VAUCANSON 1.4
TAF-KIT Documentation

ABOUT THIS DOCUMENT

This document is based on a first draft of a VAUCANSON *User's Manual written by Alexandre Duret-Lutz in April 2009 for the version 1.2.95a.*

It is first meant to describe as precisely as possible the specifications of TAF-KIT *within* VAUCANSON *1.4. It is a working document and should help to finalizing* TAF-KIT*. The discrepancies, in names and functionalities — there should not be many — will raise final discussions and decisions. When* VAUCANSON *1.4 will be released, the same document, with the adequate corrections, will serve as a rather complete user's manual for* TAF-KIT *which will be the only documented part of that version.*

The corresponding version of the VAUCANSON *platform —* VAUCANSON *1.4— is meant to be the last one of a first phase of this project. Officially starting January 2011, a second phase will be engaged, with different interface specification. It will give rise to versions* VAUCANSON *2.x.y.*

J. S.
December 2010

Comment for the VAUCANSON *Group* (101203): *Comme indiqué en bas de page, nous sommes dans la phase* working document*. Dans cette version du document, j'ai essayé de décrire l'état de* VAUCANSON*, et les commentaires que nous en avons fait lors de la réunion du 02/12/10.*

Je suis sûr de ne pas avoir tout noté ou retenu. A compléter et corriger.

Table of contents

Introduction **5**

0 Administrativia **7**

0.1 Getting Vaucanson 7
0.2 Licensing 7
0.3 Prerequisites 7
0.4 Building Vaucanson 8
0.4.1 Apple OS Specifics 8

1 Presentation of TAF-Kit **10**

1.1 First contact 10
1.2 TAF-Kit organisation 13
1.2.1 Automata types 13
1.2.2 TAF-Kit instances 14
1.2.3 Command options 14
1.2.4 TAF-Kit's modus operandi 15
1.2.5 Automata repository and factory 16
1.3 Option specifications 17
1.3.1 Information options 18
1.3.2 Alphabet specification 18
1.3.3 Input and output formats 22
1.3.4 Benchmarking options 25
1.4 The writing of rational expressions 27
1.4.1 The definition of expressions 27
1.4.2 Parsing strings into expressions 28
1.4.3 Parser parametrization 32
1.5 TAF-Kit IO functions 36
1.5.0 Data file location 36
1.5.1 `data` 37
1.5.2 `cat` 37
1.5.3 `cat-E` 37
1.5.4 `display` 37
1.5.5 `edit` 38

2 Specification of functions on automata and rational expressions **40**

2.1 General automata and rational expressions 42
2.1.1 Graph functions 43
2.1.2 Transformations of automata 44
2.1.3 Operations on automata 46
2.1.4 Operations on behaviour of automata 49
2.1.5 Automata and expressions; operations on expressions 50

2.2 Weighted automata and expressions over free monoids 54
2.2.1 Properties and transformations of automata 54
2.2.2 Behaviour of automata 57
2.2.3 From expressions to automata 58
2.2.4 Operations on automata and their behaviour 58
2.3 Automata and rational expressions on free monoids with weights in a field . . 60
2.3.1 Operations on automata 60
2.3.2 Operations on expressions 61
2.4 Boolean automata and rational expressions on free monoids 63
2.4.1 Operations on automata 63
2.4.2 Behaviour of automata 66
2.4.3 Operations on behaviours of automata 67
2.4.4 Operations on expressions 67
2.5 Weighted automata over a product of two free monoids 70
2.5.1 Transformations of transducers 71
2.5.2 Operations on transducers 73
2.5.3 Operations on behaviours of transducers 74
2.6 Weighted automata on free monoids over alphabets of pairs 76
2.6.1 Transformations of automata 76

A Automata repository and factory **77**
A.1 $\mathbb{B}$-automata 77
A.1.1 Repository 77
A.1.2 Factory 77
A.2 $\mathbb{Z}$-automata 79
A.2.1 Repository 79
A.2.2 Factory 80
A.3 $\mathbb{B}$-*fmp-transducers* 80
A.3.1 Repository 80
A.3.2 Factory 81

B FSM XML,
an `XML` format for automata **82**
1 The root tag 83
2 The value type tags 83
3 The rational (regular) expressions 86
4 The automata 88

C Algorithm specification,
description and discussion **91**
C.1 General automata and rational expressions functions 91
C.1.1 Graph functions 91
C.1.2 Transformations of automata 91

C.1.3 Operations on automata 93
C.1.4 From automata to expressions 95
C.2 Weighted automata and rational expressions over free monoids 96
C.2.1 Transformations of automata 96
C.2.2 Behaviour of automata 97
C.3 Automata and rational expressions on free monoids with weights in a field . . 97
C.3.1 Operations on automata 97
C.4 Boolean automata and rational expressions on free monoids 98
C.4.1 Operations on automata 98
C.4.3 Operations on expressions 98
C.5 Weighted automata over a product of two free monoids 99
C.5.2 Operations on transducers 99

Bibilography **101**

Index **102**

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 automata on products of two free monoids— also known as *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.

It is coupled with some other modules:

An `XML` format for automata and expressions, called FSM XML. This format aims at being an interchange format for automata and thus at making possible, and hopefully easy, the communication between various programs that input or output automata. So far, this format is used as the normal, and default, input and output format for TAF-KIT.

A graphic user interface called VGI, especially dedicated to VAUCANSON so far. It allows to describe automata and to visualize the result of operation on automata in a graphical way. All functions defined in TAF-KIT may be called via the menu of VGI.

Ideally, a user's manual for VAUCANSON should document all of these components. We decided not to do so, not so much because it is a lot of work, but also as this work would not be so useful.

After several years of hard and complex developments, the evolution and progress of the VAUCANSON platform are now stuck and we have reached the conclusion that we have to undertake a thorough revision of the VAUCANSON library that will most probably change its interface and the one of the associated API. These new developments will give rise to a new series of versions of VAUCANSON, coined VAUCANSON 2.x.

On the other hand, there will be a TAF-KIT for these future versions of VAUCANSON, whose functionalities will include all those of the present one and whose interface will essentially be the same as the present one as well. TAF-KIT VAUCANSON 1.4 will serve as a landmark for both functionalities and performance of the first version of VAUCANSON. It will be the only documented part of VAUCANSON 1.4.

Chapter 0

Administrativia

0.1 Getting Vaucanson

All the latest versions of the Vaucanson platform can be downloaded from `http://vaucanson.lrde.epita.fr/`

Please note this manual is not meant to be backward compatible with Vaucanson versions prior to 1.4.

0.2 Licensing

Vaucanson 1.4 is a free software released under the GNU General Public Licence version 2. If you are unfamiliar with this license, please refer to `http://www.gnu.org/licenses/gpl-2.0.txt` (a copy of this license is included in each copy of Vaucanson in the file *COPYING*).

Beware that the license for the next versions of Vaucanson will probably be different (although Vaucanson will stay an open and free software).

0.3 Prerequisites

`C++` compiler `G++ 4.x` where `x` < 5.

`XML` The `XML` I/O system is based on the use of the Apache `Xerces C++` library version 2.7+ (`http://apache.org/xerces-c/`). (On Ubuntu/Debian, install the following packages: `libxerces27` and `libxerces28-dev`, or `libxerces28` and `libxerces28-dev`).

`Graphviz` The display of automata is made using AT&T `GraphViz` library (On Ubuntu/Debian, install the following package: `graphviz`).

`Boost` `Boost` provides free peer-reviewed portable `C++` source libraries (On Ubuntu/Debian, install the following packages: `libboost-dev`, `libboost-serialization-dev`, `libboost-graph`, `libboost-graph-dev`). Vaucanson is compatible with `Boost` versions >= `1.34`. It shall be noted that with `Boost 1.44`, a special flag must be given to the compiler through the configure file: `CPPFLAGS='-DBOOST_SPIRIT_USE_OLD_NAMESPACE'`.

`Ncurses` needed for building TAF-Kit (On Ubuntu/Debian, install the following packages: `libncurses5`, `libncurses-dev`).

0.4 Building Vaucanson

Detailed information is provided in both `INSTALL` and `doc/README.txt` files. The following installation commands will install VAUCANSON in '`/usr/local`'.

```
$ cd vaucanson-1.4
$ ./configure
$ make
$ sudo make install
```

Depending on your architecture, both `Boost` and `Xerces` might be located in non-standard directories. If you are unsure of the location of your libraries, you may type in your shell:

```
$ whereis boost
```

These commands will return the paths to `Boost` headers. You can then specify this directories to the `configure` file through the use of two environment variables: `CPPFLAGS` for the header files and `LDFLAGS` for the library files. For instance, if your `Boost` headers are located in '`/usr/user_name/home/my_path_to_boost/include`' and its library files in '`/usr/user_name/home/my_path_to_boost/lib`' you will use the following configure line:

```
$ ./configure
CPPFLAGS='-I/usr/user_name/home/my_path_to_boost/include'
LDFLAGS='/usr/user_name/home/my_path_to_boost/lib'
```

If you did not install Vaucanson but simply compiled it, you will find the taf-kit binaries available in the directory '`vaucanson-1.4/taf-kit/tests/`' (This directory contains wrapper around the real TAF-Kit programs from '`vaucanson-1.4/taf-kit/src/`' that enable them to run locally).

0.4.1 Apple OS Specifics

In this section, we go through the installation process of Vaucanson and its dependencies on Apple systems as it is less straightforward than onto other Linux systems.

First, you should ensure your Operating System is up-to-date before going through the rest of the installation process.

Second, you will need the Macports software (see `http://www.macports.org/`) installed on your computer. A complete guide to its installation is available from `http://guide.macports.org/`. If you have already installed Macports, please ensure you have the latest version installed by synchronising your local port tree with the global Macports ports.

```
$ sudo port selfupdate
```

There is now three libraries you will need to install in order to build VAUCANSON (see Prerequisite for details): `Boost`, `Xerces`, `C++` and `Ncurses`. Executing each of the following commands will take a while:

```
$ sudo port install ncurses
...
$ sudo port install boost
...
$ sudo port install xercesc
...
$
```

We can now proceed with the Vaucanson installation process. By default, Macports will install each of the previous software to a non-standard directory on your computer: `/opt/local`. Therefore, in order to build VAUCANSON, you will have to specify this directory to the configure file:

```
$ ./configure CPPFLAGS='-I/opt/local/include' LDFLAGS='-L/opt/local/lib'
```

Moreover, if you are running with `Boost` versions >= 1.44, you need to specify an additional option to the configure:

```
$ ./configure CPPFLAGS='-I/opt/local/include
-DBOOST_SPIRIT_USE_OLD_NAMESPACE' LDFLAGS='-L/opt/local/lib'
```

You can now complete the installation by typing:

```
$ make
$ sudo make install
```

Comment for the VAUCANSON *Group* (101203): *A compléter, bien sûr. Mais c'est déjà en progrès très net, grâce à l'apport d'Alex.*

Chapter 1

Presentation of TAF-Kit

TAF-Kit stands for *Typed Automata Function Kit*; it is a *command-line interface* to Vaucanson. As stated in the introduction, the Vaucanson platform is dedicated to the computation of, and with, finite automata, where 'finite automata' means *weighted automata* over *a priori arbitrary monoids.*

In the static generic programming paradigm used in the Vaucanson library, the *types* of the automata that are treated have to be known at compile time. TAF-Kit, which is a set of programs that should be called from the `shell` and that can be used to chain operations on automata, has therefore been compiled for several predefined types of automata. It thus allows to use *already programmed* functions on automata of *predefined types.* TAF-Kit gives a *restricted access* to Vaucanson functionalities, but it is a *direct access*, without any need of programming. A basic knowledge of Unix command syntax only is necessary to make use of TAF-Kit.

In this chapter, we first give a series of examples of commands in the case of 'classical automata'. We then present the overall organisation of TAF-Kit, with the list of possible instances and options. The following section describes the syntax of options that help define the behaviour of the commands whereas the fourth section describes the syntax of rational (that is, regular) expressions within Vaucanson. The final section lists the input–output commands of TAF-Kit; all other commands are presented in the next chapter.

1.1 First contact

Let us first suppose that Vaucanson is fully installed (as explained in Section 0.4).[1] Any of the following commands could be typed and their results observed.

We describe now (some of) the functions of the instance of TAF-Kit which deals with 'classical automata', that is, *Boolean automata* over a free monoid whose generators are *characters.* These functions are called by the `vcsn-char-b` command.

