The Vaucanson TAF-Kit 1.2.91
User's Manual

The Vaucanson Group

2008-08-23

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-1.2.91
```

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 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 3.1.1 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 dump-automaton a1 | vcsn-char-b determinize - | vcsn-char-b minimize - | vcsn-char
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.

```
# vcsn-char-b dump-automaton a1 | 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: lt-vcsn-char-b [OPTION...] <command> <args...>
VCSN TAF-Kit -- a toolkit for working with automata

  -a, --alphabet=ALPHABET    Set the working alphabet for rational expressions
  -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 input type (FSM, XML or DOT)
  -O, --bench-plot-output=OUTPUT_FILENAME
                             Bench output filename
  -p, --parser=OPTIONS       Set the working parser 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, 1 as epsilon
         'alpha': Use [a-zA-Z] as the alphabet, 1 as epsilon
         'digits': Use [0-9] as the alphabet, 1 as epsilon
         'ascii': Use ascii characters as the alphabet, 1 as epsilon


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

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

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

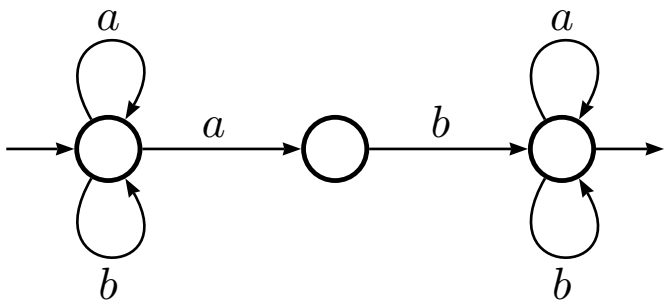
The whole list of supported commands is available via '`--list-commands`'.

2.1.2 A first example

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

```
# vcsn-char-b list-automata
The following automata are predefined:
  - a1
  - b1
```

```
- div3base2
- double-3-1
- ladybird-6
```



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.

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 `dump-automaton`:

```
# vcsn-char-b dump-automaton a1

<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 src="s0" target="s0">
          <label>
            <monElmt>
              <monGen value="b"/>
            </monElmt>
          </label>
        </transition>
        <transition src="s0" target="s1">
          <label>
            <monElmt>
              <monGen value="a"/>
```

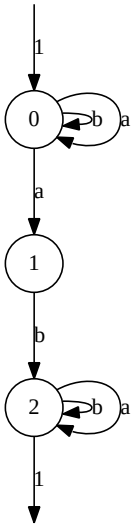
```
          </monElmt>
        </label>
      </transition>
      <transition src="s0" target="s0">
        <label>
          <monElmt>
            <monGen value="a"/>
          </monElmt>
        </label>
      </transition>
      <transition src="s2" target="s2">
        <label>
          <monElmt>
            <monGen value="b"/>
          </monElmt>
        </label>
      </transition>
      <transition src="s2" target="s2">
        <label>
          <monElmt>
            <monGen value="a"/>
          </monElmt>
        </label>
      </transition>
      <transition src="s1" target="s2">
        <label>
          <monElmt>
            <monGen value="b"/>
          </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 dump-automaton a1 >a1.xml
```

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

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



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

Determinization of $\mathcal{A}_1$

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

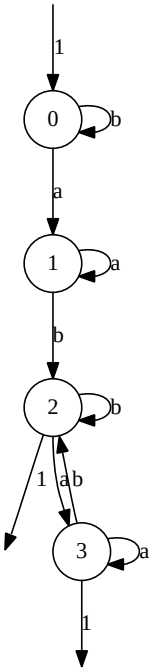
```
# vcsn-char-b dump-automaton a1 | vcsn-char-b determinize - >a1det.xml
```

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 dot-dump a1det.xml >a1det.dot
```



A { 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)
```

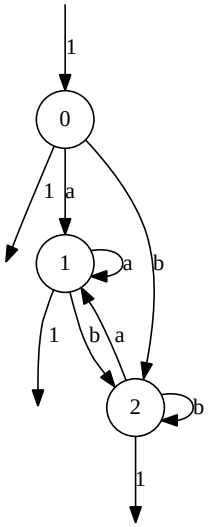
Interestingly enough, states are numbered from 0, but transitions numbers start at 1. Also, not that weights are reported, although only `1` is valid for Boolean automata.

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 dot-dump all.xml >all.dot
```



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

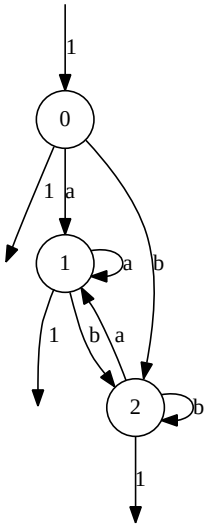
2.1.5 Token representations

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 dot-dump parser.xml >parser.dot
```

