# TAF-Kit's Documentation

compiled on September 22, 2009, for version 1.3.2

# Introduction

VAUCANSON is a free software platform dedicated to the manipulation of finite state automata. Here, 'finite state automata' is to be understood in the broadest sense: VAUCANSON supports *weighted* automata over a free monoid, and even *weighted* automata on some *non free monoids* (currently only products of free monoids—a.k.a. transducers—are supported).

The platform consists in a couple of components:

**The Vaucanson library** is a C++ library that implements objects for automata, rational expressions, as well as algorithms on these objects. This library is generic, in the sense that it makes it possible to write an algorithm once and apply it to different types of automata. However this genericity is achieved in a way that should not cause any slowdown at runtime: because the type of the automaton manipulated is known at compile time, compiling an algorithm will generate code that is almost as efficient as an algorithm written specifically for this type of automaton.

**TAF-Kit** is a command-line interface to the library that allows user to execute VAUCANSON's algorithms without any knowledge of C++. Because the VAUCANSON library needs to know the type of automata at compile time, the TAF-KIT interface has been instantiated for a predefined set of common automaton types.

TAF-KIT does not allow to write new algorithms nor to manipulate new types of automata, but it makes it possible to combine without efforts a large set of algorithms on common automata types.

**A repository of automata** that shows examples of automata of various types, and also contains tools to create families of automata.

Ideally this manual should document all of these components. Presently it mostly documents TAF-KIT, because that is the more accessible part of VAUCANSON.

# Chapter 1

# Administrativia

## 1.1 Getting Vaucanson

The latest version of the VAUCANSON platform can be downloaded from `http://vaucanson.lrde.epita.fr/`.

## 1.2 Licensing

Vaucanson is a free software released under the GNU General Public Licence version 2. If you are unfamiliar with this license, please read the file '`COPYING`' (at the root of the source tree) for details.

## 1.3 Building Vaucanson

Detailed information is provided in the files '`INSTALL`', which is generic to all packages using the GNU Build System, and '`doc/README.pdf`' which details VAUCANSON's specific build process. The following installation commands will install VAUCANSON in '`/usr/local/`'.

```
$ cd vaucanson-1.3.2
$ ./configure
$ make
$ sudo make install
```

Although we discourage it, you may also use VAUCANSON without installing it: you would have to use *`-I /full-path-to/vaucanson-1.3.2/include`* when compiling C++ programs, and move to directory '`vaucanson-1.3.2/taf-kit/tests/`' to execute TAF-KIT. (This directory contains wrapper around the real TAF-KIT programs from '`taf-kit/src/`' that enable them to run locally.)

## 1.4 Getting in Touch

Please send any question or comment to `vaucanson@lrde.epita.fr`, and report bugs to either our issue tracker at `http://vaucanson.lrde.org/` or by mail to `vaucanson-bugs@lrde.epita.fr`.

You can subscribe to these mailing lists at `https://www.lrde.epita.fr/cgi-bin/mailman/listinfo/vaucanson` and `https://www.lrde.epita.fr/cgi-bin/mailman/listinfo/vaucanson-bugs` if you like, but this is not a requirement for sending e-mails.

# Chapter 2

# TAF-Kit

TAF-Kit is a command-line interface to Vaucanson. It is a set of programs that should be called from the shell and that can be used to chain operations on automata.

## 2.1 TAF-Kit Instances

In the generic programming paradigm used in the Vaucanson library, the types of the automata manipulated have to be known at compile time. TAF-Kit has therefore been compiled for several predefined types of automata. It is actually the same program that is *instantiated* for different kinds of alphabets and weights. Here are the names of these instances, and the kind of automata they represent:

| program name | automaton type | alphabet type | weight semiring |
|---|---|---|---|
| `vcsn-char-b` | automata | characters | $\langle \mathbb{B}, \vee, \wedge \rangle$ |
| `vcsn-int-b` | automata | integers | $\langle \mathbb{B}, \vee, \wedge \rangle$ |
| `vcsn-char-z` | automata | characters | $\langle \mathbb{Z}, +, \times \rangle$ |
| `vcsn-char-zmax` | automata | characters | $\langle \mathbb{Z}, \max, + \rangle$ |
| `vcsn-char-zmin` | automata | characters | $\langle \mathbb{Z}, \min, + \rangle$ |
| `vcsn-char-r` | automata | characters | $\langle \mathbb{R}, +, \times \rangle$ |
| `vcsn-char-char-b` | automata | pairs of characters | $\langle \mathbb{B}, \vee, \wedge \rangle$ |
| `vcsn-char-int-b` | automata | pairs of character and integer | $\langle \mathbb{B}, \vee, \wedge \rangle$ |
| `vcsn-int-int-b` | automata | pairs of integers | $\langle \mathbb{B}, \vee, \wedge \rangle$ |
| `vcsn-char-fmp-b` | transducers | characters | $\langle \mathbb{B}, \vee, \wedge \rangle$ |
| `vcsn-char-fmp-z` | transducers | characters | $\langle \mathbb{Z}, +, \times \rangle$ |
| `vcsn-int-fmp-b` | transducers | integers | $\langle \mathbb{B}, \vee, \wedge \rangle$ |
| `vcsn-int-fmp-z` | transducers | integers | $\langle \mathbb{Z}, +, \times \rangle$ |

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. `vcsn-char-b` is the TAF-Kit instance they should use. A variant of this program, called `vcsn-int-b`, handles Boolean automata whose letters are integers. Other variants such as `vcsn-char-char-b`, `vcsn-char-int-b`, or `vcsn-int-int-b`, support alphabets of pairs. All of these are called Boolean automata because each word is associated to a Boolean *weight*: either the word is accepted and its weight is *true*, or it is not and its weight is *false*.

Vaucanson actually supports automata with multiplicities, where words can be associated to weights taken in any semiring. For instance `vcsn-char-z` associates each word to an integer. The previous table show other semirings that can be used as well.

Vaucanson also supports weighted transducers. These transducers are actually automata over a product two free monoids. In Vaucanson we call these FMP, for *free monoid products*. The above table lists a few TAF-Kit instance for FMP.

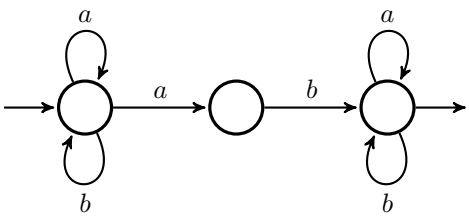



Figure 2.1: The automaton $\mathcal{A}_1$, defined over the alphabet $A = \{a, b\}$ recognizes any word of $A^\star$ that contains $ab$.

## 2.2 A First Contact

We are about to play with automaton $\mathcal{A}_1$ pictured on Figure 2.1. TAF-Kit comes with a set of predefined automata, and $\mathcal{A}_1$ happens to be one of those: it is called '`a1.xml`'. This is a Boolean automaton whose alphabet consists in two characters $\{a, b\}$ so will shall use the `vcsn-char-b` instance of TAF-Kit.

If you have fully installed Vaucanson (see section 1.3) you should be able to just type any of the following commands and observe their results. If you only compiled Vaucanson without installing it, you should `cd` into the '`vaucanson-1.3.2/taf-kit/tests/`' directory and type '`./vcsn-char-b`' instead of '`vcsn-char-b`' for each of the following commands.

The following command will just make sure that TAF-Kit knows about this automaton. It will display the number of states, transitions, initial states, and final states of $\mathcal{A}_1$.

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

If you have the GraphViz package installed (see '`doc/README.pdf`' for links) you can also display that automaton with:

```
$ vcsn-char-b display a1.xml
```

The displayed automaton won't have a layout as pretty as Figure 2.1, but it represents the same automaton nonetheless.

$\mathcal{A}_1$ is a non-deterministic automaton. We could determinize it with the `determinize` command of TAF-Kit. As most commands of TAF-Kit, `determinize` produces its output (an XML file representing the automaton) on the standard output, so we will want to divert it to a file using a shell redirection.

```
$ vcsn-char-b determinize a1.xml > a1det.xml
$ vcsn-char-b info a1det.xml
States: 4
Transitions: 8
Initial states: 1
Final states: 2
```

The determinized automaton has 4 states and 8 transitions.

Please note that '`a1det.xml`' is a file that we just created into the current directory while '`a1.xml`' is a file that is predefined in Vaucanson's predefined automata repository. We can call command `info` on either files using the same syntax because TAF-Kit will look for automata in both places. The command '`vcsn-char-b list-automata`' will list all predefined automata for this instance of TAF-Kit. See also chapter 3 for a presentation of these files.