To begin with, we have to deal with an automaton of the correct type. There are several means to build or define such an automaton, but the most direct way is to use one of those

[1]If Vaucanson is only compiled without being installed, one should first go to the '`vaucanson-1.4/taf-kit/tests/`' directory by a `cd` command, and type '`./vcsn-char-b`' instead of '`vcsn-char-b`' for each of the following commands.

whose definition comes with TAF-KIT. We choose the automaton $\mathcal{A}_1$ shown at Figure 1.1 and whose description is contained in the XML file '`a1.xml`'.

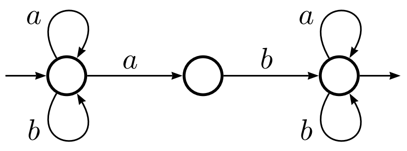


Figure 1.1: The Boolean automaton $\mathcal{A}_1$ over $\{a, b\}^*$.

The first command `data` 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 data a1.xml
States: 3
Transitions: 6
Initial states: 1
Final states: 1
```

This automaton '`a1.xml`' can also be displayed with the command `display`:[2]

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

The displayed automaton won't have a layout as pretty as in Figure 1.1, but it represents the same automaton nonetheless.

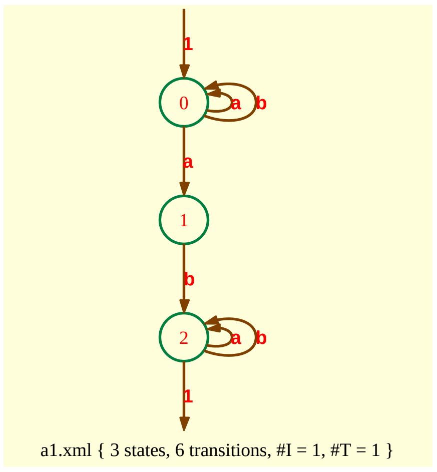


Figure 1.2: Result of the command `vcsn-char-b display a1.xml`

The command `aut-to-exp` outputs a rational expression which denotes the language accepted by $\mathcal{A}_1$. The command `eval` tells whether a word belongs to that language (answer with `1` = yes, or `0` = no).

[2]If the `GraphViz` package is installed (see Section 0.3).

```
$ vcsn-char-b aut-to-exp a1.xml
(a+b)*.a.b.(a+b)*
$ vcsn-char-b eval a1.xml 'babab'
1
$
```

The automaton $\mathcal{A}_1$ is not deterministic and the `determinize` command will compute its determinisation. As most TAF-Kit commands, `determinize` produces its output (an `XML` file representing the automaton) on the standard output which would hardly be of interest. The normal usage is to divert the output by means of a `shell` redirection to a file for subsequent computation with other commands.

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

The file '`a1det.xml`' has been created into the current directory while '`a1.xml`' is a file that is predefined in Vaucanson's predefined automata repository. We can call the command `data` on either files using the same syntax because TAF-Kit will look for automata in both places.

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

```
$ vcsn-char-b determinize a1.xml | vcsn-char-b data -
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 is called *internal* because the pipe logic is taken care of by TAF-Kit itself, and 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 an `XML` file 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 \| data -
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.*

1.2 TAF-Kit organisation

TAF-Kit is indeed *one program*, and this same program is *compiled* for different types of automata. The result of each compilation yields a command (with a distinct name) which can be called from the shell. As we have seen in the preceding examples, every such command essentially takes two arguments: the first determines a function and the second an automaton which is the operand for the function.

1.2.1 Automata types

A (finite) automaton is a (finite) directed graph, labelled by *polynomials* in $\mathbb{K}\langle M\rangle$, that is, by (finite) *linear combinations* of elements of a *monoid* M with coefficients in a *semiring* $\mathbb{K}$. The *type of an automaton* is thus entirely determined (in Vaucanson 1.4[3]) by the specification of $\mathbb{K}$ and of the type of M.

1.2.1.1 Semirings

The semirings that are instanciated in TAF-Kit 1.4 are shown in Table 1.1. All these semirings are 'numerical' in the sense their elements are implemented as numbers, but for the rationals: `float` for $\mathbb{R}$, `bool` for $\mathbb{B}$, `int` for the others. The rationals are pairs of integers and implemented as pairs of an `int` and an `unsigned`. They all are *commutative* semirings.

semiring	mathematical symbol	suffix in TAF-Kit
Boolean semiring	$\mathbb{B} = \langle\, \mathbb{B}, \vee, \wedge \,\rangle$	'`-b`'
ring of integers	$\mathbb{Z} = \langle\, \mathbb{Z}, +, \times \,\rangle$	'`-z`'
field of reals	$\mathbb{R} = \langle\, \mathbb{R}, +, \times \,\rangle$	'`-r`'
field of rationals	$\mathbb{Q} = \langle\, \mathbb{Q}, +, \times \,\rangle$	'`-q`'
two element field	$\mathbb{F}_2 = \langle\, \{0,1\}, +, \times \,\rangle$ (with $1+1=0$)	'`-f2`'
min tropical semiring	$\mathbb{Z}\mathrm{min} = \langle\, \mathbb{Z}, \mathsf{min}, + \,\rangle$	'`-zmin`'
max tropical semiring	$\mathbb{Z}\mathrm{max} = \langle\, \mathbb{Z}, \mathsf{max}, + \,\rangle$	'`-zmax`'

Table 1.1: The semirings implemented in Vaucanson TAF-Kit 1.4

1.2.1.2 Monoids

The monoids instanciated in TAF-Kit 1.4 are the *free monoids* and the *direct products of (two) free monoids*. A free monoid is completely determined by the set of generators, called *alphabet*. At compile time however, it is not necessary to know the alphabet itself: the *type* of its elements, the *letters*, will suffice. Thus, for TAF-Kit, the type of letters of one alphabet for a free monoid, of two alphabets for a direct product of two free monoids has to be defined. In TAF-Kit 1.4, the following types of letters are considered:

1. the *simple* letters, which may be *characters*: `char`, or *integers*: `int`;
2. *pairs* of simple letters.

[3]We add this precision as in the next version Vaucanson 2, the 'kind' of labels will also be a criterion in the definition of the (programming) type of an automaton.

The combinations that are instanciated in TAF-Kit 1.4 is shown in Table 1.2.

letter types	free monoids	free monoid products
characters	`char`	`char-fmp`
integers	`int`	`int-fmp`
pair of characters	`char-char`	
pair of integers	`int-int`	
pair of character and integer	`char-int`	

Table 1.2: The monoids implemented in Vaucanson TAF-Kit 1.4

1.2.2 TAF-Kit instances

As the consequence of the preceding subsection, the type of an automaton is determined by the following three data:

1. the type of the weight semiring;
2. the fact that the monoid is either a free monoid or a product of two free monoids.
3. the type of the letters that generate the free monoid(s).

Not all possible combinations derived from the types of semiring and free monoid listed above are instanciated (it would amount to over 70 possibilities — even if one restricts oneself to the same type for the input and output monoids in transducers). In Vaucanson 1.4, 'only' 18 combinations are instanciated; Table 1.3 shows these instances, their names (that is, how they should be called from the `shell`), and the type of automata they allow to work with.

The first part of the table shows *Boolean* automata. The first instance, where the letters are characters, corresponds to *classical* automata and has been used in the First contact section. The next instance handles Boolean automata whose letters are integers; the three others support alphabets of pairs. All of these are called Boolean automata because each word is associated with a Boolean *weight*: either the word is accepted and its weight is *true*, or it is not and its weight is *false*.

The instances for weighted automata are listed in the second part of Table 1.3. The first four instances work with automata with weights in the ring of integers, and over free monoids with different types of generators, the next five work with automata over a free monoid of characters and with weights in different semirings. The third part shows the transducers, instanciated in Vaucanson 1.4; they are called *fmp-transducers*, where *fmp* stands for *free monoid products*.[4]

1.2.3 Command options

Every TAF-Kit instance determines the weight semiring and the type of letters in the alphabet(s). This is sufficient at compile time, but when a TAF-Kit command is *executed*, some

[4]This name, or precision, comes from the fact that a transducer can be considered as well as an automaton over the input monoid with weights in the rational series over the output monoid. In Vaucanson, such type of transducers is called *rw-transducers*, where *rw* stands for *rational weights*, to distinguish them from the *fmp-transducers*. No *rw-transducers* are instanciated in TAF-Kit 1.4.

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-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-z`	automata	characters	$\langle \mathbb{Z}, +, \times \rangle$
`vcsn-int-z`	automata	integers	$\langle \mathbb{Z}, +, \times \rangle$
`vcsn-char-char-z`	automata	pairs of characters	$\langle \mathbb{Z}, +, \times \rangle$
`vcsn-int-int-z`	automata	pairs of integers	$\langle \mathbb{Z}, +, \times \rangle$
`vcsn-char-zmax`	automata	characters	$\langle \mathbb{Z}, \mathsf{max}, + \rangle$
`vcsn-char-zmin`	automata	characters	$\langle \mathbb{Z}, \mathsf{min}, + \rangle$
`vcsn-char-r`	automata	characters	$\langle \mathbb{R}, +, \times \rangle$
`vcsn-char-q`	automata	characters	$\langle \mathbb{Q}, +, \times \rangle$
`vcsn-char-f2`	automata	characters	$\langle \mathbb{F}_2, +, \times \rangle$
`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	$\langle \mathbb{Z}, +, \times \rangle$

Table 1.3: The TAF-Kit instances in Vaucanson 1.4

more informations or data have to be known by, or given to, the command. They roughly fall into three different classes:

1. the *letters* in the alphabet(s);
2. informations on the *input* and *output formats*, which control the way the arguments will be read and the results output by the command;
3. data, called *writing data*, which control the way *rational expressions* are written or read as symbol sequences; this is partly related with the letters in the alphabets.

The letters of the alphabets have to be given explicitely to the command. In many cases however, this is transparent to or unnoticeable by the user: if a command calls an automaton (or an expression) as a parameter and if this parameter is an `XML` file — under the the Fsm XML format which is read by Vaucanson—, the letters are contained in the file, and nothing is to be added. In the other cases, the letters have to be listed in an option.

Data of the two other classes are given default values. They may be useful in order to get the desired result, they are sometimes necessary to read the parameters as files under a certain formats.

1.2.4 TAF-Kit's modus operandi

Each instance of TAF-Kit is a compiled program which offers a set of commands. All TAF-Kit instances work identically. They differ on the type of automata they handle, and may offer different commands because not every algorithms (and thus commands) work on any automata type (*cf.* Chapter 2).

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

$$\underbrace{\texttt{vcsn-char-b}}_{\text{TAF-Kit instance}}\ \underbrace{\underbrace{\texttt{determinize}}_{\text{name}}\ \underbrace{\texttt{a1.xml}}_{\text{arg.}}}_{\text{command 1}}\texttt{\textbackslash|}\,\underbrace{\underbrace{\texttt{minimize}}_{\text{name}}\ \underbrace{\texttt{-}}_{\text{arg.}}}_{\text{command 2}}\texttt{\textbackslash|}\,\underbrace{\underbrace{\texttt{data}}_{\text{name}}\ \underbrace{\texttt{-}}_{\text{arg.}}}_{\text{command 3}}$$

The *internal pipe*, '`\|`', is used to separate commands. A command starts with a name, it 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. So 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.

As explained in Section 1.2.3, the parameter(s) of a command may be completed and its behaviour may be controlled by some options. We describe these options with more details in the next section.

For each command, TAF-Kit will

1. parse the options,
2. parse all expected arguments (using indications that may have been given by 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 printing step of the command before the '`\|`' and the parsing step of the command after the '`\|`' are of course omitted.

1.2.5 Automata repository and factory

Most of TAF-Kit functions allow to build automata from others. There are functions which take a rational expression and yield an automaton that accepts the language denoted by the expression, and a function `edit` that allows to define (or to transform) an automaton element by element (*cf.* Section 1.5.5). Other features of TAF-Kit for the definition of automata are the *automata repository* and the *automata factory.*

1.2.5.1 Automata repository

With our first example (*cf.* Section 1.1), we mentioned that an automaton '`a1.xml`' is ready and available to the functions of the instance '`vcsn-char-b`'. There exist some other automata for the same purpose, and such automata also exist for other instances of TAF-Kit 1.4; their list is available via the option `--list-automata`:

```
$ vcsn-char-b --list-automata
The following automata are predefined:
```

```
- a1.xml
- b1.xml
- div3base2.xml
- double-3-1.xml
- ladybird-6.xml
```

For every TAF-Kit instance `vcsn-xxx-y`, the `XML` files for these automata are located at in a special directory, `vaucanson-1.4/data/automata/xxx-y` (*cf.* Section 1.5.0). More details on these automata are given at Appendix A.

