# The VAUCANSON TAF-KIT 1.0
## User's Manual

The VAUCANSON GROUP

July 28th, 2006

# Contents

# Introduction

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

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

1. a *general purpose* software,

2. a software that allows a programming style natural to computer scientists who work with automata and transducers,

3. an open and free software.

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

---

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

# Chapter 1

# Installation

## 1.1 Getting Vaucanson

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

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

## 1.2 Building Vaucanson

The following commands build and install the platform:

```
# cd vaucanson-1.0
```

Then:

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

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

---

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

# Chapter 2

# The Vaucanson toolkit

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

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

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

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

It is planned that a forthcoming version will include also:

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

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

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

## 2.1 Boolean automata

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

### 2.1.1 First Contacts

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

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

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

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

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

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

and should be understood, or analyzed, as follows.

1. `vcsn-b` is the call to a `shell` command that will launch a VAUCANSON function. `vcsn-b` has 2 arguments, the first one being the `function` which will be launched, the second being the `automaton` that is the input argument of the `function`.

2. `determinize` is, as just said, a VAUCANSON function. And as it can easily be guessed, `determinize` takes an `automaton` as argument, performs the subset construction on it and outputs the result on the standard output.

3. 'a1.xml' is the description of an automaton — of the automaton of Section 3.1.1 indeed — in an XML format that is understood[1] by VAUCANSON. Which means that this file must exist before the line is executed. The 'data/automata' directory provides a number of XML files for examples of automata, a number of programs that produce the XML files for automata whose definition depend upon some variables and the TAF-KIT itself allow to define automata and thus to produce the corresponding XML files (cf. below).

4. `>` 'a1det.xml' puts the result of `determinize` 'a1.xml', that is, the XML file which describes the determinized automaton of $\mathcal{A}_1$ into the file 'a1det.xml'.

As a more elaborate example, consider the following command

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

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

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

---

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

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

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

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

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

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

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

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

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

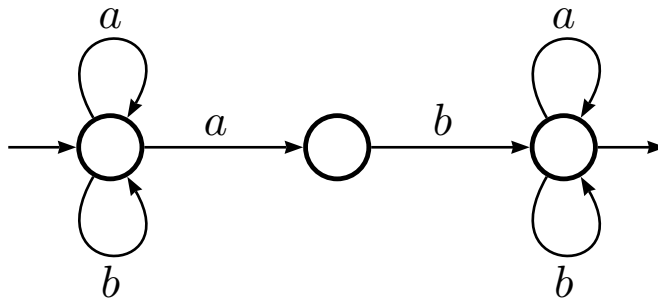The whole list of supported commands is available via '--list-commands'.

### 2.1.2 A first example

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

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

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

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

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 $\mathcal{A}_1$

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

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

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

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

```
# vcsn-b dot-dump a1.xml >a1.dot
```
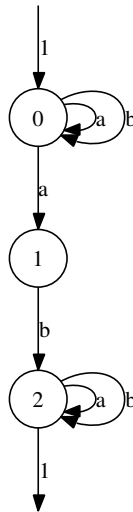
A { 3 states, 6 transitions, #I = 1, #T = 1 }

## Determinization of $\mathcal{A}_1$

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

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

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

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

Or use dotty to visualize it:
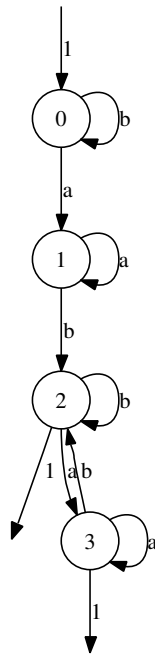
```
# vcsn-b dot-dump a1det.xml >a1det.dot
```
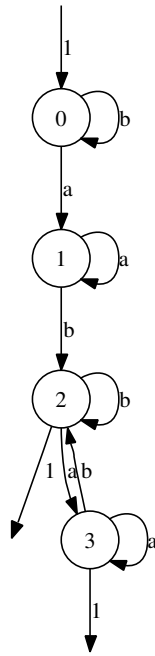
A { 4 states, 8 transitions, #I = 1, #T = 2 }