In the pure Unix tradition, we can of course chain commands with pipes. For instance the above two commands could be rewritten

```
$ vcsn-char-b determinize a1.xml | vcsn-char-b info -
States: 4
Transitions: 8
Initial states: 1
Final states: 2
```

where '`-`' stands for '*read from standard input*'.

TAF-Kit actually supports a more efficient way of chaining commands: the *internal pipe*. It's called *internal pipe* because the pipe logic is taken care of by TAF-Kit himself, but actually it is not using a Unix pipe at all: the commands are simply serialized in the same process, using the automata object created by the previous one. It is more efficient because the automaton does not have to be converted into XML for output, and then parsed back as input of the next command in the chain. Here is how the above command would look using an *internal pipe*; notice how the '`|`' symbol is protected from its evaluation by the shell.

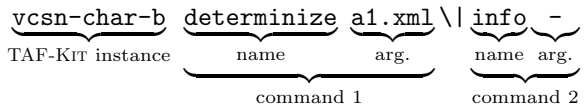
```
$ vcsn-char-b determinize a1.xml \| info -
States: 4
Transitions: 8
Initial states: 1
Final states: 2
```

In the above command, '`-`' does not designate the standard input, it denotes *the result of the previous command*.

## 2.3 TAF-Kit's Modus Operandi

All TAF-Kit instances work identically. They just differ on the type of automata they handle, and may offer different algorithms because not all algorithms work on any automata type.

Any time TAF-Kit is run, it breaks its command line into command names and arguments.




The *internal pipe*, '`\|`', is used to separate commands. A command start with a name, is can be followed by several arguments (although only one is used in the above example). These arguments can be very different depending on the command. The far we have used filenames as well as '`-`' (to designate either the standard input or the result of the previous command). Some commands will also accept plain text representing for instance a word or a rational expression.

All commands will also accept some options. There are options to define what the alphabet is, options to define the types to use for input and output, even options to fine-tune how some symbols will be printed. We shall get back to these options in **??**.

For each command, TAF-Kit will

1. parse the options
2. parse all expected arguments (using indications that may have been given as options)
3. execute the algorithm
4. print the result (in a format that can be controlled using options)

When commands are chained internally using '`\|`' and '`-`', the parsing steps and printing steps are of course omitted.

## 2.4 Writing (Weighted) Rational Expressions

### 2.4.1 Rational operators

Any word can be used as a rational expression. Additionally the following operators can be used to combine rational expressions.

| | |
|---|---|
| $e$`*` | Kleene star |
| $e_1e_2$ | implicit concatenation |
| $e_1$`.`$e_2$ | explicit concatenation |
| $e_1$`+`$e_2$ | disjunction |
| `(`$e$`)` | grouping |

For instance, on the alphabet $A = \{a, b\}$, the language denoted by the rational expression '`(a+b)*ab(a+b)*`' contains all words that contain '`ab`'.

The VAUCANSON library always needs to know on which alphabet a rational expression is defined in order to parse it. This alphabet can be indicated using option '`--alphabet=ab`' or the shorter form '`-aab`' (see subsection 2.6.1 for more details).

For instance, here is how to create an automaton that recognizes the same language as '`(a+b)*ab(a+b)*`', and make sure this automaton is equivalent to the automaton $\mathcal{A}_1$ of Figure 2.1.

```
$ vcsn-char-b exp-to-aut -aab "(a+b)*ab(a+b)*" > aut.xml
$ vcsn-char-b are-equivalent -v aut.xml a1.xml
Automata are equivalent
```

The '`-v`' is used to request a plain English output from `are-equivalent`. Without it, TAF-KIT would just set its status code. (See 2.6.2 for more details.)

**Caveat:** because VAUCANSON builds rational expressions on top of words, the Kleene star operator and the weights (introduced in subsection 2.4.3) apply to words and not letters as it is usually the case in other application. For instance '`ab*`' is the same rational expression as '`(ab)*`' for VAUCANSON, but it is different from '`a.b*`' or '`a.(b*)`'.