1.2.5.2 Automata factory

In the same directory as the automata quoted above, some programs have been compiled which generate new automata, depending on parameters given to the program. The name of the program is suffixed by the characteristic part of the name of the TAF-Kit instance.[5] For instance, the program `divkbaseb-char-b` generates the automaton that accepts the representation in base '`b`' of numbers divisible by by '`k`'.

```
$ divkbaseb-char-b 5 3 > div5base3.xml
$ vcsn-char-b data div5base3.xml
States: 3
Transitions: 6
Initial states: 1
Final states: 1
$
```

We give another example of construction of an automaton with the factory at Section 1.3.4.

1.3 Option specifications

The list of possible *options* of a TAF-Kit command is obtained with the (classical) '`--help`' option. They fall in the following categories:

1. options that give information on the instance;
2. specifications of the alphabet(s);
3. determination of the input and output formats;
4. activation of benchmarking options;
5. and finally parametrization of the grammars for rational (that is, regular) expressions.

The description of latter, called *writing data*, is postponed to the next section.

Along the Unix tradition, the options are given long names, called with the prefix '`--`', together with short equivalent names, prefixed with a simple '`-`', which, in practice, will often be prefered.

[5]If Vaucanson is only compiled without being installed, one should first go to the '`vaucanson-1.4/data/automata/char-b`' directory by a `cd` command, and type '`./divkbaseb-char-b`' instead of '`divkbaseb-char-b`' in the command of the example.

1.3.1 Information options

They are listed in Table 1.4.

long option	short	purpose of the option
`--help`	`-?`	Give the help list
`--usage`		Give a short usage message
`--version`	`-V`	Print program version
`--list-commands`	`-l`	List usual commands
`--list-all-commands`	`-L`	List all commands, including debug commands
`--list-automata`		List predefined automata

Table 1.4: Information options

Caveat: The character '`?`' being interpreted by the shell, it should be protected in order to be given as an argument to a command. Without such a protection, the behaviour may depend on the shell, and according to the files within the directory.

This option '`?`' should probably be suppressed, but it is necessary for the library '`argp`' which is used for reading the options in the command line and it does not seem easy to get around it. In any case, it should be avoided, and the '`--help`' option be used.

1.3.2 Alphabet specification

The necessity of alphabet specification As we have seen (Section 1.2.2), every TAF-Kit instance determines (or one could say, is determined by) the type of the letters that generate the free monoid(s) over which the automata or the rational expressions are built. And this is sufficient at compile time, that is, in order to generate TAF-Kit.

But Vaucanson and the TAF-Kit functions are designed in such a way that they need to know the *complete* type of an automaton or an expression in order to handle it, that is, not only the type of weights and of letters, but also the *set of letters* that constitute the alphabet(s).

The `XML` files which describe automata, or expressions, contain this information and are so to say self-contained. For instance, when we read '`a1.xml`' in Section 1.1 and determinized this automaton, we did not have to tell TAF-Kit that the alphabet was $A = \{a, b\}$. On the contrary, when the automaton, or the expression, does not exist prior to the TAF-Kit function, then specifying an alphabet *is mandatory*. For instance, the following commands[6] end in error:

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

[6]The function `edit` is described at Section 1.5.5, `exp-to-aut` which takes a rational expression and converts it into an automaton at Section 2.1.5.2.

In the latter case moreover, and as there is no *a priori* restriction on the characters that can be used as letters, VAUCANSON needs to know the alphabet over which the expression is built in order to parse the rational expression: there is no other way for guessing whether the alphabet is $A = \{a, b\}$ (and the '+' is a rational operator) or if the alphabet is $B = \{a, b, +\}$ and the '+' is just a letter.

Specifying the alphabet can be done by using '`--alphabet=ab`' or its short equivalent '`-aab`'. For instance, the correct writing of the above command reads:

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

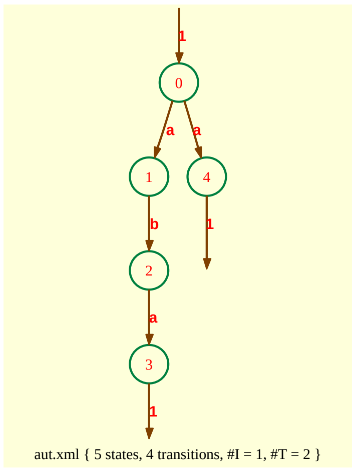


Figure 1.3: Result of the command `vcsn-char-b display aut.xml`

Table 1.5 reviews the alphabet specification options. The different possibilities: *characters*, *integers*, and *pairs* need to be described with more details.

long option	short	purpose of the option
`--alphabet`	`-a`	specify the alphabet of automata or rational expressions
`--alphabet1`	`-a`	specify the first (or input) alphabet of transducers (`fmp`)
`--alphabet2`	`-A`	specify the second (or output) alphabet of transducers (`fmp`)

Table 1.5: Alphabet options

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. Some character alphabets are predefined. These are:

‘`letters`’	for	the lower case letters $\{a, b, \ldots, z\}$.
‘`alpha`’	for	the upper and lower case letters $\{a, b, \ldots, z, A, B, \ldots, Z\}$.
‘`digits`’	for	all digits $\{0, 1, \ldots, 9\}$.

For instance, ‘`-aletters`’ is an abbreviation for ‘`-aabcdefghijklmnopqrstuvwxyz`’. The above list of *predefined alphabets* is obtained by typing ‘`vcsn-char-b --help`’.

When specifying characters alphabets, the following characters have to be escaped with a backslash:

␣ (space) ‘'’ ‘"’ ‘(’ ‘)’ ‘=’ ‘,’ ‘\’

and in this case the list of characters has to be put within quotes. The same characters are then used normally — without being escaped — in the expression. For instance, the following commands will create an automaton that recognize all ‘decimal’ numbers written in base 2, and then display the quotient[7].

```
$ vcsn-char-b -a'01\,' exp-to-aut '1(0+1)*+1(0+1)*,(0+1)(0+1)*' > dec-bin.xml
$ vcsn-char-b quotient dec-bin.xml \| display -
```

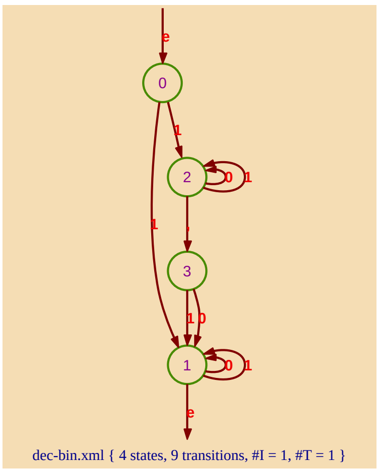


Figure 1.4: Result of the command `vcsn-char-b quotient dec-bin.xml \| display -`

Integer alphabets The letters of an integer alphabet must be specified as signed integer (they are represented by the C++ type `int`), and should be separated by commas. For instance, the following command 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 -a1,2,5,10,20,50 exp-to-aut '1*2*5*10*20*50*' > coins.xml
$ vcsn-int-b eval coins.xml '1210'
1
$ vcsn-int-b eval coins.xml '12105'
```

[7]The function `quotient` is described at Section 2.2.1.4; ‘`dec-bin.xml`’ is an automaton with 12 states and 27 transitions and diplaying it would have been messy.

```
0
$ vcsn-int-b eval coins.xml '121050'
1
$
```

Note that *digits* are *characters* and not *integers*, even if they look like the latter (for integers between 0 and 9) and if, in VAUCANSON 1.4, no operations on integer letters are implemented that could differentiate them. The only difference is thus the syntax when listing them in the option.

Pair alphabets Pair alphabets should be specified using parentheses and commas to form pairs — with types of letter that match the TAF-KIT instance, of course —, as in the following example:

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

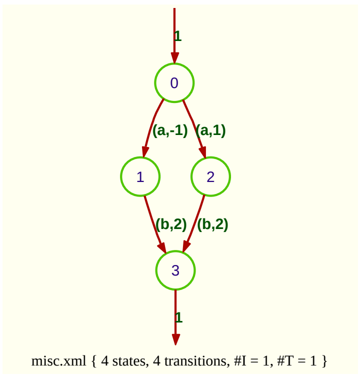


Figure 1.5: Result of the command `vcsn-char-int-b display misc.xml`

Alphabets for transducers The products of two free monoids 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 to '`-a`' and '`-A`' respectively. Table 1.3 gives the two possible choices for these alphabets in TAF-KIT 1.4: both *character*, or both *integer*, alphabets. The following command calls for the interactive construction of the right normaliser for numbers written in base 2 which is then shown below (*cf.* [4]).

```
$ vcsn-int-fmp-b -a0,1,2 -A0,1 edit norm2.xml
...
$
```

Caveat: La fonction `exp-to-aut` n'est pas implémentée dans TAF-KIT 1.4 pour les instances `fmp` (*cf.* Section 2.5).

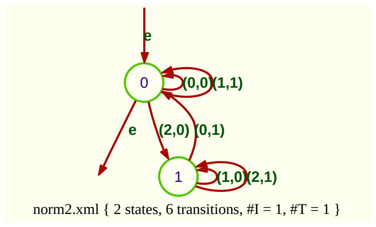


Figure 1.6: The normaliser in base 2

Unix usage The command line is first interpreted by the `shell`, which makes the characters '`.`', '`?`', '`*`', '`'`', *etc.* being given their meaning for the `shell`. In order to give them their meaning in the current alphabet and in the writing of rational expressions, they have to be protected by '`'`', or '`"`'.

```
$ vcsn-char-b -aab cat-E aab
aab
$ vcsn-char-b -aab cat-E aa(b)
zsh: unknown file attribute
$ vcsn-char-b -aab cat-E 'aa(b)'
aa.b
$ vcsn-char-b -aab cat-E aab*
zsh: no matches found: aab*
$ vcsn-char-b -aab cat-E "aab*"
aab*
```

The normal unix `shell` definition, allocation and utilisation of variables may be mixed with the usage of TAF-Kit command lines. 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:

```
$ 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
$ vcsn-char-b eval numbers.xml 1,234,987
1
$ vcsn-char-b eval numbers.xml 1,24,987
0
```

Note how the expression must be enclosed with '`"`' rather than with '`'`' in order to be correctly interpreted.

```
$ 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
Lexer error, unrecognized characters: $d+$d$d+$d$d$d)(,$d$d$d)*
```

1.3.3 Input and output formats

The TAF-Kit commands are supposed to input and output objects of different sorts: automata, rational expressions, words, weights and Boolean results. Their formats are controlled by the attributes of the input and output options. As shown on Table 1.6, there is one default format when no format option is called.

long option	short	purpose of the option
`--input`	`-i`	select input format for automata and rational expressions
`--output`	`-o`	select output format for automata and rational expressions
`--verbose`	`-v`	select verbose option for Boolean results

values for `-i` or `-o`	format for automata	format for rational expressions
(none)	Fsm XML	text string
`xml`	Fsm XML	Fsm XML
`fst`	OpenFst	—
`dot` (output only)	`dot`	—

Table 1.6: Input and output options and formats

These options are used not only to control and adequatly adjust the format of data handled by TAF-Kit in order to process them but allow also to make TAF-Kit a translator between different format for a given object.

1.3.3.1 Automata formats

Automata are always *files*; they are read from a file whose filename is specified on the command line, and the file is output on the standard output (or can be diverted to a named file in the Unix way).

Vaucanson can *read* automata in two formats: Fsm XML (the default format), or the textual format of OpenFst. It can write automata in these formats, as well as in the '`dot`' format[8] that can then be used for graphical output afterwards.

```
$ vcsn-char-b -ofst cat b1.xml
0   0   a   0
0   0   b   0
0   1   b   0
1   1   a   0
1   1   b   0
1    0
$ vcsn-char-z -ofst cat b1.xml \| -ifst eval - 'bab'
2
$
```

The first two comand lines below are equivalent to the third one.