**Minimizing**

The minimal automaton can be computed the same way:

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

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



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

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

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

**Evaluation**

To evaluate whether a word is accepted:

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

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

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

### 2.1.3 Rational expressions and Boolean automata

VAUCANSON provides functions to manipulate rational expressions associated to Boolean automata. For instance, computing the language recognized by a Boolean automaton can be done using `aut-to-exp`:
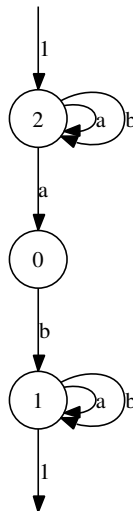
```
# vcsn-b aut-to-exp a1.xml
(a+b)*.a.b.(a+b)*

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

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

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

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



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

## 2.1.4   Available functions

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

```
# vcsn-b --list-commands
List of available commands:
 * Input/output work:
   - define-automaton file: Define an automaton from scratch.
   - display aut: Display 'aut'.
   - dot-dump aut: Dump dot output of 'aut'.
   - dump-automaton file: Dump a predefined automaton.
   - edit-automaton file: Edit an existing automaton.
   - identity aut: Return 'aut'.
   - info aut: Print useful infos about 'aut'.
   - list-automata: List predefined automata.
 * Tests and evaluation on automata:
   - are-isomorphic aut1 aut2: Return whether 'aut1' and 'aut2' are isomorphic.
   - eval aut word: Evaluate 'word' on 'aut'.
   - is-ambiguous aut: Return whether 'aut' is ambiguous.
   - is-complete aut: Return whether 'aut' is complete.
   - is-deterministic aut: Return whether 'aut' is deterministic.
   - is-empty aut: Return whether trimed 'aut' is empty.
   - is-realtime aut: Return whether 'aut' is realtime.
 * Generic algorithms for automata:
   - accessible aut: Give the maximal accessible subautomaton of 'aut'.
   - eps-removal aut: Give 'aut' closed over epsilon transitions.
   - co-accessible aut: Give the maximal coaccessible subautomaton of 'aut'.
   - complete aut: Give the complete version of 'aut'.
   - concatenate aut1 aut2: Concatenate 'aut1' and 'aut2'.
   - power aut n: Give the power of 'aut' by 'n'.
   - product aut1 aut2: Give the product of 'aut1' by 'aut2'.
   - quotient aut: Give the quotient of 'aut'.
   - realtime aut: Give the realtime version of 'aut'.
   - sum aut1 aut2: Give the sum of 'aut1' and 'aut2'.
   - transpose aut: Transpose the automaton 'aut'.
   - trim aut: Trim the automaton 'aut'.
 * Boolean automaton specific algorithms:
   - complement aut: Complement 'aut'.
   - determinize aut: Give the determinized automaton of 'aut'.
   - minimize aut: Give the minimized of 'aut' (Hopcroft algorithm).
   - minimize-moore aut: Give the minimized of 'aut' (Moore algorithm).
 * Conversion between automata and expressions:
   - aut-to-exp aut: Give the automaton associated to 'aut'.
   - derived-term exp: Use derivative to compute the automaton of 'exp'.
   - exp-to-aut exp: Alias of 'stardard'.
   - expand exp: Expand 'exp'.
   - standard exp: Give the standard automaton of 'exp'.
   - thompson exp: Give the Thompson automaton of 'exp'.
```
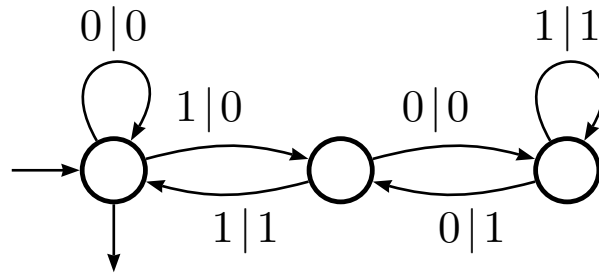
The transducer computing the quotient by 3 of a binary number.

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

## 2.2 Transducers

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

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

In a forthcoming release, TAF-KIT will provide:

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

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

### 2.2.1 Example

To experiment with transducers, we will use $\mathcal{T}_1$, described in Figure 2.2, and part of the automaton library (Section 3.3.1).
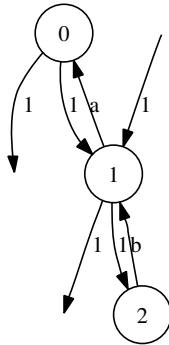
**Domain**

The transducer $T$ only accepts binary numbers divisible by 3.

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

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

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

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

## 2.2.2 Available functions

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

