XML file which describes the determinized automaton of $\mathcal{A}_1$.

As a more elaborate example, consider the following command

```
# vcsn-char-b identity a1.xml | vcsn-char-b determinize - | vcsn-char-b minimize - | vcsn-char-b
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 means of communication between various programs dealing with automata. And then it has been used as a communication tool between the invocations 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 in Appendix C of the Appendix.

```
# vcsn-char-b identity a1.xml | vcsn-char-b info -
States: 3
Transitions: 6
Initial states: 1
Final states: 1
```

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

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

  -a, --alphabet=ALPHABET    Set the alphabet for rational expressions or
                             automata
  -B, --bench=NB_ITERATIONS  Bench
  -D, --export-time-dot[=VERBOSE_DEGREE]
                             Export time statistics in DOT format
  -i, --input-type=INPUT_TYPE   Automaton input type (FSM or XML)
  -l, --list-commands        List the commands handled by the program
  -o, --output-type=OUTPUT_TYPE   Automaton output type (FSM, XML or DOT)
  -O, --bench-plot-output=OUTPUT_FILENAME
                             Bench output filename
  -p, --parser=OPTIONS       Set the parsing options for rational expressions
  -T, --report-time[=VERBOSE_DEGREE]
                             Report time statistics
  -v, --verbose              Be more verbose (print boolean results)
  -X, --export-time-xml      Export time statistics in XML format

 The following alphabets are predefined:
        'letters': Use [a-z] as the alphabet and '1' as empty word
        'alpha': Use [a-zA-Z] as the alphabet and '1' as empty word
        'digits': Use [0-9] as the alphabet and 'e' as empty word
        'ascii': Use all ascii characters as the alphabet and '_e' as empty word


  -?, --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-char-b list-automata
The following automata are predefined:
  - a1.xml
  - b1.xml
```

– div3base2.xml
– double-3-1.xml
– ladybird-6.xml
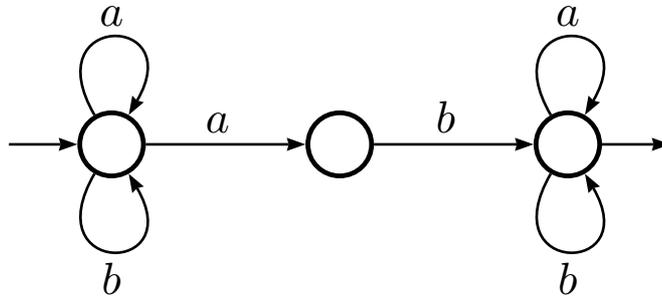
The graphical layout of this automaton was described by hand, using the Vaucanson-G LATEX package. However, the following figures are generated by TAF-KIT, giving a very nice layout, yet slightly less artistic. The automaton is taken from **?**, Fig. I.1.1, p. 58

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

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

```
# vcsn-char-b identity a1.xml

<fsmxml xmlns="http://vaucanson.lrde.epita.fr" version="1.0">

  <automaton>
    <valueType>
      <writingData closePar=")" openPar="(" plusSym="+" spacesSym=" " starSym="*" timesSym="." we
      <semiring operations="classical" set="B" type="numerical"/>
      <monoid genDescrip="enum" genKind="simple" genSort="letters" type="free">
        <writingData identitySym="1"/>
        <monGen value="a"/>
        <monGen value="b"/>
      </monoid>
    </valueType>
    <automatonStruct>
      <states>
        <state id="s0"/>
        <state id="s1"/>
        <state id="s2"/>
      </states>
      <transitions>
        <transition source="s1" target="s2">
          <label>
            <monElmt>
              <monGen value="b"/>
            </monElmt>
          </label>
        </transition>
        <transition source="s2" target="s2">
```

```
            <label>
              <monElmt>
                <monGen value="b"/>
              </monElmt>
            </label>
          </transition>
          <transition source="s2" target="s2">
            <label>
              <monElmt>
                <monGen value="a"/>
              </monElmt>
            </label>
          </transition>
          <transition source="s0" target="s0">
            <label>
              <monElmt>
                <monGen value="b"/>
              </monElmt>
            </label>
          </transition>
          <transition source="s0" target="s1">
            <label>
              <monElmt>
                <monGen value="a"/>
              </monElmt>
            </label>
          </transition>
          <transition source="s0" target="s0">
            <label>
              <monElmt>
                <monGen value="a"/>
              </monElmt>
            </label>
          </transition>
          <initial state="s0"/>
          <final state="s2"/>
        </transitions>
      </automatonStruct>
    </automaton>

</fsmxml>
```

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-char-b identity a1.xml >my-a1.xml
```

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

```
# vcsn-char-b --output-type=dot identity a1.xml >a1.dot
```
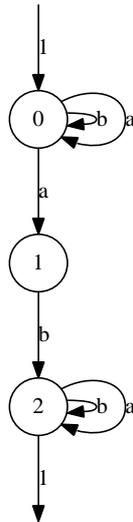
{ 3 states, 6 transitions, #I = 1, #T = 1 }

Determinization of $\mathcal{A}_1$

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

```
# vcsn-char-b identity a1.xml | vcsn-char-b determinize - >a1det.xml
```

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

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

Or use dotty to visualize it:

```
# vcsn-char-b --output-type=dot identity a1det.xml >a1det.dot
```
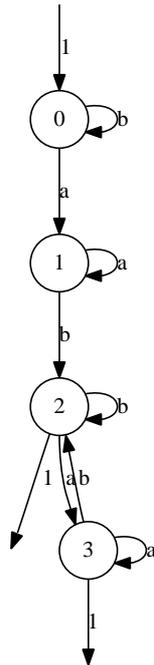
{ 4 states, 8 transitions, #I = 1, #T = 2 }

Evaluation

To evaluate whether a word is accepted:

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

# vcsn-char-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   Interactive Definition of Automata

TAF-KIT provides a text interface to define automata interactively, rather than having to deal
with XML files. Two functions are available:

define-automaton to build a fresh automaton from scratch,

edit-automaton to modify an existing automaton,

The interface is based on a menu of choices:

```
# vcsn-char-b --alphabet=ab define-automaton all.xml
Automaton description:
  States: (none)
  Initial states: (none)
  Final states: (none)

  Transitions: (none)

Please choose your action:
```

```
   1. Add states.
   2. Delete a state.

   3. Add a transition.
   4. Delete a transition.

   5. Set a state to be initial.
   6. Set a state not to be initial.

   7. Set a state to be final.
   8. Set a state not to be final.

   9. Display the automaton in Dotty.

   10. Exit.

Your choice [1-10]:
```

If you enter *1*, you will then be prompted for the number of states to add, say *1* again. The state 0 was created. To make it initial select *5*, and:

```
Your choice [1-10]: 5
  For state: 0
```

Likewise to make it final, using choice *7*. Finally, let's add a transition:

```
Your choice [1-10]: 3
  Add a transition from state: 0
  To state: 0
  Labeled by the expression: a+b
```

The automaton is generalized, that is to say, rational expressions are valid labels.

On top of the interactive menu, the current definition of the automaton is reported in a textual yet readable form:

```
Automaton description:
  States: 0
  Initial states: 0
  Final states: 0

  Transitions:
    1: From 0 to 0 labeled by ({1} a)+({1} b)
```

Interestingly enough, states are numbered from 0, but transitions numbers start at 1. Also, not that weights are reported, although only *1* is valid for Boolean automata.

Finally, hit *10* to save the resulting automaton in the file 'all.xml'.

2.1.4   Rational expressions and Boolean automata

VAUCANSON provides functions to manipulate rational expressions associated to Boolean automata. This provides an alternative means to create automata:

```
# vcsn-char-b --alphabet=ab exp-to-aut '(a+b)*' >all.xml

# vcsn-char-b --output-type=dot identity all.xml >all.dot
```
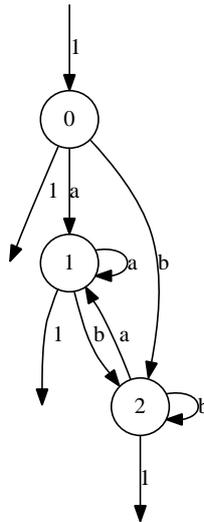
{ 3 states, 6 transitions, #I = 1, #T = 3 }

2.1.5   Token representations

When dealing with rational expressions in TAF-KIT, one may be willing to, for example, change the representation of the epsilon. More generally, the rational expressions parser understand 10 such tokens. They are:

- `OPAR`: the opening association parenthesis.

- `CPAR`: the closing association parenthesis.

- `PLUS`: the semi-ring additive law.

- `TIMES`: the monoid concatenation law.

- `STAR`: the Kleene star.

- `ONE`: the identity of the monoid.

- `ZERO`: the zero of the semi-ring.

- `OWEIGHT`: the opening weight brace.

- `CWEIGHT`: the closing weight brace.

- `SPACE`: a whitespace character.

Each token must be a non-empty string, with arbitrary length. Some checks will be done by TAF-KIT, to ensure tokens do not collide. You can also use the `ALPHABET` token an arbitrary number of times, to append letters to the current alphabet. The following commands:

```
# vcsn-char-b --alphabet=ab exp-to-aut '((a) + b)*' >parser.xml
```

```
# vcsn-char-b --output-type=dot identity parser.xml >parser.dot
```
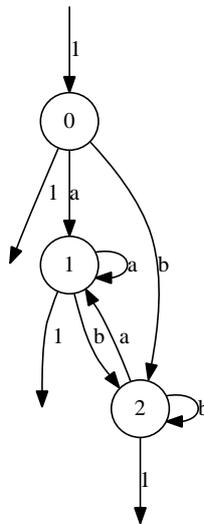
{ 3 states, 6 transitions, #I = 1, #T = 3 }

, will give the same results as:

```
# vcsn-char-b --alphabet=a --parser="ALPHABET=b ONE=e STAR=star" exp-to-aut '(a + b)star' >parser
```

```
# vcsn-char-b --output-type=dot identity parser2.xml >parser2.dot
```
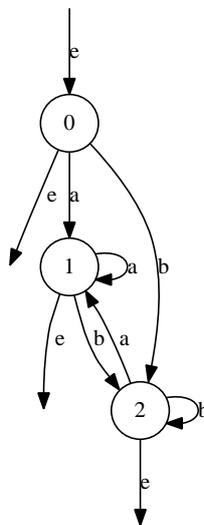

{ 3 states, 6 transitions, #I = 1, #T = 3 }

With the `ALPHABET` token, you can specify two types of letters:

- letters: one character is used to represent a letter.

- words: many characters are used to represent a letter.

For example, `ALPHABET=letters:abc` defines the alphabet with three letters $\{a, b, c\}$, while `ALPHABET=words:abc` defines the alphabet with only one letter $\{abc\}$. Letters are separated by commas, and each TAF-KIT context defaults to either letters (vcsn-char-b eg) or words (vcsn-int-b eg). `vcsn-int-b --alphabet=0,1,2,3,4,5,6,7,8,9` and `vcsn-int-b --alphabet=letters:0123456789'` will give the same alphabets.

Everywhere you want to use special characters used by the `--parser` option, prepending a backslash character will escape it. So, you can write: `ALPHABET=\,,\=` which defines the alphabet with the two letters `,` and `=`.

Finally whenever you may put a single character you may write a pair a words. The following construction is then valid: `ALPHABET=(\,,\))`, and defines the alphabet with only one letter: the pair whose first component is `,` and the second one is `)`.

For more details, the grammar of the `--parser` option is included in the source code in EBNF notations.

Minimizing

This automaton, constructed following the Thompson algorithm, is not the simplest one: it can be minimized:

```
# vcsn-char-b minimize all.xml >allmin.xml
```

```
# vcsn-char-b --output-type=dot identity allmin.xml >allmin.dot
```


{ 3 states, 6 transitions, #I = 1, #T = 3 }
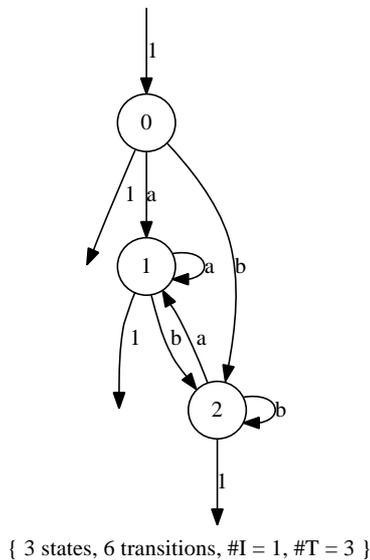
Computing the language recognized by a Boolean automaton can be done using `aut-to-exp`:

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

```
# vcsn-char-b aut-to-exp allmin.xml
(a.a*.b+b).(a.a*.b+b)*.(a.a*+1)+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-char-b --alphabet=ab standard "(a+b)*a.b.(a+b)*" | vcsn-char-b quotient - >l1.xml
```

```
# vcsn-char-b --output-type=dot identity l1.xml >l1.dot
```
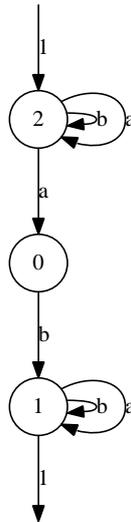
{ 3 states, 6 transitions, #I = 1, #T = 1 }

2.1.6 Available functions

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

```
# vcsn-char-b --list-commands
List of available commands:
 * Input/output work:
   - display aut: Display 'aut'.
   - edit-automaton file: Edit an automaton.
   - identity aut: Return 'aut'.
   - info aut: Print useful infos about 'aut'.
   - list-automata: List predefined automata.
 * Tests and evaluation on automata:
   - are-equivalent aut1 aut2: Do 'Aut1' and 'Aut2' recognize the same language?
   - 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 'aut' is empty.
   - has-succ-comp aut: Return whether 'aut' has successful computations (trimmed 'aut' is not en
   - is-realtime aut: Return whether 'aut' is realtime.
   - is-standard aut: Return whether 'aut' is standard.
 * Generic algorithms for automata:
   - accessible aut: Give the maximal accessible subautomaton of 'aut'.
   - eps-removal aut: Give 'aut' closed over epsilon transitions.
   - eps-removal-sp 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'.
   - standardize aut: Give the standard automaton of 'aut'.
   - union-of-standard aut1 aut2: Give the union of standard automata.
```

- concat-of-standard aut1 aut2: Give the concatenation of standard automata.
- star-of-standard aut: Give the star of automaton 'aut'.
- union aut1 aut2: Give the union 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 'standard'.
  - expand exp: Expand 'exp'.
  - identity-exp exp: Output 'exp' after trivial simplifications.
  - realtime-exp exp: Make 'exp' realtime.
  - standard exp: Give the standard automaton of 'exp'.
  - thompson exp: Give the Thompson automaton of 'exp'.
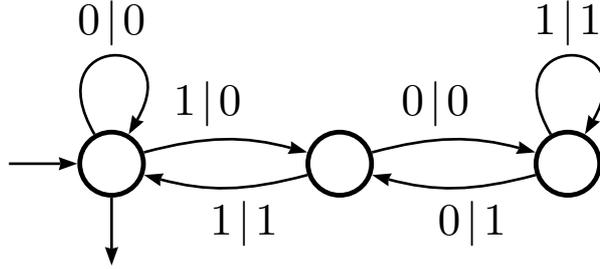
The transducer computing the quotient by 3 of a binary number.

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-char-fmp-b considering a transducer as a weighted automaton of a product of free monoid,

In a forthcoming release, TAF-KIT will provide:

vcsn-char-rw 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 (Section A.3.2).

Domain

The transducer T only accepts binary numbers divisible by 3.

```
# vcsn-char-fmp-b identity t1.xml | vcsn-char-fmp-b --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-char-b --output-type=dot identity div-by-3.xml >div-by-3.dot
```
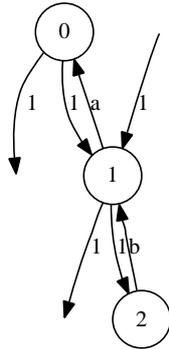
{ 3 states, 4 transitions, #I = 1, #T = 2 }

2.2.2 Available functions

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