```
$ vcsn-char-b -odot cat b1.xml > b1.dot
$ dotty b1.dot
$
$ vcsn-char-b display b1.xml
$
```

[8]`dot` files can be processed and visualized using the `GraphViz` package.

1.3.3.2 Rational expression formats

Rational expressions are given either as *character strings* — default format — or `XML` *files* — `xml` format.

By default, rational expressions are *read* as strings given on the *command line*, and *output* as strings on the *standard output*. Both can be diverted in the Unix way, but a string written in a file cannot be passed to TAF-Kit as a file.

```
$ vcsn-char-b -aab cat-E '(a+b(a(b)*a)*b)*'
(a+b.(a.b*.a)*.b)*
$ vcsn-char-b -aab cat-E '(a+b(a(b)*a)*b)*' > exp.txt
$ cat exp.txt
(a+b.(a.b*.a)*.b)*
$ vcsn-char-b -aab cat-E exp.txt
Lexer error, unrecognized characters: exp.txt
$ cat exp.txt | vcsn-char-b -aab cat-E -
(a+b.(a.b*.a)*.b)*
```

Alternatively, rational expressions can be read from an Fsm XML file whose filename is given on the command line, and output as an Fsm XML file as well.

```
$ vcsn-char-b -aab -oxml cat-E '(a+b(a(b)*a)*b)*' > exp.xml
$ vcsn-char-b -ixml cat-E exp.xml
(a+b.(a.b*.a)*.b)*
```

1.3.3.3 Word formats

Words are always *strings* of letters, that are read on the command line, and written on the standard output.

Caveat: Although *words* are, from a formal point of view, a (simple) instance of a rational expression, TAF-Kit 1.4 handles them as objects of different and uninterchangeable types. We come back to the subject in the next section.

1.3.3.4 Weight formats

Weights, that is, elements of the weight semiring, and such as the result of the evaluation of a word in an automaton for instance, are simply output as *strings* on the standard output.

```
$ vcsn-char-z eval c1.xml '101101'
45
$
```

The way they are input, as strings sa well, as part of a rational expression, is described in the next section.

1.3.3.5 Boolean result formats

Some TAF-Kit functions, such as `is-empty` which determines whether an automaton is empty or not, yield Boolean results. In the default format, such results are returned using the *status code* of the TAF-Kit instance, so that the correponding commands can be used as conditions in `shell` scripts. According to Unix convention, the status code 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 -v is-empty coins.xml
Input is not empty
```

1.3.4 Benchmarking options

The functions in Vaucanson library are interspersed with instructions which trigger time measurement in case some dedicated variables are set up in a certain way. This feature is primarily intended to the adjustment and improvement of the programming of the library rather than to the benefit of TAF-Kit users. It can nevertheless be activated through TAF-Kit by instantiating some options. As they appear when the `--help` option is called, we list them in Table 1.7 and briefly present them afterwards. We do not fully document these options as they are anyway not yet finalized.

long option	short	purpose of the option
`--report-time[=VERBOSE_DEGREE]`	`-T`	Report time statistics
`--export-time-dot[=VERBOSE_DEGREE]`	`-D`	Export time statistics in DOT format
`--export-time-xml[=VERBOSE_DEGREE]`	`-X`	Export time statistics in XML format
`--bench=NB_ITERATIONS`	`-B`	Bench
`--bench-plot-output=OUTPUT_FILENAME`	`-O`	Bench output filename

Table 1.7: Benchmarking options and formats

1.3.4.1 Time statistics

The `--report-time` option, `-T` for short, builds a file with some time statistics for the execution of the function it is called with, and outputs it on the `standard error output`. It is recommanded to divert it (with the `2>` redirection) to a file which will be exploited afterwards. The example below shows only some lines (the most important ones) of this file.[9]

```
$ vcsn-char-b -T1 determinize ladybird-10.xml > ldb10det.xml 2> ldb10-time.txt
$ cat ldb10-time.txt
Taf-kit command bench
```

[9]The automaton `ladybird-10.xml` has been built beforehand by the factory `ladybird-char-b`.

```
...
Charge  id:      <name>      total      self        calls   self avg.   total avg.
100.0%   0:      _program    343.78ms   343.78ms        1       0.34s       0.34s
 64.0%   9:  automaton output  220.02ms 220.02ms        1       0.22s       0.22s
 29.5%   7:        determinize  101.29ms 101.25ms       1    101.25ms    101.29ms
  3.9%   1:CMD[0]: determiniz  123.52ms  13.52ms        1     13.52ms    123.52ms
  2.4%   2:   automaton input    8.19ms   8.19ms        1      8.19ms      8.19ms
  0.1%   4:        eps_removal    0.29ms   0.29ms        1      0.29ms      0.29ms
  0.1%   3:             cut_up    0.21ms   0.21ms        1      0.21ms      0.21ms
  0.0%   8:is_realtime (autom    0.04ms   0.04ms        1      0.04ms      0.04ms
  0.0%   5: accessible_states    0.03ms   0.03ms        1      0.03ms      0.03ms
  0.0%   6:     sub_automaton    0.01ms   0.01ms        1      0.01ms      0.01ms
...
$
```

The content of the time statistics output is controlled by an integer called `VERBOSE_DEGREE` and which can take the values `1`, `2`, or `3`. Default value is `2`.

The `-D` and `-X` options have the same behaviour as `-T` but output the file under another format:

1.3.4.2 Benching

The `--bench` option, `-B` for short, makes TAF-Kit to repeat the functions that follow the option the number of times that is specified (compulsory parameter) with the option. The data shown in the example above are stored in a result file for each of the execution, and then a summary of these data is made, which contains the mean, the sum, the minimum and the maximum. This result file is output on the `standard error output`, which can be diverted as usual.

```
$ vcsn-char-b -B5 determinize ladybird-10.xml > ldb10det.xml 2> ldb10-bench.txt
$ cat ldb10-bench.txt
------------------------- SUMMARY -------------------------
------------------------- Arithmetic mean
[Task list:]

Charge  id:        <name>       total     self      calls   self avg. total avg.
100.0%  0:           _program  351.14ms 351.14ms        1       0.35s      0.35s
 64.0%  9:  automaton output  224.79ms 224.79ms        1       0.22s      0.22s
 30.1%  7:        determinize  105.71ms 105.69ms        1    105.69ms   105.71ms
  3.5%  1:CMD[0]: determiniz  126.12ms  12.29ms        1     12.29ms   126.12ms
  2.2%  2:   automaton input    7.65ms   7.65ms        1      7.65ms     7.65ms
  0.1%  4:        eps_removal    0.24ms   0.24ms        1      0.24ms     0.24ms
  0.1%  3:             cut_up    0.19ms   0.19ms        1      0.19ms     0.19ms
  0.0%  8:is_realtime (autom    0.03ms   0.03ms        1      0.03ms     0.03ms
  0.0%  5: accessible_states    0.02ms   0.02ms        1      0.02ms     0.02ms
  0.0%  6:     sub_automaton    0.01ms   0.01ms        1      0.01ms     0.01ms
...
$
```

1.4 The writing of rational expressions

The *definition* of rational (or regular) expressions is rather an easy and classical subject of any first year course in computer science (at least for the Boolean case). Reading and writing the same expressions prove to be a much more tricky matter, for several reasons. Some are specific to VAUCANSON: to begin with, no characters are reserved for the rational operators and the usual ones may appear as letters in the alphabet over which the expressions are built; the writing of weights, and the possibility of having integers as *letters* add to the problem. The effective implementation of reading and writing *strings* that represent *expressions*, together with the usual, and necessary, convention and simplification also conceal difficulties that have to be circumvented by any software that deals with expressions.

1.4.1 The definition of expressions

1.4.1.1 Construction of expressions

The general definition reads as follow. A rational expression *over a monoid* M *with weight in a semiring* $\mathbb{K}$ is a well-formed formula built from:

- the elements of M, which are the *atomic formulas*;
- the following operators:
  1. two 0-ary operators, or *constants*, denoted by '`0`' and '`1`' ;
  2. one unary operator *star*, denoted by '$*$' ;
  3. two binary operators, *sum* and *product*, denoted by '$+$' and '$\cdot$' ;
  4. and, for every k in $\mathbb{K}$, two unary operators, the *left* and *right exterior multiplications* by k, denoted by '`k.`' and '`.k`' .

This definition is the one taken by members of the VAUCANSON group in their writings about weighted rational expressions (*cf.* [7, 5]). It must be said that it is not the most common one. In general — if one may say so of the few publications that deal with weighted rational expressions —, the elements of $\mathbb{K}$ are atomic formulas and the left and right exterior multiplications are expressed with the product operator.

The VAUCANSON choice is more natural for the definition of the *derivation of expressions*, even if it has the theoretical drawback of introducing an infinity of operators — something that logicians do not like very much usually.

Being a formula, an expression may be viewed as a (finite) *tree* whose (inner) nodes are labelled with operators and leaves by atoms. The tree itself may be faithfully represented in different ways. The FSM XML format (*cf.* Appendix B) provides all necessary tags to describe such a tree.

1.4.1.2 Reduction of expressions

Like automata, the rational expressions are a *symbolic* (and *finite*) representation of languages or series. Natural valuation of the atoms and induction rules make every expression *denotes* a language or a series. Two rational expressions are *equivalent* if they denote the same languages, or series. We want *a priori* to distinguish between two distinct equivalent expressions — in particular since it is not always possible to decide whether two expressions are equivalent or not.

For several reasons, we distinguish indeed between expressions that are obviouly equivalent, such as $(\mathsf{E}+\mathsf{F})$ and $(\mathsf{F}+\mathsf{E})$, or $((\mathsf{E}+\mathsf{F})+\mathsf{G})$ and $(\mathsf{E}+(\mathsf{F}+\mathsf{G}))$. There are however expressions which can be constructed by the above rules, such as $(\mathsf{E}+\mathsf{0})$ or $(\mathsf{1}\cdot\mathsf{E})$, and which we do not want to *exist.* Such convention are not only useful for simplifying expressions, they are also *necessary* to make some computation processes (such as *derivation*) finite.

Everytime a rational expression is constructed inside VAUCANSON, either as the result of a computation or as the mere consequence of the *reading* of a string of symbols that represents it, the following rewriting rules, called *trivial identities*, and listed in Table 1.8, are automatically applied, giving rise to a so-called *reduced expression* which is obviously equivalent to the original expression.

In this table, E stands for any rational expression, m is any monoid element (that is, a *word*, or a *pair of words*), k and h are weights, while $\{0_{\mathbb{K}}\}$ and $\{1_{\mathbb{K}}\}$ designate the zero and unit of the weight semiring. Any subexpression of a form listed to the left of a '$\Rightarrow$' is rewritten as indicated on the right.

$$\mathsf{E.0} \Rightarrow \mathsf{0} \quad \mathsf{0.E} \Rightarrow \mathsf{0} \qquad \mathsf{E}+\mathsf{0} \Rightarrow \mathsf{E} \quad \mathsf{0}+\mathsf{E} \Rightarrow \mathsf{E} \qquad \mathsf{E.1} \Rightarrow \mathsf{E} \quad \mathsf{1.E} \Rightarrow \mathsf{E} \qquad \mathsf{0}^{\star} \Rightarrow \mathsf{1} \qquad (\mathbf{T})$$

$$\{0_{\mathbb{K}}\}\mathsf{E} \Rightarrow \mathsf{0} \quad \mathsf{E}\{0_{\mathbb{K}}\} \Rightarrow \mathsf{0} \qquad \{k\}\mathsf{0} \Rightarrow \mathsf{0} \quad \mathsf{0}\{k\} \Rightarrow \mathsf{0} \qquad \{1_{\mathbb{K}}\}\mathsf{E} \Rightarrow \mathsf{E} \quad \mathsf{E}\{1_{\mathbb{K}}\} \Rightarrow \mathsf{E} \qquad (\mathbf{T}_{\mathbb{K}})$$

$$\{k\}(\{h\}\mathsf{E}) \Rightarrow \{kh\}\mathsf{E} \qquad (\mathsf{E}\{k\})\{h\} \Rightarrow \mathsf{E}\{kh\} \qquad (\{k\}\mathsf{E})\{h\} \Rightarrow \{k\}(\mathsf{E}\{h\}) \qquad (\mathbf{A}_{\mathbb{K}})$$

$$\mathsf{1}\{k\} \Rightarrow \{k\}\mathsf{1} \qquad \mathsf{E}.(\{k\}\mathsf{1}) \Rightarrow \mathsf{E}\{k\} \qquad (\{k\}\mathsf{1}).\mathsf{E} \Rightarrow \{k\}\mathsf{E} \qquad (\mathbf{U}_{\mathbb{K}})$$

$$m\{k\} \Rightarrow \{k\}m \qquad (\mathbf{C}_{\mathsf{at}})$$

Table 1.8: The trivial identities

```
$ vcsn-char-z -aab cat-E '{2}ab{3}'
{2} (ab {3})
$ vcsn-char-z -aab cat-E '{2}a{3}'
{6} a
```

These rewriting rules mean that it is *impossible* for VAUCANSON to output a rational expression such as '`({3}(0(ab)))*{4}`'. This expression is *by construction* equal to '`{4}1`' as it can be verified with the following command:

```
$ vcsn-char-z  -aab cat-E '({3}(0(ab)))*{4}'
{4} 1
```

This command `cat-E` does not apply any algorithm to 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 performed while reading the expression.