```
# vcsn-tdc --list-commands
List of available commands:
 * Input/output work:
   - define-automaton file: Define an automaton from scratch.
   - display aut: Display 'aut'.
   - dot-dump aut: Dump dot output of 'aut'.
   - dump-automaton file: Dump a predefined automaton.
   - edit-automaton file: Edit an existing automaton.
   - identity aut: Return 'aut'.
   - info aut: Print useful infos about 'aut'.
   - list-automata: List predefined automata.
 * Tests and evaluation on transducers:
   - are-isomorphic aut1 aut2: Test if 'aut1' and 'aut2' are isomorphic.
   - is-empty aut: Test if 'aut' realizes the empty relation.
   - is-sub-normalized aut: Test if 'aut' is sub-normalized.
 * Generic algorithm for transducers:
   - eps-removal aut: epsilon-removal algorithm.
   - domain aut: Give the automaton that accepts all inputs accepted by 'aut'.
   - eval aut exp: Give the evaluation of 'exp' against 'aut'.
   - eval-aut aut: Evaluate the language described by the Boolean automaton
       'aut2' on the transducer 'aut1'.
   - image aut: Give an automaton that accepts all output produced by 'aut'.
   - trim aut: Trim transducer 'aut'.
 * Algorithms for transducers:
   - sub-normalize aut: Give the sub-normalized transducer of 'aut'.
   - composition-cover aut: Outsplitting.
   - composition-co-cover aut: Insplitting.
   - compose aut1 aut2: Compose 'aut1' and 'aut2', two (sub-)normalized
       transducers.
   - u-compose aut1 aut2: Compose 'aut1' and 'aut2', two Boolean transducers,
       preserve the number of path.
   - to-rt aut: Give the equivalent realtime transducer of 'aut'.
   - intersection aut: Transform a Boolean automaton in a fmp transducer by
       creating, for each word, a pair containing twice this word.
```
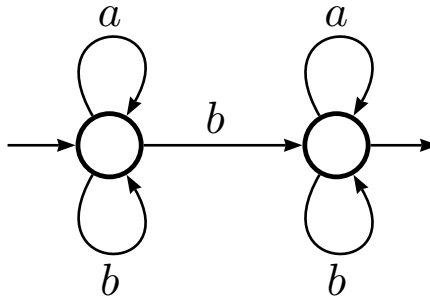
Considered without weight, $\mathcal{B}_1$ accepts words with a 'b'. With weights, it counts the number of 'b's.

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

## 2.3 $\mathbb{Z}$-Automata

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

Again, we will toy with some of the automata provided by `vcsn-z`, see 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-z dump-automaton b1 | vcsn-z eval - 'abbb'
3