```
# vcsn-char-fmp-b --list-commands
List of available commands:
 * Input/output work:
   - display aut: Display 'aut'.
   - edit-automaton file: Edit an automaton.
   - identity aut: Return 'aut'.
   - info aut: Print useful infos about 'aut'.
   - list-automata: List predefined automata.
 * Tests and evaluation on transducers:
   - is-empty aut: Return whether 'aut' is empty.
   - has-succ-comp aut: Return whether 'aut' has successful computations (trimmed 'aut' is not em
   - is-sub-normalized aut: Test if 'aut' is sub-normalized.
   - is-ltl aut: Test if 'aut' is letter-to-letter.
 * Generic algorithm for transducers:
   - eps-removal aut: epsilon-removal algorithm.
   - eps-removal-sp 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 aut1 aut2: Evaluate the language described by the
        Boolean automaton 'aut2' on the transducer 'aut1'.
   - ltl-to-pair aut: Give an automaton defined over a pair letter alphabet same as 'aut'.
   - image aut: Give an automaton that accepts all output produced by 'aut'.
   - transpose aut: Give the transposed of the transducer '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 transducers,
        preserve the number of paths.
   - to-rw aut: Give the equivalent rational weight transducer of 'aut'.
   - invert aut: Give the inverse of 'aut'.
   - intersection aut: Transform a Boolean automaton into an fmp transducer by
        creating, for each word, a pair containing twice this word.
```
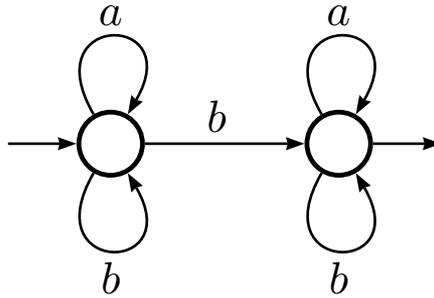
Considered without weight, $\mathcal{B}_1$ accepts words with a 'b'. With weights, it counts the number of 'b's. Taken from Sakarovitch (2003, Fig. III.2.2, p. 434).

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

2.3 $\mathbb{Z}$-Automata

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

Again, we will toy with some of the automata provided by `vcsn-char-z`, see Section A.2.

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-char-z identity b1.xml | vcsn-char-z eval - 'abbb'
3

# vcsn-char-z identity b1.xml | vcsn-char-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-char-z identity b1.xml | vcsn-char-z power - 4 >b4.xml

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

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

Quotient

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

```
# vcsn-char-z identity b1.xml | vcsn-char-z info -
States: 2
Transitions: 5
Initial states: 1
Final states: 1

# vcsn-char-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-char-z quotient b4.xml | vcsn-char-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-char-z, vcsn-char-z-max-plus, and vcsn-char-z-min-plus run as '*program algorithm-name [arguments]*'.

```
# vcsn-char-z --list-commands
List of available commands:
 * Input/output work:
   - display aut: Display 'aut'.
   - edit-automaton file: Edit an automaton.
   - identity aut: Return 'aut'.
   - info aut: Print useful infos about 'aut'.
   - list-automata: List predefined automata.
 * Tests and evaluation on automata:
   - 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 'aut' is empty.
   - has-succ-comp aut: Return whether 'aut' has successful computations (trimmed 'aut' is not en
   - is-realtime aut: Return whether 'aut' is realtime.
   - is-standard aut: Return whether 'aut' is standard.
 * Generic algorithms for automata:
   - accessible aut: Give the maximal accessible subautomaton of 'aut'.
   - eps-removal aut: Give 'aut' closed over epsilon transitions.
   - eps-removal-sp 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'.
- standardize aut: Give the standard automaton of 'aut'.
- union-of-standard aut1 aut2: Give the union of standard automata.
- concat-of-standard aut1 aut2: Give the concatenation of standard automata.
- star-of-standard aut: Give the star of automaton 'aut'.
- union aut1 aut2: Give the union 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 'standard'.
  - expand exp: Expand 'exp'.
  - identity-exp exp: Output 'exp' after trivial simplifications.
  - realtime-exp exp: Make 'exp' realtime.
  - standard exp: Give the standard automaton of 'exp'.
  - thompson exp: Give the Thompson automaton of 'exp'.

Chapter 3

Vaucanswig

3.1 Introduction to Vaucanswig

Vaucanswig is a set of SWIG definitions which allow to use VAUCANSON in a high-level dynamic language such as Python, Perl, PHP or Ruby.

3.1.1 Introduction

VAUCANSON is a C++ library that uses static genericity.

SWIG is an interface generator for C and C++ libraries which enables their use from a variety of languages, including CHICKEN, C#, Scheme, Java, O'Caml, Perl, Pike, PHP, Python, Ruby, Lua, Lisp, and TCL.

Unfortunately, running SWIG directly on the VAUCANSON library does not work: most of VAUCANSON features are expressed using C++ meta-code, which means that basically there is no real code in VAUCANSON for SWIG to work on.

Vaucanswig comes between SWIG and VAUCANSON: it describes to SWIG some explicit VAU-CANSON types and algorithms implementations so that SWIG can generate the inter-language interface.

3.1.2 Usage

For any SWIG-supported language, using Vaucanswig requires the following steps:

1. Generate the language interface from SWIG input sources (`.i` files) provided by Vaucanswig,

2. Compile the interface into extensions to the language library (e.g. dynamically-loadable shared package modules for Python).

3. Load the extensions into the target language environment.

Vaucanswig does not help with the latter two steps, except for the Python language target (see below). Refer to the SWIG documentation for information about generating language extensions from SWIG input files for other languages.

3.1.3 What is provided?

Glossary

In the following sections, the term "category" refers to the set of features related to a particular algebraic configuration in VAUCANSON.

The following categories are predefined in Vaucanswig:

Category	Semiring values	Monoid values	Series	Series values	Expression values
usual	bool	string	$\mathbb{B}\langle\!\langle A^*\rangle\!\rangle$	polynom	exp
numerical	int	string	$\mathbb{Z}\langle\!\langle A^*\rangle\!\rangle$	polynom	exp
tropical_min	int	string	$\mathbb{Z}(\min,+)\langle\!\langle A^*\rangle\!\rangle$	polynom	exp
tropical_max	int	string	$\mathbb{Z}(\max,+)\langle\!\langle A^*\rangle\!\rangle$	polynom	exp

These are the standard contexts defined in Vaucanson. They are defined in Vaucanswig in the file `expand.sh`.

What is in a category?

For a given category D, Vaucanswig defines the following **modules**:

vaucanswig_D_context Algebra and algebraic context.

vaucanswig_D_automaton Automata types (standard and generalized).

vaucanswig_D_alg_... Algorithm wrappers.

vaucanswig_D_algorithms General wrapper for all algorithms.

Each of these modules becomes an extension package/module/namespace in the target language.

Algebra

For a given category D, the module **vaucanswig_D_context** contains the following **classes**:

D_alphabet_t Alphabet element with constructor from a string of generator letters:

> **(constructor)** string → D_alphabet_t

D_monoid_t Monoid structural element with the following members:

- standard Vaucanson constructors and operators,
- method to construct a word element from a simple string:
  > **make** string → D_monoid_elt_t
- method to generate the identity value:
  > **identity** → D_monoid_elt_t

D_monoid_elt_t Word (monoid element) with standard Vaucanson constructors and operators.

D_semiring_t Semiring structural element with the following members:

- standard Vaucanson constructors and operators,
- method to construct a weight element from a number:
  > **make** int → D_semiring_elt_t
- methods to generate the identity and zero values:
  > **identity** → D_semiring_elt_t
  > **zero** → D_semiring_elt_t

D_semiring_elt_t Weight (semiring element) with standard Vaucanson constructors and operators.

D_series_set_t Series structural element with the following members:

- standard VAUCANSON constructors and operators,
- methods to construct a series element from a number or string:

  make int → D_series_set_elt_t

  make string → D_series_set_elt_t
- methods to generate the identity and zero values as polynoms or expressions:

  identity → D_series_set_elt_t

  zero → D_series_set_elt_t

  exp_identity → D_exp_t

  exp_zero → D_exp_t

D_series_set_elt_t, D_exp_t Polynom and expressions (series elements with polynom and expression implementations) with standard VAUCANSON constructors and operators.

D_automata_set_t Structural element for automata. Include standard VAUCANSON constructors.

D_context Convenience class with utility methods. It provides the following members:

- constructors

  (constructor) D_automata_set_t → D_context

  (copy constructor) D_context → D_context
- accessors for structural elements:

  automata_set → D_automata_set_t

  series → D_series_set_t

  monoid → D_monoid_t

  semiring → D_semiring_t

  alphabet → D_alphabet_t
- shortcut constructors for elements:

  semiring_elt int → D_semiring_elt_t

  word string → D_monoid_elt_t

  series int → D_series_set_elt_t

  series word → D_series_set_elt_t

  series D_exp_t → D_series_set_elt_t

  exp D_series_set_elt_t → D_exp_t

  exp string → D_expt_t

In addition to these classes, the module `vaucanswig_D_context` contains the following **function**:

make_context D_alphabet_t → D_context

Algebra usage

All classes are equipped with a `describe` method for textual representation of values. Example use (Python):

```
>>> from vaucanswig_usual_context import *
>>> c = make_context(usual_alphabet_t("abc"))

>>> c.exp("a+b+c").describe()
'usual_exp_t@0x81a2e60 = ((a+b)+c)'

>>> (c.exp("a")*c.exp("a+b+c")).star().describe()
```

26

```
'usual_exp_t@0x81a20f8 = (a.((a+b)+c))*'
>>> from vaucanswig_tropical_min_context import *
>>> c = make_context(tropical_min_alphabet_t("abc"))