### 2.4.2 Empty word and null series

The default representation of the empty word (identity of the monoid) is '`1`' when using characters or pair alphabets. For instance let us try the command `expand`, that distributes concatenations over disjunctions:

```
$ vcsn-char-b expand -aab '(a+1)(1+b)'
a+ab+b+1
```

Of course if we use '`1`' as one character in the alphabet, the same symbol cannot be used for representing the empty word. VAUCANSON actually choose the first available representation of the empty word from the following list of candidate symbols: '`1`', '`e`', '`_e`', or '`eps`'.

```
$ vcsn-char-b expand -a01 '(0+e)(e+1)'
0+01+1+e
$ vcsn-char-b expand -a1e '(1+_e)(_e+e)'
1+1e+e+_e
```

For integer alphabets, the empty word is of course always '`e`'.

Similarly the symbol used to represent the null series defaults to the first representation from the following list that is compatible with the alphabet: '`0`', '`z`', '`_z`', or '`zero`'.

Section 2.6.3 shows how you can actually specify you own representation for these symbols

### 2.4.3 Weights

Weights are written in braces as in '`{3}`'.

For instance the automaton $\mathcal{C}_1$ from subsection 3.2.2 corresponds to the (weighted) rational expression '`(a+b)*.b.({2}a+{2}b)*`'.

```
$ vcsn-char-z aut-to-exp c1.xml
(a+b)*.b.({2} a+{2} b)*
```

In VAUCANSON, and even if this does not appear yet in the TAF-KIT instances, the weight semirings are not necessarily commutative: the simple case where this will occur is when an "FMP transducer" will be transformed into an automaton with weights in the semiring of rational languages over the output alphabet.

For this reason, the multiplication by a weight on the left and on the right are two distinct operations in the building of weighted rational expressions. For instance, '`{3}({2}a+b)`' and '`({2}a+b){3}`' are two distinct expressions even if they denote the same polynomial: '`{6}a + {3}b`', in the same way as '`{y}({x}a+b)`' and '`({x}a+b){y}`' are distinct expressions, which denote distinct polynomials: '`{yx}a + {y}b`' and '`{xy}a + {y}b`' respectively.

### 2.4.4 Trivial Identities

Anytime a weighted rational expression is constructed inside VAUCANSON, the following rewritings, called *trivial identities*, are automatically applied.

Here $E$ stands for any weighted rational expression, $w$ is any word, and $k$ and $h$ are weights. In these notations it should also be obvious that 0 and 1 designate the identity of the monoid (empty word) and null series, while $\{0\}$ and $\{1\}$ designate the zero and identity of the semiring (weights). Any subexpression of a form listed to the left of a '$\Rightarrow$' is rewritten as indicated on the right.

$$
\begin{array}{rcl}
E + 0 = 0 + E \Rightarrow E & E.1 = 1.E \Rightarrow E & \{k\}(\{h\}E) \Rightarrow \{kh\}E \\
E.0 = 0.E \Rightarrow 0 & \{1\}E = E\{1\} \Rightarrow E & (E\{k\})\{h\} \Rightarrow E\{kh\} \\
0^\star \Rightarrow 1 & 1\{k\} \Rightarrow \{k\}1 & (\{k\}E)\{h\} \Rightarrow \{k\}(E\{h\}) \\
\{k\}0 = 0\{k\} \Rightarrow 0 & (\{k\}1).E \Rightarrow \{k\}E & w\{k\} \Rightarrow \{k\}w \\
\{0\}E = E\{0\} \Rightarrow 0 & E.(\{k\}1) \Rightarrow E\{k\} &
\end{array}
$$

These rewriting mean that it is *impossible* for VAUCANSON to emit a rational expression such as '`({3}(0(ab)))*{4}`'. This expression is *by construction* equal to '`{4}1`'. We can verify this using `identity-exp`:

```
$ vcsn-char-z identity-exp --alphabet=ab "({3}(0(ab)))*{4}"
{4} 1
```

the command `identity-exp` does not apply any algorithm on the rational expression. Its only purpose is to read and write the rational expression using any I/O option supplied on the command-line. The trivial identities are rewritten while reading the expression.

## 2.5 Interactive Definition of Automata

The TAF-KIT command `edit-automaton` provides a textual interface to define automata interactively. The commands takes the filename of the automata to define or modify in argument. If the files does not yet exist, you should specify the alphabet of your automaton on the command line (using '`--alphabet=`' or '`-a`' as will any other command), and the file will be created when

you exit the editor. If the file does exist, the alphabet will be read from the file along with the automaton itself, and the file will be overwritten upon exit.