1.4.2 Parsing strings into expressions

As we wrote above, there are several classical ways of faithfully representing an expression by a string of symbols. We are nevertheless faced with two, and even three, problems.

First, we want to avoid the blotted form of marking languages, and even of fully parenthesised forms, and to be able to use the more natural and common way of writing expressions with implicit precedence of operators. Another difficulty arises when the operators, letters, and weights share the same alphabet of characters for their represention. Finally, the possibility of having *integers* as generators of a free monoid, that is, ‘letters’ that are written as sequences of characters, brings in another problem. We treat these questions one after the other, and begin with what can be considered as the *default conventions*.

We first suppose that the alphabet is an alphabet of *characters* (letters and/or digits for the time being) and has been defined by means of the `--alphabet` option. According to the above definition, we define in Vaucanson rational expressions *over* A^* (as opposed to rational expressions over A), that is, any *word* of A^* — string of letters of A — is seen as an *atomic* expression. This feature may prove to be somewhat misleading (see below).

1.4.2.1 The rational operators

The three *rational operators*, sum, product and (Kleene) star are represented — by default — as in the following Table 1.9. The representation of the (left and right) exterior multiplications, that is, of weights, is described at Section 1.4.2.2.

Input	Output	Operator
E*	E*	Kleene star
EF or E.F	E.F	concatenation (implicit or explicit)
E+F	E+F	disjunction
(E)	as necessary	grouping

Table 1.9: Rational operators

Operators precedence The classical precedence relation between operators which allows to spare grouping symbol is extended in order to include the exterior multiplcations:

$$`\, * > \mathsf{k.} > \mathsf{.k} > \cdot > + \,' .$$

For instance, the rational expression which denotes the language that consists of all words that contain '`ab`' as a factor can be written (by a user) as '`(a+b)*ab(a+b)*`'. VAUCANSON outputs it by making the product between non-atomic subexpressions explicit.

```
$ vcsn-char-b -aab cat-E '(a+b)*ab(a+b)*'
(a+b)*.ab.(a+b)*
$ vcsn-char-b -aab cat-E '((a+b)*)(((ab))(a+b)*)'
(a+b)*.ab.(a+b)*
```

An atom which is enclosed in grouping symbols is not an atom anymore.

```
$ vcsn-char-b -aab cat-E '((a)(b))'
a.b
```

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

Associativity Sum and product of languages or series are associative, but it is not the case of the corresponding rational operators, as we have recalled above. The default bracketing is *on the left*, that is, $a + b + c$ is the same as $(a + b) + c$, and $a + (b + c)$ is another expression, as shown by the construction of the Thompson automata: the outcome of the following commands is shown at Figure 1.7.

```
$ vcsn-char-b -aabc thompson 'a+b+c' > thomp-abc-left.xml
$ vcsn-char-b display thomp-abc-left.xml
$ vcsn-char-b -aabc thompson 'a+(b+c)' > thomp-abc-right.xml
$ vcsn-char-b display thomp-abc-right.xml
```

```
$ vcsn-char-b -aabc cat-E  'a+b+c'
a+b+c
$ vcsn-char-b -aabc cat-E  'a+(b+c)'
a+b+c
```

```
$ vcsn-char-b -aabc cat-E  ’a.b.c’
a.b.c
$ vcsn-char-b -aabc cat-E  ’a.(b.c)’
a.b.c
```

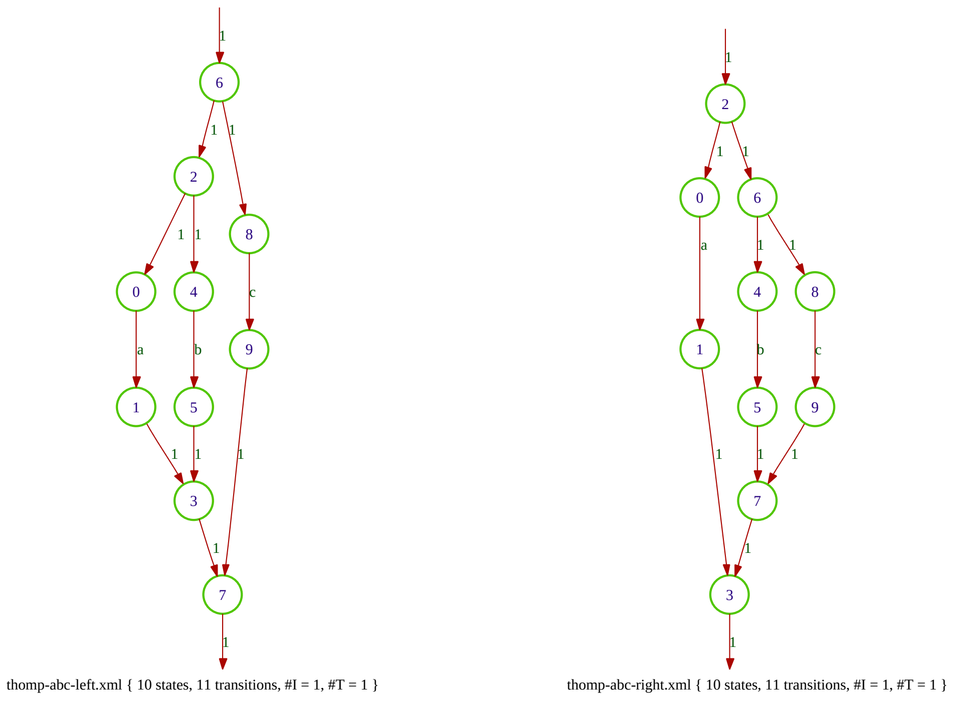

thomp-abc-left.xml { 10 states, 11 transitions, #I = 1, #T = 1 }
thomp-abc-right.xml { 10 states, 11 transitions, #I = 1, #T = 1 }

Figure 1.7: The operator ‘+’ is not associative

1.4.2.2 The weights

Weights are written in *braces*, as in ‘{3}’. When the expression is output by VAUCANSON, weights are also followed[10] by a blank space.

```
$ vcsn-char-z -aab cat-E ’{2}a + {2}   b’
{2} a+{2} b
```

As another example, the automaton $\mathcal{C}_1$ of Figure 1.8 is described in the file `c1.xml` and gives rise to the following command and output:

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

Eventhough all semirings which are instantiated in TAF-KIT 1.4 are *commutative*, this is not an assumption which is made in VAUCANSON in general. In any case, the weight semiring be commutative or not, the left and right exterior multiplications yield distinct expressions, from which distinct automata are built.

[10]This is not so good and will hopefully be corrected in further versions of VAUCANSON.

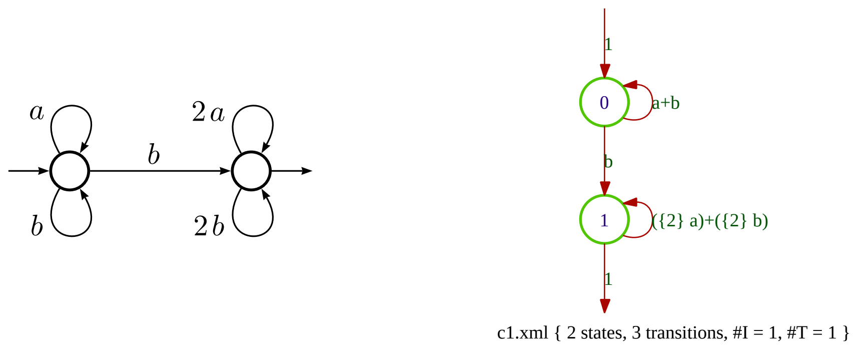


Figure 1.8: The $\mathbb{Z}$-automaton $\mathcal{C}_1$ and its display by `Graphviz`.

```
$ vcsn-char-z -aab cat-E '{2}ab{3}'
{2} (ab {3})
$ vcsn-char-z -aab cat-E '{2}{3}ab'
{6} ab
```

1.4.3 Parser parametrization

As there is *a priori* no restriction on the alphabet, the representation of the rational operators — called *token* — may collide with the one of elements of the monoid. VAUCANSON actually allows every operator to be represented by an *arbitrary string*. The set of these representations is called the *writing data*.

It is a feature of VAUCANSON that some different default values are prepared for the constants so that TAF-KIT may try to choose a representation which does not collide with the words. For the same purpose, the other tokens have to be given explicitely.

1.4.3.1 Implicit parametrization: the constants

The constants 0 and 1 are naturally written by default as `0` and `1`. This is witnessed, for instance, in the following command call that instantiates the last of the trivial identities (**T**) (*cf.* Table 1.8):

```
$ vcsn-char-b -aab cat-E '0*'
1
```

If '`1`' is a *letter* in the alphabet — as a character (digit) — the same symbol cannot be used for representing the constant 1 nor the identity of the monoid, that is, the empty word.[11] VAUCANSON chooses the first available representation of the identity from the following list of candidate symbols: '`1`', '`e`', or '`_e`', which does not collide with any letter of the alphabet. If the alphabet contains the three characters '`1`', '`e`', and '`_`', the default representation of the constant 1 is still '`_e`' and another less ambiguous representation has to be chosen explicitely (*cf.* below).

[11] In TAF-KIT 1.4, the functions which parse or compute with rational expressions over a product of free monoids are not implemented (*cf.* Section 2.5).

```
$ vcsn-char-b -aab1 cat-E '0*'
e
$ vcsn-char-b -aabe1 cat-E '0*'
_e
$ vcsn-char-b -a_abe1 cat-E '0*'
_e
```

Similarly, if '0' is a *letter* in the alphabet — as a character (digit) — the same symbol cannot be used for representing the constant 0 nor the null series and VAUCANSON chooses the first available representation of the zero from the following list of candidate symbols: '0', 'z', or '_z', which does not collide with any letter of the alphabet. Because of the trivial identities (see Section 1.4.1.2), this is a much rarer situation. The following calls to the `expand` function (*cf.* Section 2.1.5.3) yields 0 in a non trivial way:

```
$ vcsn-char-z -aa1 expand 'a+{-1}a'
0
$ vcsn-char-z -aa01 expand 'a+{-1}a'
z
$ vcsn-char-z -aaz01 expand 'a+{-1}a'
_z
$ vcsn-char-z -a_az01 expand 'a+{-1}a'
_z
```

For integer alphabets, the constant 1 and the empty word on one hand, the constant 0 and the null series on the other, are always (that is, even if the integers '1' or '0' are not in the alphabet) written as 'e' and 'z' respectively.

```
$ vcsn-int-z -a'2,3' expand '2+{-1}2'
z
$ vcsn-int-z -a'2,3' expand '(2+{-1}2)*'
e
```

1.4.3.2 Explicit parametrization: the `parser` option

Table 1.10 shows the tokens that are used in the writing of rational expressions within VAUCANSON, together with their meaning and default values. The `--parser` option can be used to modify the values of these tokens. Each of them must be defined as a *non-empty* string.

This ability of the user to defne the tokens at will allows to use characters of any kind as letters of the alphabet. For instance, one may define the language of well-parenthetized words of nested depth at most 2, over the alphabet $\{(,)\}$, for which one should obviously rename the '`OPAR`' and '`CPAR`' tokens.

```
$ vcsn-char-b -a'\(\)' --parser='OPAR=[ CPAR=]' cat-E '[([()]*)]*'
[(.()*.)]*
```

The values of the *writing data* are stored[12] in the `XML` file which contains the automaton or the expression, so there is no need to specify them again when working from a file.