>>> c.series().identity().describe()
'tropical_min_serie_t@0x81ad8b8 = 0'
>>> c.series().zero().describe()
'tropical_min_serie_t@0x81a6de8 = +oo'
```

Automata

For a given category D, the module `vaucanswig_D_automaton` contains the following **classes**:

D_auto_t The standard automaton type for this category.

gen_D_auto_t The generalized (with expression labels) automaton type for this category.

These class provides the following constructors:

(constructor) $D\_context \rightarrow D\_auto\_t$

(constructor) $D\_context \rightarrow gen\_D\_auto\_t$

(copy constructor) $D\_auto\_t \rightarrow D\_auto\_t$

(copy constructor) $gen\_D\_auto\_t \rightarrow gen\_D\_auto\_t$

(constructor) $D\_auto\_t \rightarrow gen\_D\_auto\_t$

For convenience purposes, a `gen_D_auto_t` instance can be constructed from a `D_auto_t` (generalization). The opposite is not possible, of course.

In addition to the standard VAUCANSON methods, these classes have been augmented with the following operators:

describe() Give a short description for the object.

save(filename) Save data to a file.

load(filename) Load data from a file. The automaton must be already defined (empty) and its structural element must be compatible with the file data.

dot_run(tmpf, cmd) Dump the automaton to file named `tmpf`, then run command `cmd` on file `tmpf`. The file is in dot format compatible with Graphviz.[1]

Example use:

```
>>> from vaucanswig_usual_automaton import *
>>> a = usual_auto_t(c)
>>> a.add_state()
0
>>> a.add_state()
1
>>> a.add_state()
2
>>> a.del_state(1)
>>> for i in a.states():
...     print i
```

_____

[1] http://www.research.att.com/sw/tools/graphviz/

```
   ...
   0
   2
15 >>> a.dot_run("tmp", "dot_view")

   >>> a.save("foo")
   >>> a2 = usual_auto_t(c)
   >>> a2.load("foo");
20 >>> a2.states().size()
   2
```

Algorithms

As a general rule of thumb, if some algorithm `foo` is defined in the source file `vaucanson/algorithms/bar.hh` then:

- the module `vaucanswig_D_alg_bar` contains a function `foo`,

- the module `vaucanswig_D_algorithms` contains `D.foo`.

3.1.4 Adding new algorithms

The Vaucanswig generator automatically build Vaucanswig modules from definitions found in the VAUCANSON source files.

You can add a new algorithm to vaucanswig simply by adding declarations of the form:

```
// INTERFACE:  ....
```

to the VAUCANSON headers.

Example

Let's consider the VAUCANSON header `foo.hh` in `include/vaucanson/algorithms`, which contains the following code:

```
// INTERFACE: Exp foo1(const Exp& other) { return vcsn::foo1(other); }
template<typename S, typename T>
Element<S, T> foo1(const Element<S, T>& exp);

5 // INTERFACE: Exp foo1(const Exp& other1, const Exp& other2) { return vcsn::foo2(other1, oth
template<typename S, typename T>
Element<S, T> foo1(const Element<S, T>& exp);
```

Then, after running `expand.sh` (the Vaucanswig generator) for category D, the module `vaucanswig_D_alg_foo` becomes available:

```
    foo1: D_exp_t $\to$ D_exp_t
    foo2: (D_exp_t, D_exp_t) $\to$ D_exp_t
```

In addition, the special algorithm class `D`, defined in `vaucanswig_D_algorithms`, also contains 'foo1' and 'foo2'.

Limitations

When writing `// INTERFACE:` comments, the following notes must be taken into consideration:

- The comment must stand on a single line. Indeed, `expand.sh` does not currently support multi-line interface declarations.

- The following special macro names are available:

  Exp The expression type for the category.

  Serie The polynom/serie type for the category.

  Automaton, GenAutomaton The automaton types for the category.

  HList A list of state or transition handlers (integers). This type is `std::list<int>` in C++ and a standard sequence of numbers in the target language.

  Word The word type for the category.

- When accessing automata, a special behavior stands. Instead of writing:

  ```
  // INTERFACE: void foo(Automaton& a) { return vcsn::foo(a); }
  // INTERFACE: void foo(GenAutomaton& a) { return vcsn::foo(a); }
  ```

  one should write instead:

  ```
  // INTERFACE: void foo(Automaton& a) { return vcsn::foo(*a); }
  // INTERFACE: void foo(GenAutomaton& a) { return vcsn::foo(*a); }
  ```

  Indeed, **Automaton** and **GenAutomaton** do not expand to VAUCANSON automata types, but to a wrapper type. The real automaton can be reached by means of operator*().

3.1.5 Python support

For convenience purposes, Python interfaces for Vaucanswig are included in the distribution. They are automatically compiled and installed with VAUCANSON if enabled. To enable these modules, run the `configure` script like this:

```
configure —enable−vaucanswig
```

3.2 Building language interfaces with Vaucanswig

This section describes how to use Vaucanswig to produce interfaces with other languages.

3.2.1 Background

Vaucanswig is a set of SWIG wrapper definitions for the VAUCANSON library.

SWIG takes Vaucanswig as input, and generates code to link between any supported scripting language and C++. In that sense, Vaucanswig is already "meta", because it ultimately supports several scripting languages. But still, even Vaucanswig itself is automatically generated, and this "meta-build" process is described in Section 3.3.

The document you are reading explains how to *use* Vaucanswig once it has been generated.

3.2.2 General idea

Once Vaucanswig has been generated, it is composed of input files to SWIG.

To use VAUCANSON in a target scripting language, two steps are necessary:

1. Produce C++ sources for the interface (running SWIG).
   This step only requires Vaucanswig sources and a decent version of SWIG.

2. Compile these sources.
   This step requires the VAUCANSON library and the extension libraries for the selected target language.

3.2.3 SWIG modules (`MODULES`)

Vaucanswig defines a number of SWIG modules.

The list of SWIG modules, hereinafter named `MODULES`, contains:

Name of module	Description
`core`	the core of vaucanswig.
`K_context`	for each K, the definition of the algebraic context K (K can be `usual`, `numerical`, `tropical` and so on)
`K_automaton`	definition of the Automaton and Expression types in context K
`K_alg_A`	for each algorithm A, the definition of the specific instance of A in context K. (A can be "complete", "standard", "product" and so on)
`K_algorithms`	a convienient wrapper for context K with "shortcuts" to all the algorithms instanciated for K.

Note that the name of SWIG modules are closely related to the namespace where the corresponding features can be found in the target scripting language.

Then, for each module M, two items are available:

Item	Description
'`src/vaucanswig_M.i`'	the dedicated SWIG source file
'`src/M.deps`'	(optional, may not exist) a file containing a list of modules that M is dependent upon. If the file is empty, two cases apply: • M is "core" - no dependency • M is not "core" - it depends on "core".

The first item is the most important. The second is only useful to create automated build processes which require dependency rules.

3.2.4 C++ sources specific to the target scripting language (T.S.L.)

Each TSL needs a different set of wrapper for the Vaucanswig modules.

For any given TSL, source files for the `MODULES` can be created by SWIG by running the following pseudo-algorithm:

```
$ for M in ${MODULES}; do
    ${SWIG} −noruntime −c++ −${TSL} \
      −I${VAUCANSWIGDIR}/src \
      −I${VAUCANSWIGDIR}/meta \
      −I${VAUCANSON_INCLUDES}    \
      ${VAUCANSWIGDIR}/src/vaucanswig_${M}.i
  done
```

Where:

- `${TSL}` is the SWIG option pertaining to the language (python, java ...)

- `${VAUCANSWIGDIR}` is the root directory of Vaucanswig.

- `${SWIG}` is the path to the SWIG binary.

- `${VAUCANSON_INCLUDES}` is the base directory of the VAUCANSON library.

3.2.5 Compilation of the binaries for the target scripting language

The previous step creates a bunch of C++ source files of the form 'vaucanswig_${M}_wrap.cxx'.
They should be compiled with the C++ compiler supported by the TSL.

The C++ compilation should use the following flags:

- '-DINTERNAL_CHECKS -DSTRICT -DEXCEPTION_TRAPS'
  Use for more secure code in VAUCANSON.

- '-I${VAUCANSON_INCLUDES}'
  Specify the location of the VAUCANSON library headers.

- '-I${VAUCANSWIGDIR}/src -I${VAUCANSWIGDIR}/meta'
  Needed by Vaucanswig.

In addition, any "compatibility" flags required by VAUCANSON for this particular C++ compiler should be used as well.

3.2.6 Automake support for Python as a TSL

According to the previous section, a 'Makefile.am' file is generated in the subdirectory 'python/'.
It contains four main parts:

A header