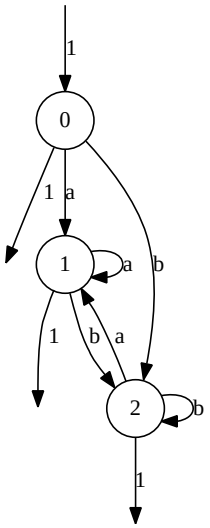


A { 3 states, 6 transitions, #I = 1, #T = 3 }

, 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 dot-dump parser2.xml >parser2.dot
```



A { 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 `=`.

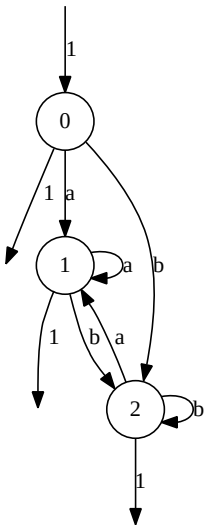
Finally whenever you may put a single character you may write a pair a words. The following construction is then valid: `ALPHABET=(\,,\))`, and defines the alphabet with only one letter: the pair whose first component is `,` and the second one is `)`.

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 dot-dump allmin.xml >allmin.dot
```



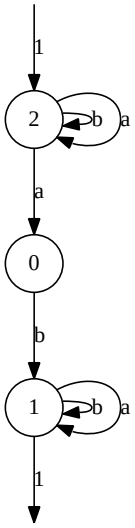
A { 3 states, 6 transitions, #I = 1, #T = 3 }

Computing the language recognized by a Boolean automaton can be done using `aut-to-exp`:

```
# vcsn-char-b aut-to-exp all.xml
(b+a.a*.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 dot-dump l1.xml >l1.dot
```



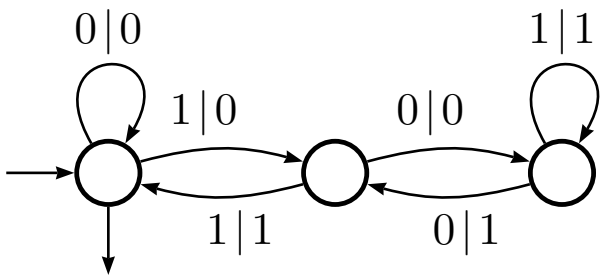
A { 3 states, 6 transitions, #I = 1, #T = 1 }

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:
   - define-automaton file: Define an automaton from scratch.
   - display aut: Display ‘aut’.
   - dot-dump aut: Dump dot output of ‘aut’.
   - dump-automaton file: Dump a predefined automaton.
   - edit-automaton file: Edit an existing automaton.
   - identity aut: Return ‘aut’.
   - info aut: Print useful infos about ‘aut’.
   - list-automata: List predefined automata.
 * Tests and evaluation on automata:
   - are-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 em
   - 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’.
  - 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 3.3.1).

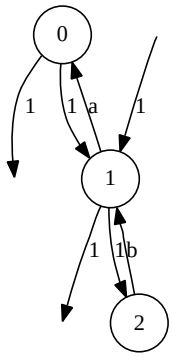
Domain

The transducer T only accepts binary numbers divisible by 3.

```
# vcsn-char-fmp-b dump-automaton t1 | 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 dot-dump div-by-3.xml >div-by-3.dot
```

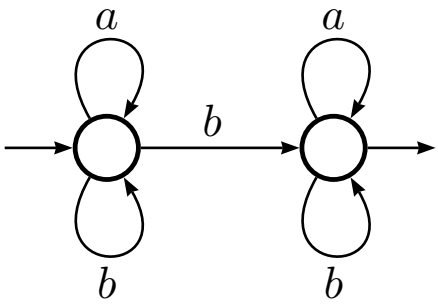


A { 3 states, 4 transitions, #I = 1, #T = 2 }

2.2.2 Available functions

The following functions are available for both `vcsn-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:
   - define-automaton file: Define an automaton from scratch.
   - display aut: Display 'aut'.
   - dot-dump aut: Dump dot output of 'aut'.
   - dump-automaton file: Dump a predefined automaton.
   - edit-automaton file: Edit an existing automaton.
   - identity aut: Return 'aut'.
   - info aut: Print useful infos about 'aut'.
   - list-automata: List predefined automata.
 * Tests and evaluation on transducers:
   - 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.
 * 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'.
   - image aut: Give an automaton that accepts all output produced by 'aut'.
   - trim aut: Trim transducer 'aut'.
 * Algorithms for transducers:
   - sub-normalize aut: Give the sub-normalized transducer of 'aut'.
   - composition-cover aut: Outsplitting.
   - composition-co-cover aut: Insplitting.
   - compose aut1 aut2: Compose 'aut1' and 'aut2', two (sub-)normalized
       transducers.
   - u-compose aut1 aut2: Compose 'aut1' and 'aut2', two Boolean transducers,
       preserve the number of path.
   - to-rw aut: Give the equivalent rational weight transducer of 'aut'.
   - invert aut: Give the inverse of 'aut'.
   - intersection aut: Transform a Boolean automaton in a fmp transducer by
       creating, for each word, a pair containing twice this word.
```



Considered without weight, $\mathcal{B}_1$ accepts words with a '`b`'. With weights, it counts the number of '`b`'s.

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 3.2.

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 dump-automaton b1 | vcsn-char-z eval - 'abbb'
3

# vcsn-char-z dump-automaton b1 | 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 dump-automaton b1 | 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 dump-automaton b1 | 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:
   - define-automaton file: Define an automaton from scratch.
   - display aut: Display 'aut'.
   - dot-dump aut: Dump dot output of 'aut'.
   - dump-automaton file: Dump a predefined automaton.
   - edit-automaton file: Edit an existing automaton.
   - identity aut: Return 'aut'.
   - info aut: Print useful infos about 'aut'.
   - list-automata: List predefined automata.
 * Tests and evaluation on automata:
   - 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 em
   - 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’.
  - standard exp: Give the standard automaton of ‘exp’.
  - thompson exp: Give the Thompson automaton of ‘exp’.
```