The interface is based on a menu of choices:

```
$ vcsn-char-b edit-automaton --alphabet=ab test.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)
```

States are numbered from 0, but transitions numbers start at 1. Also, note that weights are reported, although only `1` is valid for Boolean automata.

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

## 2.6 Command I/O options

As we said in section 2.3, each TAF-Kit command has to read its input and write its output. In this section we only cover the options that may modify the Input/Output behaviors of commands.

| long option | short | purpose | documentation |
|---|---|---|---|
| '`--alphabet`' | '`-a`' | specify the alphabet of automata or rational expressions | §2.6.1 |
| '`--alphabet1`' | '`-a`' | specify the first alphabet on transducers | §2.6.1 |
| '`--alphabet2`' | '`-A`' | specify the second alphabet on transducers | §2.6.1 |
| '`--input`' | '`-i`' | select input format for automata and rational expressions | §2.6.2 |
| '`--output`' | '`-o`' | select output format for automata and rational expressions | §2.6.2 |
| '`--parser`' | '`-p`' | fine-tune the symbols used for input and output of rational expressions and automata | §2.6.3 |

The full list of options can be obtained with *`vcsn-char-b --help`*.

### 2.6.1 Specifying alphabets

When TAF-Kit reads an XML file, there is no need to specify any other information besides the name of the file. For instance when we read '`a1.xml`' in section 2.2 and determinized this automaton, we did not have to tell TAF-Kit that the alphabet was $A = \{a, b\}$. The XML file is self-contained and already contains this information.

Here is a situation where specifying an alphabet is mandatory:

```
$ vcsn-char-b exp-to-aut aba+a
Error: alphabet should be explicitly defined using --alphabet
```

`exp-to-aut` is a command that takes a rational expression and converts it into an automaton. To be able to parse the rational expression, Vaucanson needs to know what alphabet it its using. Here there is no ways it can guess whether the alphabet is $A = \{a, b\}$ and the '`+`' is a rational operator or if it is $A = \{a, b, +\}$ and the '`+`' is just a letter. Specifying the alphabet can be done using '`--alphabet=ab`' for instance.

```
$ vcsn-char-b exp-to-aut --alphabet=ab aba+a > aut.xml
```

In practice, the long '`--alphabet=`' option can be tedious to type and we will often prefer its short equivalent '`-a`':

```
$ vcsn-char-b exp-to-aut -aab aba+a > aut.xml
```

**Character alphabets** For characters alphabets (as with the '`char`' TAF-Kit instances used in the above examples), the letters of the alphabets can be arbitrary ASCII characters, and need just to be listed after the '`--alphabet=`' or '`-a`' option.

When specifying characters alphabets, the characters ' ' (space), '`"`', '`(`', '`)`', '`'`', '`=`', and '`\`', have to be escaped with a backslash. For instance the following command will create an automaton that recognize numbers of the form '`12,456,789`', where a comma must be used as thousand separator: note how the comma must be escaped in the alphabet

```
$ d="(0+1+2+3+4+5+6+7+8+9)"
$ vcsn-char-b exp-to-aut -a'0123456789\,' "($d+$d$d+$d$d$d)(,$d$d$d)*" > numbers.xml
```

Some character alphabets are predefined. These are:

| | |
|---|---|
| '`letters`' | The lower case letters $\{a, b, \ldots, z\}$. |
| '`alpha`' | The upper and lower case letters $\{a, b, \ldots, z, A, B, \ldots, Z\}$. |
| '`digit`' | All digits $\{0, 1, \ldots, 9\}$. |
| '`ascii`' | All ASCII characters. |

This means that '`-aletters`' is an abbreviation for '`-aabcdefghijklmnopqrstuvwxyz`'. You can always get the above list of predefined alphabet by typing '`vcsn-char-b --help`'.

**Integer alphabets** Using integers alphabets, letters must be specified as signed integer (they are represented by the `int` C++ type), and should be separated by commas. For instance the following commands will construct an automaton that reads any sequence of coins of 1, 2, 5, 10, 20, or 50 cents, as long as the values are increasing.

```
$ vcsn-int-b exp-to-aut -a1,2,5,10,20,50 '1*2*5*10*20*50*' > coins.xml
```

**Pair alphabets** Pair alphabets should be specified using parentheses and commans to form pairs. For instance:

```
$ vcsn-char-int-b exp-to-aut -a'(a,1)(b,2)(a,-1)' '(a,-1)(a,1)+(b,2)' > misc.xml
```

**Transducer alphabets** Free monoid products have two alphabets, one for each monoid. The instances of TAF-Kit that handle transducers consequently support two options '`--alphabet1=`' and '`--alphabet2=`', that can be abbreviated respectively '`-a`' and '`-A`'.