```
   ##
   ## Set INCLUDES for compilation of C++ code.
   ##

 5 # FIXME: the python path is hardcoded, this is NOT good.
   INCLUDES = -I/usr/include/python2.2 \
              -I$(srcdir)/../src -I$(srcdir)/../meta \
              -I$(top_srcdir)/include -I$(top_builddir)/include

10 ##
   ## Set AM_... flags.
   ##

   # According to spec.
15 AM_CPPFLAGS = -DINTERNAL_CHECKS -DSTRICT -DEXCEPTION_TRAPS
   # We want lots of debugging information in the wrapper code.
   AM_CXXFLAGS = $(CXXFLAGS_DEBUG)
   # For Libtool, to generate dynamically loadable modules.
   AM_LDFLAGS = -module -avoid-version
```

The list of binary targets (the shared objects - DLL)

```
# for each MODULE:
pyexec_LTLIBRARIES += libvs_$(MODULE).la
```

The list of Python source files

```
# for each MODULE:
python_PYTHON += vaucanswig_$(MODULE).py
```

Build specifications for binary targets

```
# for each MODULE:
libvs_$(MODULE)_la_SOURCES = vaucanswig_$(MODULE)_wrap.cxx

# If the module is "core":
#     # This should be the only dependency against static, non-template
#     # Vaucanswig code. And make it a dependency to the SWIG runtime.
#     libvs_core_la_LIBADD = ../meta/libvv.la -lswigpy

# Else:
#     If src/$(MODULE).deps is empty:
#             libvs_$(MODULE)_la_LIBADD = libvs_core.la
#     Else:
#         for each DEPENDENCY in src/$(MODULE).deps do:
#             libvs_$(MODULE)_la_LIBADD += libvs_$(DEPENDENCY).la
```

Additionnaly, the following (not important) parts are generated for convenience purposes:

- Rules to rerun SWIG in case something changes in Vaucanswig:

```
vaucanswig_*_wrap.cxx vaucanswig_*.py: ../src/vaucanswig_*.i
        $(SWIG) -noruntime -c++ -python -I... \
                    -o vaucanswig_*_wrap.cxx \
                    ../src/vaucanswig_*.i
```

- Installation and uninstallation hooks.

3.2.7 Automake support for the TSL-independent code

In order to make things comply to the spirit of the Autotools, a convenience 'Makefile.am' is generated in the 'src/' directory.

It contains a definition of EXTRA_DIST with all the SWIG module source files, of the form: 'vaucanswig_$(MODULE).i'.

3.3 Generating and extending Vaucanswig sources

The Section 3.2 describes how to use Vaucanswig to create a wrapper for VAUCANSON in a scripting language. (read it first)

This document instead describes how Vaucanswig itself is generated, currently using the infamous 'expand.sh' script.

3.3.1 The list of Vaucanswig modules

Once generated, Vaucanswig is a set of SWIG modules. This list of modules is algorithmically generated. The overall process to build the list of module names is as follows:

1. put `core` in the `MODULES` list.

2. create an auxiliary list `ALGS` of algorithm families.

   (detailed below, gives `alg_sum`, `alg_complete`, ...)

3. create an auxiliary list `KINDS` of algebra contexts

   (contains `boolean`, `z`, `z_max_plus`, ...)

4. extend `ALGS` with "`context`", "`algorithms`" and "`automaton`".

5. make the cross product of `KINDS` and `ALGS` putting a "`_`" between the two parts of each generated name.

6. add the results of this cross product to the `MODULES` list.

3.3.2 The list of algorithm families (`ALGS` in step 2 above)

In Vaucanswig, an "algorithm family" is the set of algorithms declared in a single Vaucanson header file. Most families declare only one algorithm, but usually with several forms (using overloading). In Vaucanswig, each algorithm family is related to a SWIG source file: `src/vaucanswig_alg_NAME.i` where `NAME` is the name of the algorithm family.

Each family source file contains the following items:

- a link to its C++ header.

- the definition of a bunch of SWIG macros which are able to instanciate the algorithm *declarations* for the type set given as parameters.

- the definition of a bunch of SWIG macros which are able to instanciate algorithm *wrappers* for the set of types given as parameters.

To create the list of algorithm families and associated SWIG sources, the geneeration script proceeds as follows:

1. Find all files in the Vaucanson includes that declare algorithms using the "`// INTERFACE:`" construct.

2. For each such include file, proceed as follows:

   (a) Prepend the base name of the file with "`alg_`" to make a "family name".
   (b) Create `src/vaucanswig_(family_name).i` containing the relevant SWIG code
   (c) Put the generated family name (with prefix) in the `ALGS` list.

3.3.3 The cross-product of contexts and generic code (step 5 above)

This is where you find all the magic. :)

This is the step where *real* code (i.e. non-template) is produced.

The goal of this step is to build the list of SWIG modules names *and* the source file for each SWIG module. The basic idea is simple. It relies on the following two facts:

1. each algorithm family defined above defines macros that take types as parameters and produce non-template declarations and definitions.

2. each algebra context defines a set of types, that fit as parameters in the macros for algorithm families.

Now the rest is quite simple. Since we have two lists `KINDS` (contexts) and `ALGS` (algorithm families), proceed as follows:

```
for each K of KINDS, do:
  for each A of ALGS, do:

    # Step 5.1
    instanciate macros...
    ... from src/vaucanswig_alg_${A}.i
    ... using ${K}
```

```
            . . .   into  src/vaucanswig_${K}_${A}.i

10       # Step  5.2
         add  ”${K}_${A}”  to  the MODULES  list .

     # the  following  step  is  not  fundamental ,  but  required  for  later
     # compilation :
15
     # Step  5.3  ( still  in  the  K  loop )
     add  ”${K}_context”  to  src/${K}_automaton.deps

     for  each  algorithm  family  F,  do:
20
        # Step  5.4
        add  ”${K}_automaton”  to  src/${K}_${F}.deps

        # Step  5.5
25       add  ”${K}_${F}”  to  src/${K}_algorithms.deps
```

The result of steps 5.3, 5.4 and 5.5 above can later be used to create dynamic link dependencies between object code for modules (see `build-process.txt`). It creates the following dependency graph:

```
   core  $\to$  K1_context  $\to$  K1_automaton  $\to$  K1_F1  $\to$  K1_algorithms
                                          $\to$  K1_F2  $\to$
                                          $\to$  K1_F3  $\to$

5        $\to$  K2_context  $\to$  K2_automaton  $\to$  K2_F1  $\to$  K2_algorithms
                                          $\to$  K2_F2  $\to$
                                          $\to$  K2_F3  $\to$
```

(and so on)

3.3.4   The transparency property

At every level, a property can be recognized. If an algorithm `foo()` is declared (C++) in 'bar.h', then:

- `bar` is the "algorithm family" of `foo()`

- for each selected context K, exactly one SWIG module exists and is called called 'K_bar'.

- the goal is that at the end of the compilation, in the target scripting language you can write:

```
K_bar.foo ()
# ( or  equivalent )
```

3.3.5   What is *not* automatic

Some work is required from the part of the developer:

- keeping '`// INTERFACE:`' tags in VAUCANSON headers.

- deciding a list of contexts to instantiate in Vaucanswig.

- running the generator for Vaucanswig generic code whenever the VAUCANSON library is updated.

- distributing the generated generic sources and building rules afterwards.

3.3.6 Things not easy to change *yet*

In this section, *K* stands for any algebra context.

The set of *K*-dependent types available in wrapper code in the '`// INTERFACE:`' tags is not yet easily configurable, because it involves a huge piece of hand-written dedicated code.

For the moment, the following types are available for each context `K`:

Name of type	Description
Automaton	the automaton type labeled by series
GenAutomaton	the corresponding type labeled by expressions
Series	the type of series in K
Exp	the type of expressions in K
HList	a type for lists of unsigned integer (to be used as automaton handlers where required)
Word	the type of the monoid elements

Adding more of these is not difficult, but very tedious. It involves adding a new argument in various argument list in various SWIG macros in the code. These will be documented later.

But still, it remains **very difficult** to bind in Vaucanswig any algorithm that operates on more than one algebra context at the same time. "Very difficult" here means that some major work is required to change Vaucanswig to support this case.

Chapter 4

Vaucanson as a library

To be written.

Chapter 5

Developer Guide

The chapter is a work in progress. It is not meant for users of the VAUCANSON library, but for developers and those who wish to contribute code to VAUCANSON.

5.1 Tools

5.1.1 Maintainer Tools

We use a number of tools during development which we call *maintainer tools*, because they are not required by the end user.

Autoconf Generates `configure` which probes the user system to configure the compilation.

Doxygen Reference documentation generator. Available as `doxygen` in most package systems.

rst2latex, rst2html These tools are used to convert reStructuredText into more common formats. Available in the DarwinPorts as `py-docutils`.

5.1.2 Developer Tools

Some tools help to improve the code. Use them liberally!

Valgrind

Valgrind help catching incorrect memory usage: double deletes, memory leaks, uninitialized memory readings, and so forth. Usually, to optimize speed, implementations of the C++ library don't free all the memory they allocated unless asked. Read the GCC's C++ Library FAQ, especially the item "memory leaks" in containers.

Use the following shell script to track memory leaks:

```
#! /bin/sh

exec 3>&1

5 # Ask the GNU libstdc++ to free allocated structures.
export GLIBCXX_FORCE_NEW=1

exec valgrind --num-callers=20 \
               --leak-check=yes \
10             --leak-resolution=high \
               --show-reachable=yes \
               "$@" 2>&1 1>&3 3>&- |
    sed 's/^==[0-9]*==/==/' >&2 1>&2 3>&-
```

Debugging STL

STLPort is known to be a nice implementation of STL with lots of added assertions that allow to catch common errors (mixing iterators of different containers, unstable sorting order and so forth).

I, Akim Demaille, have not succeeded to use it though. This is mainly because Debian does not provide STLPort compiled with debugging support. Nevertheless the GNU standard C++ library comes with some useful debugging features. Just add '-D_GLIBCXX_DEBUG' to your CPPFLAGS when compiling Vaucanson.

You are likely to encounter issues with *singular* iterators. It refers to default-constructed iterators, which is not the same thing as being the `end()` of a given container. They are write-only according to the STL, and copying them is not allowed.

As a result, the following code is incorrect:

```
std :: vector<std :: list <int >:: iterator > is (10);
```

because this vector constructor builds a single default object (here a default `std::list<int>::iterator`, then *copies* it ten times, which (ten times) forbidden. It there is no clear means to rewrite the code to avoid this violation, you might want to use `end()` iterators, which can be used in comparisons, can be copied, but cannot be dereferenced.

```
std :: vector<std :: list <int >:: iterator > is (10,
                             std :: list <int >().end ());
```

5.2   Contributing Code

5.2.1   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`	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/bench`	Benchmarks.
`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).

5.2.2 Writing Makefiles

Produce the output atomically Generating an output bits by bits, say with a series of '`>>` `$@`', or even with some common programs, can result in invalid files if the process failed at some point, or was interrupted. Note that in that case the (invalid) output file is newer than its dependencies, therefore it will remain. Instead, create an temporary output, say '`$@.tmp`', and as last step, rename it a '`$@`'. Sometimes, using `move-if-change` makes more sense.

Look for the economy Do what you can to save useless recompilations. Using move-if-change, especially for generated files, does save cycles. But then, beware that time stamps are not updated, which can be troublesome if the Makefile includes dependency tracking, as they will not be satisfied.

Always include dependencies included for the bootstrapping process. In their regular development process, the contributors should not have to bootstrap again, that should be only for the initial check-out, and some other situation where the layout of the project has deeply changed. Therefore, always hook the generated to the changes in the generators.

Hunt Makefile duplication Prefer an included Makefile to copy-and-paste of bits. That's also true for generated Makefiles: put the constant parts in a Makefile to be included in the generated Makefiles, rather than copying these bits several times.

This makes it easier to maintain, and also improves locality: you can edit the included Makefile and try it in just one directory, instead of having to relaunch the generation of all the Makefiles.

5.2.3 Coding Style

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

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

Document in the '`*.hh`' Do not document in the implementation files, but in the declaration files. Unless, of course, the function is private and not "exported".

Use the same signature in the declaration and in the implementation It's a bad idea not to follow the same name on both sides, and it confuses the reader. Even worse is not giving names to the argument in the declarations, giving the impression that the argument is ignored, while using it for real in the implementation.

Keep also the same template parameter names.

Inconsistency confuses users, peer developers, and... Doxygen too.

Includes

Be extremely conservative with header inclusions Do not include something that is not needed **by the file itself**. Do not include in a '***.hh**' something that is required by the '***.hxx**' file itself.

Sort the inclusions Always include standard headers first, then foreign headers we might depend upon, and then Vaucanson headers. Inside these groups, sort the includes.

Qualify header names 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**').

5.2.4   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>)
```

5.2.5   File Names

In VAUCANSON the separator is '_', not '-'. We use the following file extensions:

cc Implementation (compilation unit)

hh Declarations and documentation

hxx Inline Implementation

File names should match the class they declare, with the conversion of name conventions (i.e., from `MyClass` to '`my_class.*`').

5.2.6   Type Names

Although some coding standards recommend against this practice, types in VAUCANSON should end with '`_t`'. One exception is traits, where `ret` is commonly used.

`self_t`, when defined, always refers to the current class.

`super_t`, when defined, always refers to *the* super class. When there are several, `super_t` is not used. The macros `INHERIT_TYPEDEF` and `INHERIT_TYPEDEF_` rely on this convention.

5.2.7 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_;
```

5.2.8 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
15 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.
5 * @arg io     Related stream.
 * @arg idx    Index in the internal extensible array of a pointer to delete.
 * ... */
```

5.2.9   Writing Algorithms

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

Document the algorithm See Section 5.2.8.

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 5.2.10 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]").

5.2.10   Writing Tests

5.2.11   Mailing Lists

Vauc comes with a set of mailing lists:

vaucanson@lrde.epita.fr General discussions, feature requests 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.

5.3 Vaucanson I/O

January 2005

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

5.3.1 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`.

5.3.2 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.

5.3.3   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.

5.3.4   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.

5.3.5   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),

- more compact,

- not strongly typed (can be dumped from one automaton type and loaded to another).

Drawbacks relative to XML:

- not strongly 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.

5.3.6 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.)