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

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

**Power**

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

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

This is implemented by the `power` function:

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

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

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

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

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

**Quotient**

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

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

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

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

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

## 2.3.2   Available functions

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

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

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

# Chapter 3

# 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.
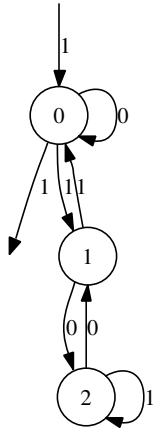
## 3.1 Boolean Automata

### 3.1.1 a1
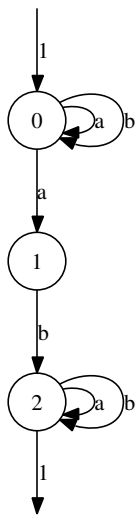


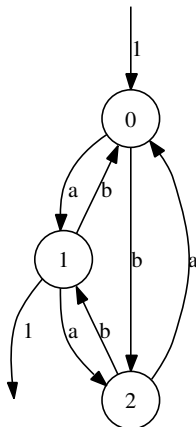A { 3 states, 6 transitions, #I = 1, #T = 1 }

### 3.1.2 b1
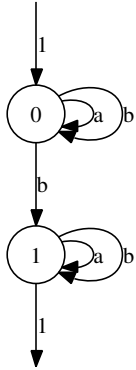


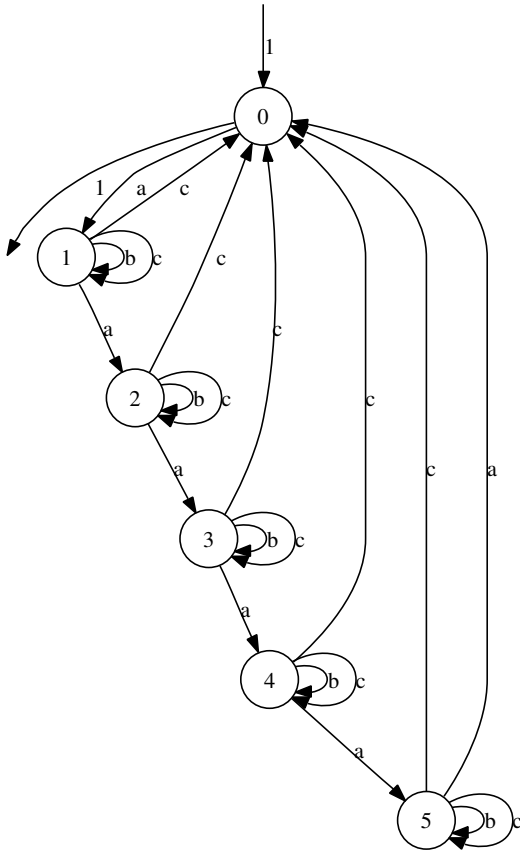A { 2 states, 5 transitions, #I = 1, #T = 1 }

### 3.1.3 div3base2



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

### 3.1.4 double-3-1
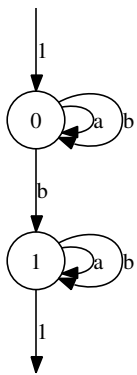


A { 3 states, 6 transitions, #I = 1, #T = 1 }

### 3.1.5 ladybird-6



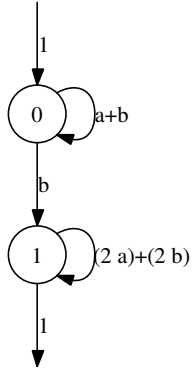A { 6 states, 21 transitions, #I = 1, #T = 1 }

## 3.2 ℤ-Automata

### 3.2.1 b1



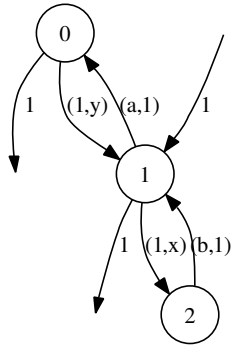A { 2 states, 5 transitions, #I = 1, #T = 1 }

### 3.2.2   c1



1

0    a+b

b

1    (2 a)+(2 b)

1

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

## 3.3    Transducers

### 3.3.1   t1



0

1   (1,y) (a,1)   1

1

1   (1,x) (b,1)

2

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

### 3.3.2   u1



1

0

(x,1)   (1,v)   1

1

1 (y,1) (1,u)

2

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