[12]This is a questionable feature of both VAUCANSON 1.4 and the corresponding version of FSM XML, but it is so.

token	meaning	default value(s)
‘ZERO’	constant ‘0’ and the null series	‘0’, ‘z’, ‘_z’
‘ONE’	constant ‘1’ and the identity of the monoid	‘1’, ‘e’, ‘_e’
‘STAR’	Kleene star	‘*’
‘PLUS’	sum	‘+’
‘TIMES’	product	‘.’
‘CONCAT’	concatenation (product within the monoid)	‘’, ‘#’
‘OPAR’	group start	‘(’
‘CPAR’	group end	‘)’
‘OWEIGHT’	weight start	‘{’
‘CWEIGHT’	weight end	‘}’
‘SPACE’	space character (to be ignored)	‘ ’

Table 1.10: Tokens of the `parser` option: the writing data

```
$ vcsn-char-b -a'\(\)' --parser='OPAR=[ CPAR=]' exp-to-aut '[([()]*)]*' > par.xml
$ vcsn-char-b aut-to-exp par.xml
(.[(.).[(.)]*.)+)].[(.[(.).[(.)]*.)+)]]*+1
```

Caveat: It is the responsability of the user to define the tokens in such a way there is no collision between them nor with the elements of the monoid.

In case there exists such collisions, the way the tokens are recognized in a string of letters may depend upon the token.[13]

```
$ vcsn-char-b -a_abe1 cat-E '_e0*+e_e_.e'
_e+e_.e
$ vcsn-char-z -a_aez01 cat-E 'z_a0+a_z0+a(_z)0+a_z_e0'
z_a0+a_z0+a._z.0+a_z0
```

In the first line, the string `_e` has been recognized as the constant `1`; in the second, the string `_z` has not been recognized as the constant `0`.

As a consequence, it is not possible in Vaucanson 1.4 to use the alphabet of all ASCII characters.

The token `TIMES` As noted at Table 1.9, the token `TIMES` is given a unique value for the output of strings by Vaucanson, but the *empty string* is always accepted as input for the ‘representation’ of the same operator product.

```
$ vcsn-char-b -aab cat-E '(a+b)(b+a)'
(a+b).(b+a)
$ vcsn-char-b -a' -.'  --parser='TIMES=x PLUS=|' cat-E '... --- (...  |-... )'
... --- x(...  |-... )
```

[13]This has to be corrected in the forthcoming versions of Vaucanson.

The token `CONCAT` The token `CONCAT` is used to represent the same operator product, but between letters of the alphabet, when such a sequence forms an element of the monoid. As for `TIMES`, `CONCAT` is given a unique value for the output of strings by Vaucanson, but the *empty string* is always accepted as input for the 'representation' of the same operator. Indeed, the existence of this token is hardly noticeable when one uses alphabet of *characters* as its default value in this case is the empty string as well. It is necessary to explicitely give it a non empty value in order to make it appear.

```
$ vcsn-char-b -aab cat-E '(aba)(bab)'
aba.bab
$ vcsn-char-b -aab --parser='CONCAT=-' cat-E '(aba)(bab)'
a-b-a.b-a-b
```

This token is useful, and necessary, when the generators of the monoid, that is, the *letters*, are not characters but written as sequences of symbols. In TAF-Kit 1.4, this happens for the instances in which the type of letters are *integers*. In this case, the default value of `CONCAT` is '`#`'. The token is necessary when the set of letters, viewed as a set of words on the alphabet of digits, is not a prefix code.

```
$ vcsn-int-b -a'0,1,2' cat-E '10(12+21)*'
1#0.(1#2+2#1)*
$ vcsn-int-b -a'0,1,12,22' cat-E '10(12+122)*'
vcsn-int-b: Lexer error, unrecognized characters: 2)*
$ vcsn-int-b -a'0,1,12,22' cat-E '10(12+1#22)*'
1#0.(12+1#22)*
```

One understands that the parser matches the *longest prefix* of the string it reads with the letters of the alphabet.

The token `SPACE` The token `SPACE` is meant to be a character or a string that is equivalent to the empty sequence and that makes the writing of expressions as strings more readable by the users. Of course, its default value is the space character and is likely to keep this value unless the space character itself is a letter of the alphabet (as in the Morse alphabet considered in the example above).

Unfortunately, TAF-Kit 1.4 does not exactly implement this specification. When `SPACE` is used between letters of the alphabet, it is replaced by `TIMES`, instead of `CONCAT` as it should be.

```
$ vcsn-char-b -aab  cat-E '(aba)(bab)'
aba.bab
$ vcsn-char-b -aab  cat-E '(a b a)   (b a b)'
a.b.a.b.a.b
$ vcsn-char-b -aab --parser='CONCAT=-' cat-E '(aba)(bab)'
a-b-a.b-a-b
$ vcsn-char-b -aab --parser='CONCAT=-' cat-E '(a b a) (b a b)'
a.b.a.b.a.b
$ vcsn-char-b -aab --parser='CONCAT=- SPACE=#' cat-E '(a#b#a)(b#a#b)'
a.b.a.b.a.b
```

1.4.3.3 Overwriting the writing data

The writing data are used when *parsing* a string into a rational expression and when writing back a rational expression as a string, or even when displaying an automaton. A rational expression or an automaton themselves do not call on the writing data. Nevertheless, and as we said above, the writing data are embarked in the XML file that contains an automaton or an expression (*cf.* Appendix B). It makes these objects fully self-contained and allows for instance to convert them as a rational expression written as a string without giving additional information.

The '--parser=' option can then be used to modify the way the object will be *output*.

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

If we edit the file `p.xml` and suppress the writing data in it (and write the result in the file `pp.xml`), we then get the output with the default values for the tokens.

```
$ vcsn-char-b -ixml cat-E pp.xml
((.((.))*.))*
```

1.5 TAF-Kit IO functions

We end this chapter with the description of the input and output commands available within TAF-Kit. The other commands that perform computations on the automata and expressions are described in the next chapter.

1. `data` <*aut*>
2. `cat` <*aut*>
3. `cat-E` <*exp*>
4. `display` <*aut*>
5. `edit` <*aut*>

1.5.0 Data file location

TAF-Kit works (or a user works with TAF-Kit) in a current directory called *working directory*. On the other hand, every instance `vcsn-xxx-y` of TAF-Kit knows a directory, called *data directory*, located at `vaucanson-1.4/data/automata/xxx-y`, and where automata predefined by Vaucanson are stored. The latter form the *automata repository* of the instance (*cf.* Section 1.2.5).

Every TAF-Kit command *writes* in the working directory (or in any directory which is assigned by the usual `Unix` file path scheme). As we mentioned in Section 1.1, every TAF-Kit command *first reads* in the working directory, and, if the automaton is not found there, it *then reads* from the data directory.

1.5.1 data

```
$ vcsn data a.xml
$
```

Prints some characteristic data on the automaton *a.xml*.

***Specification*:**

The printed data are

```
#states, #transitions, #initial states, #final states.
```

1.5.2 cat

```
$ vcsn cat a.xml > b.xml
$
```

Reads the automaton *a.xml* and writes it in the file *b.xml*.

***Comments*:** The `cat` function of VAUCANSON works very much in the same way as the Unix `cat` command and allows in the same way to write a file on the standard output or in another file.

The main difference is the behaviour described above: the `cat` command first reads from the *working directory* and then from the *data directory* and thus allows to 'load' predefined automata from the data directory to the working one.

The next difference is that the format of both the input and output may be controlled via the `-i` and `-o` options, as described at Section 1.3.3.1. The `cat` function thus allows to convert a representation in one format into a representation in another one.

1.5.3 cat-E

```
$ vcsn-char-b -aab cat-E 'exp'
<red-exp>
$ vcsn-char-b -oxml cat-E 'exp' > e.xml
$ vcsn-char-b -ixml cat-E e.xml
<red-exp>
```

Read the expression ***exp*** given as a string, stores it in the memory, and writes it back, as a string by default.
It can also read and write the expression as an XML file.

***Comments*:** The different behaviours of the `cat-E` function according to the possible formats have been described at Section 1.3.3.2.

A rational espression output by `cat-E` is in reduced form (*cf.* Section 1.4.1.2).

1.5.4 display

```
$ vcsn display a.xml
$
```

Display the automaton *a.xml* via `Graphviz`.

***Comments*:** The ame functionality may be achieved by outputting the automaton *a.xml* in the `dot` format and then calling `dotty` directly (*cf.* Section 1.3.3.1).

The possibility of using VGI, a graphic interface written within the VAUCANSON project, will be given as soon as possible.

1.5.5 edit

```
$ vcsn edit a.xml
$
```

Create and edit the automaton *a.xml* via keyboard interface.

This command `edit` provides a textual interface to define automata interactively. It takes as argument the filename of the automaton to be defined or modified. If the file does not yet exist, the alphabet of the automaton should be specified on the command line (using the `--alphabet` or `--a` option as with any other command), and the file will be created when the editor is exited; 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 as shown on the following example.

```
$ vcsn-char-z --alphabet=ab edit 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.
  11. Exit without saving.

Your choice [1-11]:
```

Enter *1*, you will be prompted for the number of states to add; enter *1* again. The state `0` is created. To make it initial, select *5*, and:

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

To make it final, use choice *7* likewise.

```
Your choice [1-10]: 7
  For state: 0
 With weight:-3
```

Add finally a transition:

```
Your choice [1-10]: 3
  Add a transition from state: 0
  To state: 0
  Labeled by the expression: a+2b
```

As shown above, rational expressions are valid labels, that is, the automaton created is a *generalized automaton.* 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 (W: 1)
  Final states: 0 (W: -3)