5.3.7   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 ()
```

5.3.8 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.

5.3.9 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`.
> As of today (2006-03-17) the FSM format is only available using the fsm_load() and fsm_dump() interface.

Appendix A

Automaton Library

VAUCANSON comes with a set of interesting automata that can be used to toy with TAF-KIT (Chapter 2) for instance. In the chapter, we present each one of these automata.

A.1 Boolean Automata

A.1.1 a1
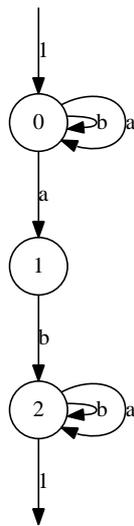
{ 3 states, 6 transitions, #I = 1, #T = 1 }

A.1.2   b1
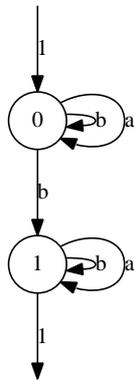
{ 2 states, 5 transitions, #I = 1, #T = 1 }

A.1.3   div3base2
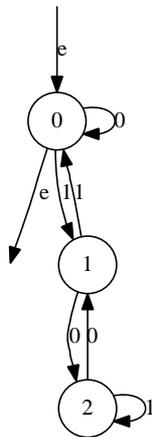
{ 3 states, 6 transitions, #I = 1, #T = 1 }

A.1.4   double-3-1
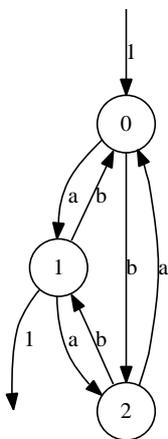
{ 3 states, 6 transitions, #I = 1, #T = 1 }

A.1.5 ladybird-6
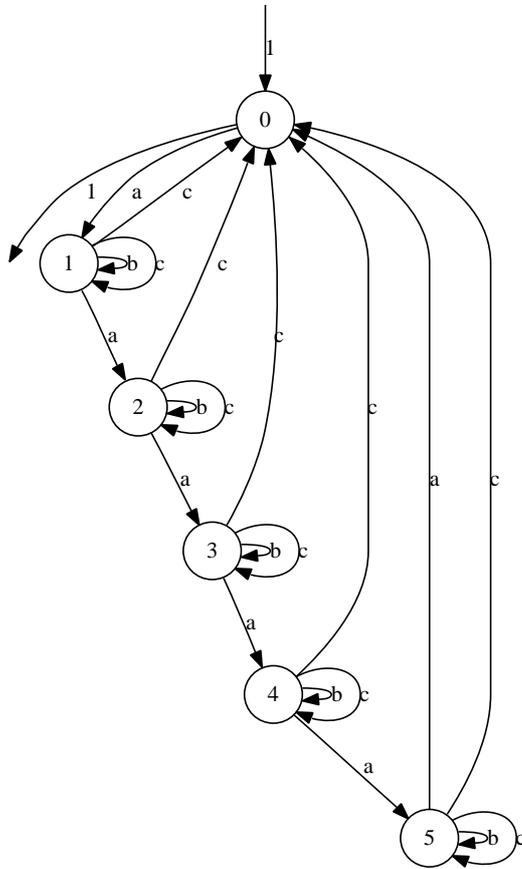
{ 6 states, 21 transitions, #I = 1, #T = 1 }

A.2 $\mathbb{Z}$-Automata

A.2.1 b1
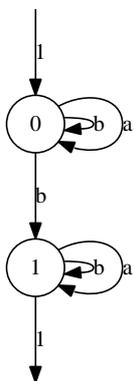
{ 2 states, 5 transitions, #I = 1, #T = 1 }

A.2.2 c1
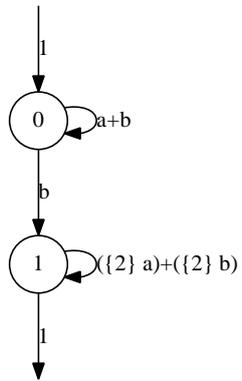
{ 2 states, 3 transitions, #I = 1, #T = 1 }

A.2.3 div3base10
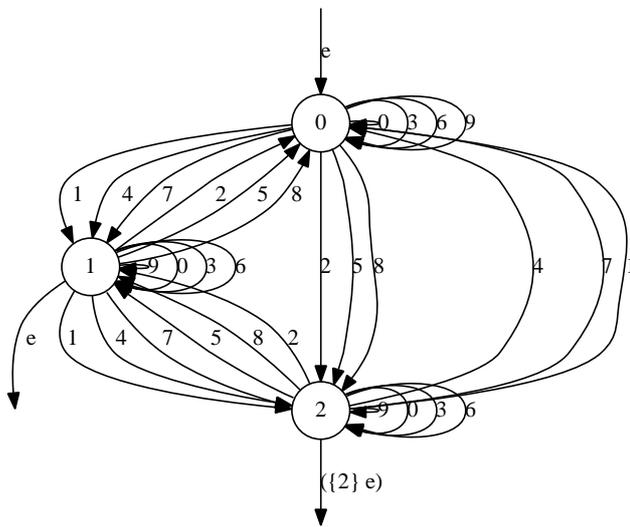

{ 3 states, 30 transitions, #I = 1, #T = 2 }

A.3 Boolean FMP Transducers

A.3.1 quot3base2
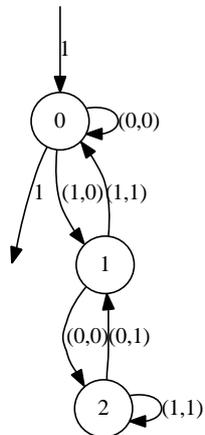
{ 3 states, 6 transitions, #I = 1, #T = 1 }

A.3.2 t1
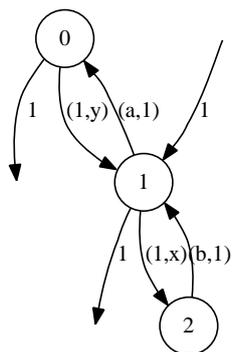
{ 3 states, 4 transitions, #I = 1, #T = 2 }

A.3.3 u1
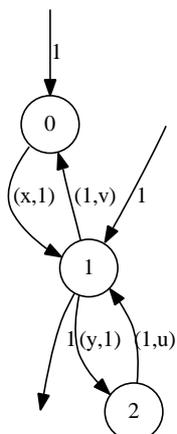
{ 3 states, 4 transitions, #I = 2, #T = 1 }

Appendix B

Bits of Automaton Theory

B.1   On standard and normalized automata

B.1.1   Standard automata

Definition

Definition B.1 (Standard Automaton) *An automaton (any kind, automata over any monoid with any multiplicity) is said to be* standard *if it has a unique initial state which is the destination of no transition and whose 'initial multiplicity' is equal to the identity (of the multiplicity semiring or of the series semiring, according to the current convention).*

remark B.2 *These terminology and definition are to be found in ETA and are not (yet) universally known or accepted.*

Standardization

Not only every automaton is equivalent to a standard one, but a simple procedure, called 'standardization', transforms every automaton A in an equivalent standard one, and goes as follows.

1. Add a new state s, make it initial, with initial multiplicity equal to the identity.

2. For every initial state i of A, with initial multiplicity $I(i)$, add a transition from s to i with label I(i), and set I(i) to 0 (the zero of the semiring, or of the series – as above).

3. Suppress all epsilon-transition from the created transitions by a backward closure.

4. Take the accessible part of the result.

remark B.3 *Steps item 3 and item 4 are necessary to insure the following property:*
The standardization of a standard automaton A is isomorphic to A .
More informally, but more generally, they insure that the result of the standardization is of the same "kind" as the automaton on which it is applied (in particular, without epsilon-transition if A is without epsilon-transition).

Standard automaton of an expression

A classical algorithm — often credited to Glushkov — transforms a rational (ie regular) expression (of literal length n) into a standard automaton (with n+1 states). This automaton is known in the literature as the 'Glushkov automaton' or as the 'position automaton' of the expression.

remark B.4 *It is folklore that the epsilon-transition removal — via a "backward closure" — applied to the 'Thompson automaton' of a rational expression produces the standard automaton of the expression.*

remark B.5 *These definitions, constructions and properties are fairly classical for classical automata. Their generalization to automata with multiplicity is more recent (mostly written by the "Rouen school" around the year 2000).*

B.1.2   Normalized automaton

Definition

An automaton (any kind, automata over any monoid with any multiplicity) is said to be *normalized* if

1. it has a unique initial state

   - which is the destination of no transition,
   - whose 'initial multiplicity' is equal to the identity (of the multiplicity semiring or of the series semiring, according to the current convention),
   - and whose 'final multiplicity' is equal to the zero (with the same convention);

2. and, symmetrically, it has a unique final state

   - which is the source of no transition,
   - whose 'final multiplicity' is equal to the identity,
   - and whose 'initial multiplicity' is equal to the zero.

remark B.6 *The terminology is rather unfortunate, for there are already so many different "normalized" things. The notion however, is rather classical, under this name, at least for classical Boolean automata, because of one classical proof of Kleene theorem. For the same reason, it is a proposition credited to Schutzenberger that every weighted automaton A is equivalent to a normalized one, provided the empty word is not in the support of the series realized by A, although the word normalized is not used there. The terminology is even more unfortunate since "normalized transducer" has usually an other meaning, and corresponds to transducers whose transitions have label of the form either (a,1) or (1,b) .*

Normalization

It is not true that every automaton is equivalent to a normalized one. This holds only for automata whose accepted language does not contain the empty word (for classical automata) or whose realized series gives a zero coefficient to the empty word (for weighted automata). There exists however a "normalization procedure" which plays mutatis mutandis the same role as the standardization and which is best described with the help of the standardization.

Let A be an automaton.

Let $B = $ standardize(transpose(standardize(transpose(A))))

Let i be the (unique) initial state of B and let C be the automaton obtained from B by setting T(i)=0 — ie setting to 0 the terminal function. Then, C is normalized, we write C = normalize(A) and it holds:

for classical automata: The language accepted by normalize(A) is equal to the language accepted by A minus the empty word (if it is accepted by A):

$$L(normalize(A)) = L(A) \; 1_{X^*}$$

(where X is the alphabet.)

for weighted automata: the series realized by normalize(A) is eke to the one realized by A, but for the coefficient of 1_{X^*} which is 0:

$$|normalize(A)| = |A| \odot char(X^+)$$

(where $char(X^+)$ is the characteristic series of X^+.

that is, in both cases, normalize(A) accepts or realizes the 'proper' part of the language accepted, or of the series realized by, A.

B.1.3 Operations on automata

These families of automata have been considered in order to establish one direction of Kleene's theorem, the one that amounts to show that languages accepted (or series realized) by finite automata are closed under rational operations: sum, product and star.

The sum

The sum is never a problem: the union of two automata is an automaton whose behavior is the sum of the behaviors of these automata.

remark B.7 *If we consider automata with unique initial and/or final state, it would be a bad idea to realize the sum by merging the initial and/or final states of the two automata in order to recover automata of the same kind — unless these initial states have no incoming transitions and/or these final states have no outgoing transitions, that is if we consider standard automata, transpose of standard automata, or normalized automata.*

The concatenation

The product (of accepted language or of realized series) is carried out by the "concatenation" of automata — since we keep the word "product" for the Cartesian product of automata which realizes the intersection of languages or the Hadamard product of series. For classical automata, the concatenation of A and B can be described as follows: add an epsilon-transition from every final state of A to every initial state of B, and suppress the epsilon-transition (if necessary, and by any closure algorithm). For weighted automata, the 'same' algorithm is more easily described by using a standardization step: compute A' the 'co-standardized' automaton of A, compute B' the standardized automaton of B, add an epsilon-transition (with label identity) from the unique final state of A' to the unique initial state of B' and suppress this new transition (if necessary and by any closure algorithm).

remark B.8 *Along the same line as above, if A has a unique final state t and B a unique initial state j, it would be a bad idea to realize the concatenation of A and B by merging t and j —unless t has no outgoing transition, that is if A is 'co-standard' or normalized.*

The star

The "star" of an automaton A, realizing the star of the accepted language or of the realized series, is even more subtle.

If A is normalized, it is easily carried out by the merging of the initial and final states of A. Since the series accepted by a normalized automaton is proper, its star is always defined, this is the advantage of the construction. On the other hand, star(A) is not normalized anymore, and if this operation is used inside an algorithm that builds an automaton from an expression, it yields an explosion of the number of states.

If A is standard, with initial state i and initial multiplicity c (usually a scalar), the star of —A— is defined if, and only if, the c^* is defined (Sakarovitch, 2003, Prop. III.2.6) — if A has no epsilon-transition. In this case, star(A) is defined as follows:

1. replace the initial multiplicity by c^*;

2. for every final state t of A, add a new transition from t to i with label $T(t) \times 1_{X^*}$;

3. Suppress the epsilon-transition via backward closure.

remark B.9 *If A is not standard, it would be a bad idea to use the above construction, even letting aside the multiplicity — although it may have occurred to knowledgeable people.*

B.1.4 Conclusion

Normalized and standard automata have been introduced in relation with the proof of Kleene's theorem. If one does not want to introduce epsilon-transition, the notion of normalized automata yields certainly the most straightforward argument. The advantage of standard automata is that they not only can be used for the same proof, but they also yields an efficient algorithm, both for the size of the result and for the computational complexity, to transform a rational expression into an automaton.

It took me some times to get to this conclusion. If I were to rewrite a new edition of Sakarovitch (2003), I would not mention normalized automata besides exercises and historical notes. All the theory would be presented with standard automata only.

Appendix C

A proposal for an XML format for automata

This is not a complete description of the VAUCANSON proposal for an XML format for automata. The interested reader will find such a description at the following URL. We just present here few examples of files, that should give an idea on how these files are built.