### 2.6.2 Input and Output Formats

TAF-Kit can input and output several kind of objects: automata, rational expressions, words, weights and Boolean results.

**words** are always read as strings given on the command line, and written to standard output.

**automata** are read from a file whose filename specified on the command line, and output on standard output. Vaucanson can read automata in two formats: FSMXML (the default), or the textual format of FSM. It can also write automata in these formats, as well as in the '`dot`' format that can be used for graphical output.

**rational expression** are by default read as strings given on the command line, and output as strings on standard output. Alternatively rational expression can be read from an FSMXML file whose filename is given on the command line, and output in FSMXML as well.

**weight results** (such as the result of the evaluation of a word on an automaton) are simply output as strings on the standard output.

**Boolean results** (such as the result of asking whether an automaton is empty) are returned using the status code of the TAF-Kit instance, so that these commands can be used as conditions in shell scripts.

**Changing the format for automata and rational expressions** The format used to input automata and rational expressions can be controlled using the '`--input=`' and '`--output=`' options (or '`-i`', '`-o`' for short). These options control the I/O formats for both automata and rational expressions at once. So for instance using '`-ixml`' will ask TAF-Kit to read any automaton or rational expression using the FSMXML format. Because rational expression are not supported in as much formats as automata, they will be read or written as text string if an unsupported format is requested.

| values for '`-i`' or '`-o`' | format for automata | format for rational expressions |
|---|---|---|
| (none) | FSMXML | text string |
| '`xml`' | FSMXML | FSMXML |
| '`fsm`' | FSM | text string |
| '`dot`' (for output only) | dot | text string |

For instance to convert an automaton from XML to dot[1], we would use:

```
$ vcsn-char-b identity -odot aut.xml > aut.dot
```

**Verbose Boolean results** As said above, boolean results are returned using the program's status code using the Unix convention (that is 0 for *true* and any other value for *false*). The shell makes this value available in the '`$?`' variable. The TAF-Kit option '`--verbose`' or '`-v`' can be used to request an English interpretation of this value.

```
$ vcsn-int-b is-empty coins.xml
$ echo $?
1
$ vcsn-int-b is-empty -v coins.xml
Input is not empty
```

### 2.6.3 Specifying writing data

Section 2.4.2 showed how the empty word and null series can have different representations depending on the alphabet. Vaucanson actually allows other symbols used in rational expression to be changed to arbitrary strings.

Here is the list of named symbols with their meaning and default values:

| symbol | meaning | default value(s) |
|---|---|---|
| '`OPAR`' | group start | '`(`' |
| '`CPAR`' | group end | '`)`' |
| '`PLUS`' | disjunction (additive law of the series) | '`+`' |
| '`TIMES`' | concatenation (multiplicative law of the series) | '`.`' |
| '`STAR`' | Kleene star | '`*`' |
| '`ONE`' | empty word (identity of the monoid) | '`1`','`e`','`_e`','`eps`' |
| '`ZERO`' | null series | '`0`','`z`','`_z`','`zero`' |
| '`OWEIGHT`' | weight start | '`{`' |
| '`CWEIGHT`' | weight end | '`}`' |
| '`SPACE`' | space characters (to ignore) | ' ' |

The '`--parser=`' can be used to change the value of the above tokens. Each of them must be defined as a non-empty string. TAF-Kit will check that these tokens do not collide. For instance to use $\{(,)\}$ as alphabet, we should obviously rename the '`OPAR`' and '`CPAR`' tokens.

The following command creates an automaton that recognizes the words '`()`', '`(())`', '`(()())`', '`((()()))`', etc.