  Transitions:
    1: From 0 to 0 labeled by a+(2 b)
```

Note that states are numbered from 0, but transitions numbers start at 1.

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

Chapter 2

Specification of functions on automata and rational expressions

Functions are classified according to the type of automata they are applied to. They depend upon the type of the monoid: free monoid and direct product of two free monoids at this stage of TAF-Kit (Vaucanson 1.4) and upon the type of the multiplicity semiring: 'numerical' semiring in general, and $\mathbb{R}$ (implemented as `float`) as an example of a *field* and Boolean semiring in particular. Some functions are specialised to even more particular type of alphabets. Note that TAF-Kit (Vaucanson 1.4) offers no instance where the multiplicity semiring is a semiring of series (over a free monoid with multiplicity in a numerical semiring)

In this chapter, we give the specifications of the functions, that is, the preconditions on their arguments, and the description of the result and how it is related to the argument.

1. General automata and rational expressions
2. Weighted automata and rational expressions over free monoids
3. Automata and rational expressions over free monoids with weights in a field.
4. Boolean automata and rational expressions over free monoids
5. Weighted automata over product of two free monoids
6. Weighted automata over free monoids over alphabets of pairs

This classification is used to organise the lists of commands. Every instance of TAF-Kit contains the commands of the first section and of one or several others, as indicated in Table 2.1 below. A command with input and output arguments with different types belongs to the instance corresponding to the *input* type. Moreover, such a command exists only if the type of the *output* argument is instanciated as well (*cf.* `partial-identity`, Section 2.2.1.5).

Comment for the Vaucanson *Group* (101205): *Chacune des sections suivantes commence par la liste des commandes dont on a pu penser, à un moment ou un autre et en particulier au cours de la seconde Jam session, qu'elles devraient être implémentées dans* Vaucanson *1.4.*

En commentaire, et dans le corps de la section, l'état actuel et les projets.

command name	alphabet type	weight semiring	function sections
`vcsn-char-b`	characters	$\langle\mathbb{B},\vee,\wedge\rangle$	1, 2, 4
`vcsn-int-b`	integers	$\langle\mathbb{B},\vee,\wedge\rangle$	1, 2, 4
`vcsn-char-z`	characters	$\langle\mathbb{Z},+,\times\rangle$	1, 2
`vcsn-int-z`	integers	$\langle\mathbb{Z},+,\times\rangle$	1, 2
`vcsn-char-zmax`	characters	$\langle\mathbb{Z},\mathsf{max},+\rangle$	1, 2
`vcsn-char-zmin`	characters	$\langle\mathbb{Z},\mathsf{min},+\rangle$	1, 2
`vcsn-char-r`	characters	$\langle\mathbb{R},+,\times\rangle$	1, 2, 3
`vcsn-char-q`	characters	$\langle\mathbb{Q},+,\times\rangle$	1, 2, 3
`vcsn-char-f2`	characters	$\langle\mathbb{F}_2,+,\times\rangle$	1, 2, 3
`vcsn-char-char-b`	pairs of characters	$\langle\mathbb{B},\vee,\wedge\rangle$	1, 2, 4, 6
`vcsn-char-int-b`	pairs of character and integer	$\langle\mathbb{B},\vee,\wedge\rangle$	1, 2, 4, 6
`vcsn-int-int-b`	pairs of integers	$\langle\mathbb{B},\vee,\wedge\rangle$	1, 2, 4, 6
`vcsn-char-char-z`	pairs of characters	$\langle\mathbb{Z},+,\times\rangle$	1, 2, 6
`vcsn-int-int-z`	pairs of integers	$\langle\mathbb{Z},+,\times\rangle$	1, 2, 6
`vcsn-char-fmp-b`	characters	$\langle\mathbb{B},\vee,\wedge\rangle$	1, 5
`vcsn-char-fmp-z`	characters	$\langle\mathbb{Z},+,\times\rangle$	1, 5
`vcsn-int-fmp-b`	integers	$\langle\mathbb{B},\vee,\wedge\rangle$	1, 5
`vcsn-int-fmp-z`	integers	$\langle\mathbb{Z},+,\times\rangle$	1, 5

Table 2.1: The TAF-Kit instances in Vaucanson 1.4 and their commands

2.1 General automata and rational expressions

Automata are 'labelled graphs', and these labels are, in full generality, elements of a monoid associated with a multiplicity (taken in a semiring). The commands considered in this section make assumption neither on the monoid, nor on the weight semiring. They are thus called by any instance of TAF-Kit, for automata of *any type*.[1]

1. Graph functions

   (1.1) `accessible` <*aut*>, `coaccessible` <*aut*>
   (1.2) `trim` <*aut*>, `is-trim` <*aut*>
   (1.3) `is-empty` <*aut*>
   (1.4) `is-useless` <*aut*>

2. Transformations of automata

   (2.1) `proper` <*aut*>, `is-proper` <*aut*>
   (2.2) `standardize` <*aut*>, `is-standard` <*aut*>

3. Operations on automata

   (3.1) `union` <*aut1*> <*aut2*>
   (3.2) `sum` <*aut1*> <*aut2*>
   (3.3) `concatenate` <*aut1*> <*aut2*>
   (3.4) `star` <*aut*>
   (3.5) `left-mult` <*aut*> <*k*>
   (3.6) `right-mult` <*aut*> <*k*>
   (3.7) `chain` <*aut*> <*n*>

4. Operations on behaviours of automata

   (4.1) `sum-S` <*aut1*> <*aut2*>
   (4.2) `cauchy-S` <*aut1*> <*aut2*>
   (4.3) `star-S` <*aut*>

5. Automata and expressions; operations on expressions

   (5.1) `aut-to-exp` <*aut*>, `aut-to-exp-DM` <*aut*>, `aut-to-exp-SO` <*aut*>
   (5.2) `expand` <*exp*>
   (5.3) `exp-to-aut` <*exp*>

The following function is not implemented. It is just convenient to *describe specification* of 'dual' functions in this section. It differs from `transpose` as it has no effect on the labels.

```
$ vcsn reverse a.xml > b.xml
$
```

Reverses every edge of the underlying graph of the automaton *a.xml*, as well as exchanges the initial and final edges and write the result in *b.xml*.

[1]Allowing some exceptions, mentioned when describing the functions.

2.1.1 Graph functions

Automata are 'labelled graphs': a number of functions on automata are indeed functions on the graph structure, irrespective of the labels.

2.1.1.1 `accessible`, `coaccessible`

```
$ vcsn accessible a.xml > b.xml[2]
$
```

Computes the accessible part of the automaton *a.xml* and writes the result in *b.xml*.

***Specification*:**

The description of the function is the specification. It is realised by a traversal of the underlying graph of *a.xml*.

```
$ vcsn coaccessible a.xml > b.xml.
$
```

Computes the co-accessible part of the automaton *a.xml* and writes the result in *b.xml*.

***Specification*:**

`coaccessible(a.xml)` = `reverse(accessible(reverse(a.xml)))`

2.1.1.2 `trim`, `is-trim`

```
$ vcsn trim a.xml > b.xml
$
```

Computes the trim part of the automaton *a.xml* and writes the result in *b.xml*.

***Specification*:**

`trim(a.xml)` = `coaccessible(accessible(a.xml))`

```
$ vcsn  -v is-trim a.xml
Input is not trim
```

Tells whether or not the automaton *a.xml* is trim.

***Specification*:**

`is-trim(a.xml)` = `is-accessible(a.xml)` $\wedge$ `is-coaccessible(a.xml)`[3]

2.1.1.3 `is-empty`

```
$ vcsn -v is-empty a.xml
Input is not empty
```

Tells whether or not the automaton *a.xml* is empty.

[2]As the functions of this section are valid for all instances of TAF-Kit 1.4, the instance in the description is shown under the generic name `vcsn`.

[3]Even if the functions `coaccessible` and `is-coaccessible` are not implemented, the specification is clear.

2.1.1.4 `is-useless`

```
$ vcsn -v is-useless a.xml
Input is has successful computations
```

Tells whether or not the automaton ***a.xml*** has successful computations.

***Specification*:**

`is-useless(a.xml)` = `is-empty(trim(a.xml))`

***Comments*:** `is-useless` is a *graph function* and tests whether there are *successful computations* in the automaton, that is a sequence of co-terminal transitions, the first one beginning in an 'initial state', the last one ending in a 'final state'. By definition, or by the way automata are specified in VAUCANSON, each of these transitions have a non-zero label. This does not imply that the label of the computation itself is different from zero, nor that the behaviour of the automaton is different from zero.

For instance, the behaviour of the $\mathbb{Z}$-automaton `usl.xml` of Figure 2.1 is the null series. Nevertheles one has:

```
$ vcsn-char-z -v is-useless usl.xml
Input has a successful computation
```

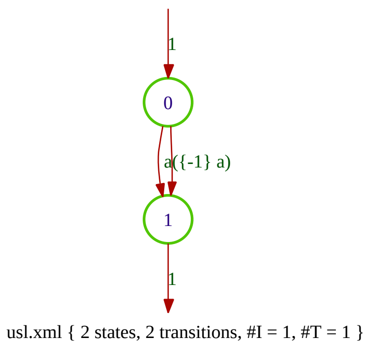


Figure 2.1: The $\mathbb{Z}$-automaton `usl.xml`

2.1.2 Transformations of automata

2.1.2.1 `is-proper`, `proper`

```
$ vcsn -v is-proper a.xml
Input is not proper
```

Tells whether or not the automaton ***a.xml*** is proper.

***Specification*:**

An automaton is *proper* if it has no *spontaneous transitions*,[4] that is, no transition labelled by the identity of the monoid (empty word for free monoids, the pair of empty words for product of free monoids). If a transition is labelled by a polynomial and not by a monomial, this means that the support of the polynomial does not contain the identity.

[4]Often called also *epsilon transitions*.

```
$ vcsn proper a.xml > b.xml
$
```

Computes a proper automaton equivalent to *a.xml* and writes the result in *b.xml*.

***Specification*:**

(i) This procedure can be called for automata of any type.

(ii) The procedure eliminates the *spontaneous transitions* of the automaton. The result may not be defined for some automata of certain type. We follow the definition taken in [7, 8] and consider that the result is defined if, and only if, the family of weights of computations labelled by the identity is *summable.*

(iii) The spontaneous-transition elimination algorithm implemented in VAUCANSON 1.4 is novel. It is valid for automata whose weight semiring is *positive* (such as $\mathbb{K} = \mathbb{B}$, $(\mathbb{Z}, \mathsf{min}, +)$, $(\mathbb{Z}, \mathsf{max}, +)$) or *ordered*, with a 'positive' part which is a subsemiring and a 'negative' part which is the opposite of tbe positive part (such as $\mathbb{K} = \mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$). Finally, the case of $\mathbb{K} = \mathbb{F}_2$ is treated separately.

Altogether, the algorithm is valid for all instances of TAF-KIT 1.4. It is (will be indeed) documented in [6] and at Section C.1.2.1.

2.1.2.2 `is-standard`, `standardize`

```
$ vcsn -v is-standard a.xml
Input is standard
```

Tells whether or not the automaton *a.xml* is standard.

***Specification*:**

An automaton is said to be *standard* if it has a *unique initial state* which is the destination of no transition and whose *initial multiplicity* is equal to the *unit* (of the multiplicity semiring).

```
$ vcsn standardize a.xml > b.xml
$
```

Transforms *a.xml* into a standard automaton and writes the result in *b.xml*.

***Specification*:**

(i) If *a.xml* is standard, *b.xml*=*a.xml*.

(ii) As a standard automaton is not necessarily proper, nor accessible, and the initial function of a state may a priori be any polynomial, `standardize` is not completely specified by the definition of standard automaton and (i) above.

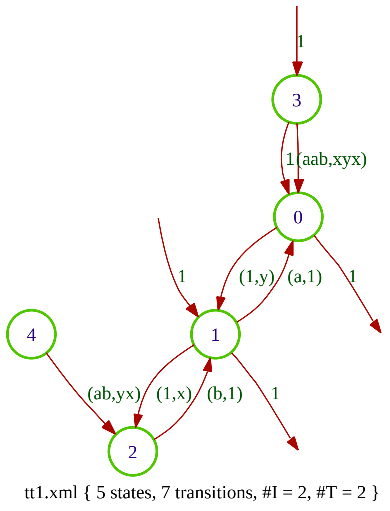
(iii) Roughly, the procedure amounts to make 'real' the *subliminal* initial state, eliminate by a *backward closure* the spontaneous transitions thus created, and suppress among the former initial states those ones that have become not accessible after the closure.

A more precise specification is given by the description of the algorithm at Section C.1.2.2.

***Example*:** Figure 2.2 shows a transducer `tt1.xml` built for the sake of the example and the result of the command:

```
$ vcsn-char-fmp-b standardize tt1.xml \| display -
```

***Comments*:** Every automaton is equivalent to a standard one.



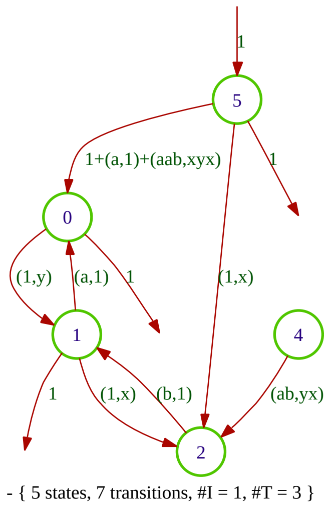


Figure 2.2: A transducer and its standardization

2.1.3 Operations on automata

2.1.3.1 `union`

```
$ vcsn union a.xml b.xml > c.xml
$
```

Builds the automaton that is the union of *a.xml* and *b.xml* and writes the result in *c.xml*.

***Precondition*:** No precondition.

2.1.3.2 `sum`

```
$ vcsn sum a.xml b.xml > c.xml
$
```

Build the automaton that is the 'sum' of *a.xml* and *b.xml* and writes the result in *c.xml*.

***Precondition*:** *a.xml* and *b.xml* are *standard* for the sum operation is defined only on standard automata.

***Specification*:**

cf. Section C.1.3.2

2.1.3.3 `concatenate`

```
$ vcsn concatenate a.xml b.xml > c.xml
$
```

Build the automaton that is the 'concatenation' of *a.xml* and *b.xml* and writes the result in *c.xml*.

***Precondition*:** *a.xml* and *b.xml* are *standard* for the concatenation operation is defined only on standard automata.

***Specification*:**

cf. Section C.1.3.3.

***Comments*:** The `concatenate` function of two automata realises the *(Cauchy) product* of their behaviours. We keep the word 'product' for a `product` function which is based on the *Cartesian product* of the automata and which realises the *intersection* of the accepted languages in the case of Boolean automata, and the *Hadamard product* of the behaviours in the general case of weigted automata (*cf.* Section 2.2.4.1).

2.1.3.4 `star`

```
$ vcsn star a.xml > b.xml
$
```

Build the automaton that is the star of *a.xml* and writes the result in *b.xml*.

***Precondition*:** *a.xml* is *standard* for the star operation is defined only on standard automata.

***Specification*:**

cf. Section C.1.3.4

2.1.3.5 `left-mult`

```
$ vcsn left-mult a.xml k > b.xml
$
```

Build the automaton that is obtained by multiplication on the left of *a.xml* by *k* and writes the result in *b.xml*.

***Precondition*:** *a.xml* is *standard* for the left 'exterior' multiplication operation is defined only on standard automata.

***Specification*:**

cf. Section C.1.3.5

***Comments*:** Beware that although the multiplication is on the left, the operand *k* is the *second* argument, and thus written on the right of *a.xml*.

2.1.3.6 `right-mult`

```
$ vcsn right-mult a.xml k > b.xml
$
```

Build the automaton that is obtained by multiplication on the right of *a.xml* by *k* and writes the result in *b.xml*.

***Precondition*:** *a.xml* is *standard* for the right 'exterior' multiplication operation is defined only on standard automata.

***Specification*:**

cf. Section C.1.3.6

***Example*:** Figure 2.3 shows the effect of a left and a right exterior multiplication on the standardization of the $\mathbb{Z}$-automaton `c1.xml`.