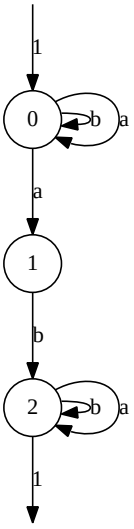
Chapter 3

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.

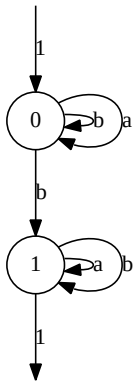
3.1 Boolean Automata

3.1.1 a1



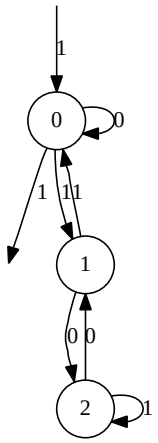
A { 3 states, 6 transitions, #I = 1, #T = 1 }

3.1.2 b1



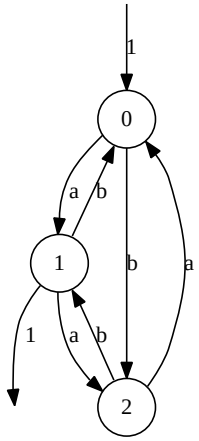
A { 2 states, 5 transitions, #I = 1, #T = 1 }

3.1.3 div3base2



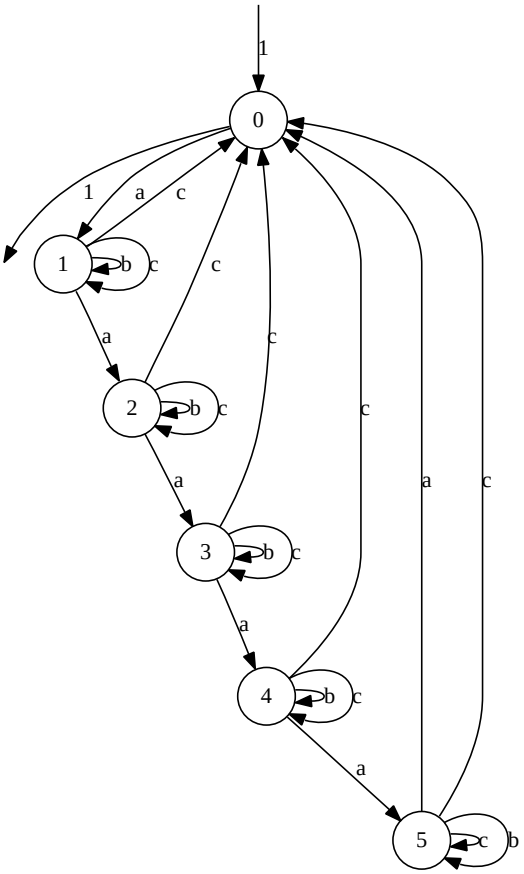
A { 3 states, 6 transitions, #I = 1, #T = 1 }

3.1.4 double-3-1



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

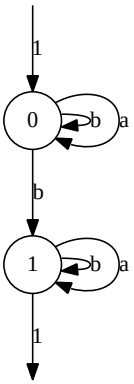
3.1.5 ladybird-6



A { 6 states, 21 transitions, #I = 1, #T = 1 }

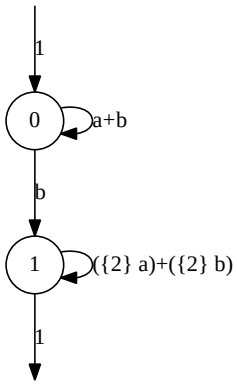
3.2 $\mathbb{Z}$-Automata

3.2.1 b1



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

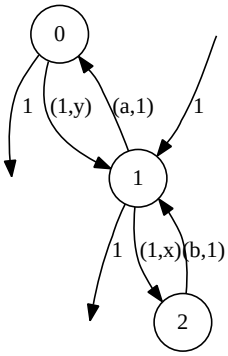
3.2.2 c1



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

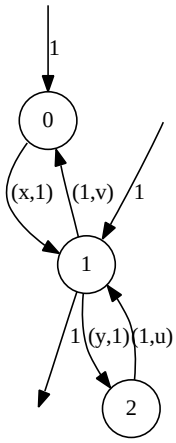
3.3 Boolean FMP Transducers

3.3.1 t1



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

3.3.2 u1



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