[1] dot files can be processed using the GraphViz package

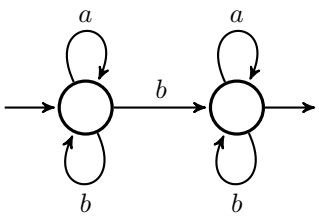



Figure 2.2: $\mathcal{B}_1$: When defined over the Boolean semiring $\langle \mathbb{B}, \vee, \wedge \rangle$, $\mathcal{B}_1$ accepts words with at least one $b$. When defined over the integer semiring $\langle \mathbb{Z}, +, \times \rangle$, $\mathcal{B}_1$ counts the number of $b$ in a word.

```
$ vcsn-char-b -a'' --parser='OPAR=[ CPAR=]' exp-to-aut '([()]*)' >parens.xml
```

The values of these symbols, which we call the *writing data*, are stored in the XML file, so there is no need to specify them again when working from a file.

```
$ vcsn-char-b aut-to-exp parens.xml
(.(.[).(]*.).)+(.)
$ vcsn-char-b eval parens.xml '(()())'
1
```

**Overwriting the writing data** When TAF-Kit reads an automaton or a rational expression from an XML file (that contains writing data) or from the internal pipe, it does not need additional information to read its input. However the '`--parser=`' option can still be used to modify the way the object will be *output*.

Here is an example where we create a rational expression over the alphabet $\{(,)\}$ using '`[`' and '`]`' for grouping, and store it into the file '`p.xml`'. We can then convert this file back into a string using either the original writing data that were stored in the file, or overwriting these data with different ones (here using '`<`' and '`>`' for grouping).

```
$ vcsn-char-b -a'' --parser='OPAR=[ CPAR=]' identity-exp -oxml '([()]*)' >p.xml
$ vcsn-char-b identity-exp -ixml p.xml
(.[(.)]*.)
$ vcsn-char-b --parser='OPAR=< CPAR=>' identity-exp -ixml p.xml
(.<(.)>*.)
```

## 2.7 An example of $\mathbb{Z}$-automaton

This part shows some uses of the program `vcsn-char-z`.

### 2.7.1 Counting '`b`'s

Let's consider $\mathcal{B}_1$ for Figure 2.2: a $\mathbb{Z}$-automaton, *i.e.* an automaton whose label's weights are in $\mathbb{Z}$. 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 '`abbb`' and '`abab`':

```
$ vcsn-char-z eval b1.xml abbb
3
$ vcsn-char-z eval b1.xml 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 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 info b1.xml
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
```

# Chapter 3

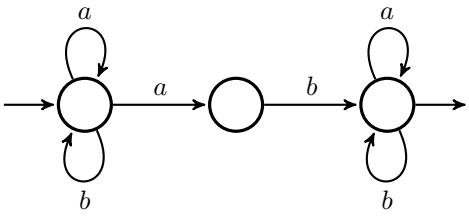
# Automaton Repository

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

## 3.1 $\mathbb{B}$ automata

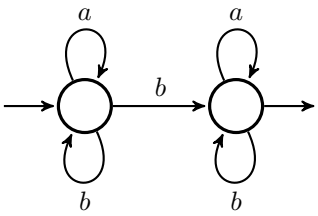
The following automata are predefined files that can be loaded with `vcsn-char-b`.

### 3.1.1 'a1.xml' ($\mathcal{A}_1$)



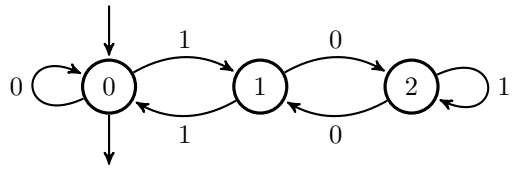

The automaton $\mathcal{A}_1$, defined over the alphabet $A = \{a, b\}$, recognizes any word of $A^\star$ that contains $ab$.

### 3.1.2 'b1.xml' ($\mathcal{B}_1$)




The Boolean automaton $\mathcal{B}_1$ accepts words with at least one $b$.
See also subsection 3.2.1 where $\mathcal{B}_1$ is defined over $\langle \mathbb{Z}, +, \times \rangle$.

### 3.1.3 'div3base2.xml'

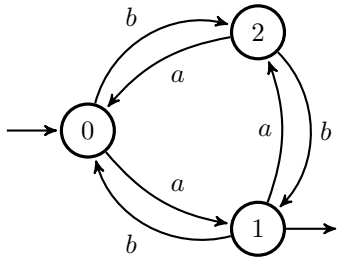



This deterministic automaton is defined on the alphabet $A = \{0, 1\}$ and recognizes the base-2 representations of all the integers that are divisible by 3.

The source code distribution of VAUCANSON actually comes with a programs called `divkbaseb` that can generate a divisor by any integer $k$ in any base $b$. The program is located in '`data/automata/char-b/divkbas` in the source tree, and take $k$ and $b$ as arguments.