```
<fsmxml xmlns="http://vaucanson.lrde.epita.fr" version="1.0">

  <automaton>
    <valueType>
      <semiring operations="classical" set="B" type="numerical"/>
      <monoid genDescrip="enum" genKind="simple" genSort="letters" type="free">
        <monGen value="a"/>
        <monGen value="b"/>
      </monoid>
    </valueType>
    <automatonStruct>
      <states>
        <state id="s0"/>
        <state id="s1"/>
        <state id="s2"/>
      </states>
      <transitions>
        <transition source="s1" target="s2">
          <label>
            <monElmt>
              <monGen value="b"/>
            </monElmt>
          </label>
        </transition>
        <transition source="s0" target="s0">
          <label>
            <monElmt>
              <monGen value="b"/>
            </monElmt>
          </label>
        </transition>
        <transition source="s0" target="s1">
          <label>
            <monElmt>
              <monGen value="a"/>
            </monElmt>
```

```
              </label>
            </transition>
            <transition source="s0" target="s0">
40              <label>
                <monElmt>
                  <monGen value="a"/>
                </monElmt>
              </label>
45            </transition>
            <transition source="s2" target="s2">
              <label>
                <monElmt>
                  <monGen value="b"/>
50              </monElmt>
              </label>
            </transition>
            <transition source="s2" target="s2">
              <label>
55              <monElmt>
                  <monGen value="a"/>
                </monElmt>
              </label>
            </transition>
60          <initial state="s0"/>
            <final state="s2"/>
          </transitions>
        </automatonStruct>
      </automaton>
```

Appendix D

Algorithms specifications

D.1 Vocabulary

In VAUCANSON, we use a precise vocabulary to speak about automaton. As is it specific to our project and some expressions may be not widely used or approved by the automata community, we choose to define them here.

$\mathbb{B}$ Boole's semiring.

Boolean automaton is a "classical" automaton. Precisely, it is a automaton over a free monoid which transitions are labeled by letters of an alphabet with multiplicity in $\mathbb{B}$.

automaton with multiplicity in $\mathbb{B}$ is an automaton over any kind of monoid (in VAUCANSON we have free monoid and product of free monoids) with its multiplicity in $\mathbb{B}$.

realtime automaton is an automaton over a monoid which transitions are labelled by letters only (not words).

FMP-transducer is a transducer over a free monoid product.

RW-transducer is a transducer over a series $\mathbb{K}' << \mathbb{K} << A^* >>>>$.

D.2 Algorithms applicability in Vaucanson

D.2.1 Algorithms on graph

accessible (accessible.hh)

accessible_states (accessible.hh)

coaccessible (accessible.hh)

coaccessible_states (accessible.hh)

trim (trim.hh)

useful_states (trim.hh)

sub_automaton (sub_automaton.hh)

is_void

D.2.2 Algorithms on labeled graphs

are_isomorphic (isomorph.hh)

aut_to_exp (aut_to_exp.hh)

sum (sum.hh)

thompson_of (thompson.hh)

is_normalized (normalized.hh)

normalize (normalized.hh)

union_of_normalized (normalized.hh)

concatenate_of_normalized (normalized.hh)

star_of_normalized (normalized.hh)

standard_of (standard_of.hh)

standardize (standard.hh)

is_standard (standard.hh)

union_of_standard (standard.hh)

concat_of_standard (standard.hh)

star_of_standard (standard.hh)

D.2.3 Algorithms on labeled graphs (epsilon-transitions are distinguish)

generalized

closure (closure.hh)

backward_closure (closure.hh)

forward_closure (closure.hh)

concatenate (concatenate.hh)

cut_up (cut_up.hh)

D.2.4 Algorithms on graphs labeled on $\mathbb{K} << A^* >>$

is_realtime (realtime_decl.hh)

backward_realtime (backward_realtime.hh)

forward_realtime (forward_realtime.hh)

realtime (realtime.hh)

D.2.5 Algorithms on graphs labeled on series of letter with multiplicities $\left(\sum \left(a, \mathbb{K}_{a*}a\right)\right)$

product (product.hh)

eval (eval.hh)

evaluation (evaluation.hh)

is_ambiguous

is_deterministic (determinize.hh)

is_sequential

quotient (minimization_hopcroft.hh)

derived_term_automaton (derived_term_automaton.hh)

broken_derived_term_automaton (derived_term_automaton.hh)

complete (complete.hh)

is_complete (complete.hh)

transpose (transpose.hh)

D.2.6 Algorithms on Boolean automata

determinize (determinize.hh)

brzozowski (brzozowski.hh)

berry_sethi (berry_sethi.hh)

canonical (aci_canonical.hh)

complement (complement.hh)

minimization_moore (minimization_moore.hh)

co_minimization_moore (minimization_moore.hh)

minimization_hopcroft (minimization_hopcroft.hh)

search (search.hh)

D.2.7 Algorithms on automata with multiplicities in $\mathbb{K} << A^* >>$

domain

image

extension (extension.hh)

inverse

D.2.8 Algorithms on realtime transducers

evaluation (evaluation.hh)

realtime_composition (realtime_composition.hh)

realtime_to_fmp (realtime_to_fmp.hh)

D.2.9 Algorithms on realtime RW-transducers

letter_to_letter_composition (letter_to_letter_composition.hh)

D.2.10 Algorithms on FMP-transducers

domain (projections_fmp.hh)

image (projections_fmp.hh)

extension (extension.hh)

identity (projections_fmp.hh)

evaluation_fmp (evaluation_fmp.hh)

inverse

insplitting (outsplitting.hh)

outsplitting (outsplitting.hh)

sub_normalize (sub_normalize.hh)

normalized_composition (normalized_composition.hh)

fmp_to_realtime (fmp_to_realtime.hh)

D.2.11 Algorithms on Boolean FMP-transducers

b_composition (normalized_composition.hh)

D.2.12 Algorithms on regular expressions over $\mathbb{K} << A^* >>$

flatten (krat_exp_flatten.hh)

expand (krat_exp_expand.hh)

Index

#include, 40
cpp, 40
maintainer tools, 37
eval, 12
-o dot identity, 10
aut-to-exp, 16
determinize, 11
display, 10
identity, 9
info, 11
list-automata, 8
power, 21
quotient, 22
vcsn-char-b, 7
vcsn-char-z, 21

$\mathcal{A}_1$, 9

$\mathcal{B}_1$, 21

directories, 38

iterator
    singular, 38

maintainer tools, 37, 65
minimize, 16

$\mathbb{N}$-automaton, 21
normalized, 56

singular, 38
standard, 55
Standard Automaton, 55

$\mathcal{T}_1$, 19

Bibliography

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