In the general case the binary digits should be read from left to right. In the case of '`div3base2.xml`' the digits can also be read from right to left because the automaton is symmetric.

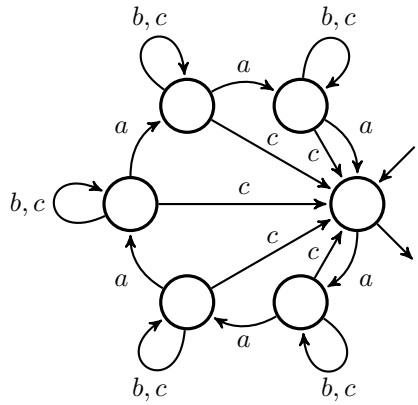
### 3.1.4 'double-3-1.xml'




This deterministic automaton has three states organized as a double ring. It can read $a$ in one direction and $b$ in the other direction.

The program `data/automata/char-b/double_ring` can be used to generate a ring of any number of states. It takes that number of states as first argument, and the list of initial states in the remaining arguments. The above example uses 3 states and state 1 as initial state, hence the name '`double-3-1.xml`'.

### 3.1.5 'ladybird-6.xml'



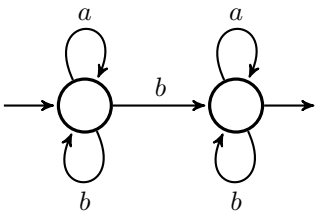

A non-deterministic automaton of 6 states whose determinized version (using the powerset construction) has $2^6$ states. We call it ladybird simply because its drawing looks like a *Coccinella*.

The `data/automata/char-b/ladybird` program can be used to produce ladybirds of any size, just pass the number of states $n$ as first argument. We like to use ladybirds for benchmarking because their determinization always produce the worst case with $2^n$ states.

## 3.2 $\mathbb{Z}$ automata

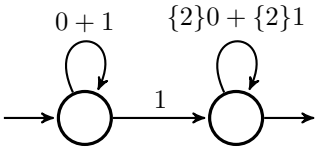
The following automata are predefined files that can be loaded with `vcsn-char-z`.

### 3.2.1 'b1.xml' ($\mathcal{B}_1$)



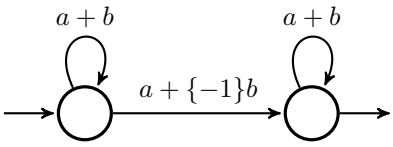

This automaton has the same structure as the one from subsection 3.1.2 but a different type. When defined over the $\langle \mathbb{Z}, +, \times \rangle$ semiring, this automaton is counting the number of $b$ in a word.

### 3.2.2 'c1.xml' ($\mathcal{C}_1$)




This automaton converts a binary representation into an integer. For instance the word '101010' is associated to the weight 42. Note that the alphabet is $A = \{0, 1\}$ while the weights are taken in $\mathbb{Z}$. Because weights are always specified in braces, there is no confusion possible.

### 3.2.3 'd1.xml' ($\mathcal{D}_1$)

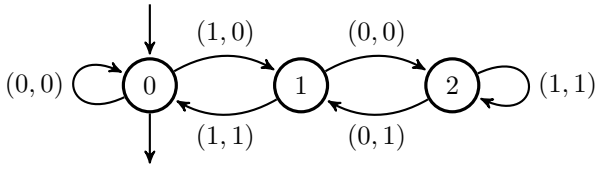



This automaton defined on the alphabet $A = \{a, b\}$ computes the difference between the number of $a$ and $b$ in a word.

## 3.3 $\mathbb{B}$ FMP

The following transducers are predefined files that can be loaded with `vcsn-char-fmp-b`.

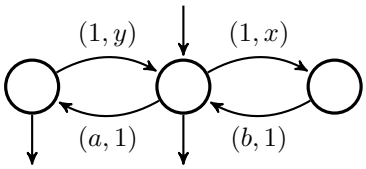
### 3.3.1 'quot3base2.xml'




A transducer computing the quotient by 3 of a binary number, read from left to right.

The program 'data/automata/char-fmp-b/quotkbaseb', in the source tree, can be used to construct such a divisor by $k$ in any base $b$.

### 3.3.2 't1.xml'




### 3.3.3 'u1.xml'