```
$ vcsn-char-z standardize c1.xml \| left-mult - 3 \| display -
$ vcsn-char-z standardize c1.xml \| right-mult - 5 \| display -
```

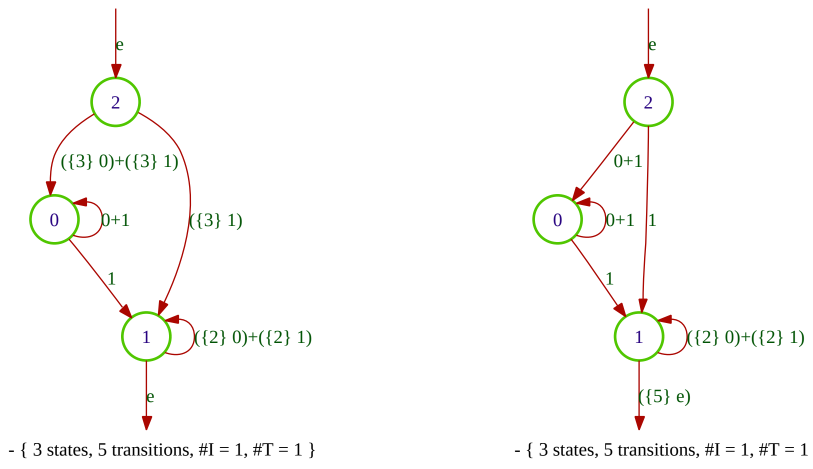

e
2
({3} 0)+({3} 1)
0
0+1
({3} 1)
1
1
({2} 0)+({2} 1)
e
- { 3 states, 5 transitions, #I = 1, #T = 1 }
e
2
0+1
0
0+1
1
1
1
({2} 0)+({2} 1)
({5} e)
- { 3 states, 5 transitions, #I = 1, #T = 1 }

Figure 2.3: Left and right multiplication on a standard $\mathbb{Z}$-automaton

2.1.3.7 `chain`

```
$ vcsn chain a.xml n > b.xml
$
```

Build the concatenation of n copies of *a.xml* by and writes the result in *b.xml*.

Precondition: *a.xml* is *standard* for the concatenation operation is defined only on standard automata.

Example: Figure 2.4 shows the effect of a concatenation of 3 copies of the standardization of the ($\mathbb{B}$-)automaton `a1.xml`.

```
$ vcsn-char-z standardize a1.xml \| chain - 3 \| display -
```

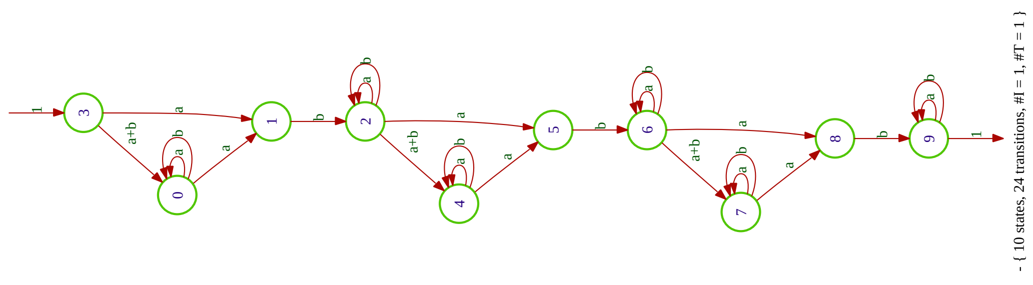


Figure 2.4: Concatenation of 3 copies of the standardization of `a1.xml`.

Comments: This function compensates for the absence of exponents in the writing of rational expressions. Note that it may easily yield large automata and entail long execution time.

```
$ vcsn-char-b -T exp-to-aut -aa '1+a' \| chain - 1000 > e.xml
Charge  id:        <name>        total      self      calls    self avg. total avg.
100.0%   0:          _program  357.13s  357.13s          1       5.95m       5.95m
```

```
 58.1%   2:     CMD[1]: chain  334.75s  207.43s         1    207.43s      5.58m
 35.6%   4:concat_of_standard  127.30s  127.30s       999    127.43ms   127.43ms
  6.3%   5:  automaton output   22.37s   22.37s         1     22.37s     22.37s
  0.0%   3:       is_standard    0.02s    0.02s      1998      0.01ms     0.01ms
  0.0%   1:CMD[0]: exp-to-aut    0.00s    0.00s         1      0.31ms     0.31ms
$ vcsn-char-b data e.xml
States: 1001
Transitions: 500500
Initial states: 1
Final states: 1001
$ vcsn-char-b -T exp-to-aut -aa 'a' \| chain - 1000 > f.xml
Charge  id:        <name>       total     self     calls   self avg. total avg.
100.0%   0:          _program  870.36ms 870.36ms        1      0.87s      0.87s
 58.4%   2:     CMD[1]: chain  814.54ms 508.55ms        1      0.51s      0.81s
 34.6%   4:concat_of_standard  300.73ms 300.73ms      999      0.30ms     0.30ms
  5.9%   5:  automaton output   51.30ms  51.30ms        1     51.30ms    51.30ms
  0.6%   3:       is_standard    5.27ms   5.27ms     1998      0.00ms     0.00ms
  0.0%   1:CMD[0]: exp-to-aut    0.21ms   0.21ms        1      0.21ms     0.21ms
$ vcsn-char-b data f.xml
States: 1001
Transitions: 1000
Initial states: 1
Final states: 1
$ vcsn-char-b concatenate e.xml f.xml > g.xml
$ vcsn-char-b -T eval g.xml 'a^1024'[5]
Charge  id:        <name>       total     self     calls   self avg. total avg.
100.0%   0:          _program  410.71s  410.71s         1      6.85m      6.85m
 67.7%   7:              eval  277.97s  277.97s         1    277.97s    277.97s
 27.7%   4:       eps_removal  113.62s  113.62s         1    113.62s    113.62s
  3.6%   2:   automaton input   14.77s   14.77s         1     14.77s     14.77s
  0.5%   1:     CMD[0]: eval   410.71s    2.12s         1      2.12s      6.85m
  0.5%   3:            cut_up    1.88s    1.88s         1      1.88s      1.88s
  0.1%   5: accessible_states    0.33s    0.33s         1      0.33s      0.33s
  0.0%   6:     sub_automaton    0.03s    0.03s         1     26.80ms    26.80ms
```

2.1.4 Operations on behaviour of automata

These functions implement somehow (one direction of) Kleene's Theorem by building standard automata which realize the rational operations on the behaviour of the parameters (the `-S` stands for 'series', as the behaviour is a series in general).

[5]Bien sûr pas sous cette forme.

2.1.4.1 `sum-S`

```
$ vcsn sum-S a.xml b.xml > c.xml
$
```

Build a standard automaton whose behaviour is the sum of the behaviours of *a.xml* and *b.xml* and writes the result in *c.xml*.

Precondition: No precondition.

Specification:

`sum-S(a.xml, b.xml)` = `sum(standardize(a.xml),standardize(b.xml))`

2.1.4.2 `cauchy-S`

```
$ vcsn cauchy-S a.xml b.xml > c.xml
$
```

Build a standard automaton whose behaviour is the (Cauchy) product of the behaviours of *a.xml* and *b.xml* and writes the result in *c.xml*.

Precondition: No precondition.

Specification:

`cauchy-S(a.xml, b.xml)` = `concatenate(standardize(a.xml),standardize(b.xml))`

Comments: The terminology used here is meant to recall that the *product* of behaviours of automata, seen as *series*, is the Cauchy product, and corresponds to the *concatenation* of automata (when they are standard automata) and *not to their product.* The latter is defined for *realtime automata* over a free monoid only (*cf.* Section 2.2.4.1).

2.1.4.3 `star-S`

```
$ vcsn star a.xml > b.xml
$
```

Build a standard automaton whose behaviour is the star of the behaviour of *a.xml* and writes the result in *b.xml*.

Precondition: No precondition.

Specification:

`star-S(a.xml)` = `star(standardize(a.xml))`

2.1.5 Automata and expressions; operations on expressions

2.1.5.1 `aut-to-exp`, `aut-to-exp-DM` <*aut*>, `aut-to-exp-SO` <*aut*>

In Vaucanson, expressions are computed from automata by the *state elimination method.* The algorithm is then specified by the *order* in which the states are eliminated. In TAF-Kit 1.4, the order is either an order computed by a heuristics called the *naive heuristics* — which is the default option —, or an order computed by a heuristics due to Delgado–Morais [3], or simply the order of the states identifiers.

```
$ vcsn aut-to-exp a.xml > e.xml
$
$ vcsn aut-to-exp-DM a.xml > e.xml
$
$ vcsn aut-to-exp-SO a.xml > e.xml
$
```

Build a rational expression which denotes the behaviour of *a.xml* and writes the result in *e.xml*.

***Precondition*:** No precondition.

***Specification*:**

cf. Section C.1.4.

***Example*:** The three orders applied to the automaton `ladybird-3.xml` (Figure 2.5) give the following results.

```
$ vcsn-char-b aut-to-exp ladybird-3.xml
a.(c.a+b+c+a.(b+c)*.(c+a).a)*.(c+a.(b+c)*.(c+a))+1
$ vcsn-char-b aut-to-exp-DM ladybird-3.xml
(a.(b+c)*.c+a.(b+c)*.a.(b+c)*.(c+a))*
$ vcsn-char-b aut-to-exp-SO ladybird-3.xml
a.(c.a+b+c)*.a.((c+a).a.(c.a+b+c)*.a+b+c)*.((c+a).a.(c.a+b+c)*.c+c+a)+a.(c.a+b+c)*.c+1
```

On this example the DM heuristics seems to be better than the naive one. They give indeed the same results in many cases (eg for `ladybird-n.xml` for $n \geqslant 4$). A thorough comparison between the two heuristics remains to be done.

The same functions apply of course to weighted automata and transducers as well.

```
$ vcsn-char-z aut-to-exp c1.xml
(0+1)*.1.(2 0+2 1)*
$ vcsn-char-fmp-b aut-to-exp t1.xml
((a,1).(1,y)+(1,x).(b,1))*.((a,1)+1)
$ vcsn-char-fmp-b aut-to-exp-SO t1.xml
((a,1).(1,y))*.(1,x).((b,1).((a,1).(1,y))*.(1,x))*.(b,1).((a,1).(1,y))*.
((a,1)+1)+((a,1).(1,y))*.((a,1)+1)
```

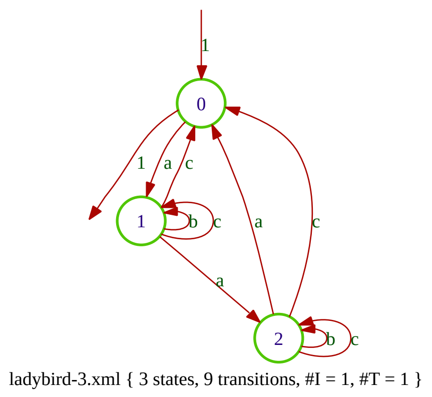


Figure 2.5: The automaton `ladybird-3.xml`

2.1.5.2 `exp-to-aut`

```
$ vcsn -ixml exp-to-aut e.xml > a.xml
$
```

Build an automaton whose behaviour is denoted by the expression *e.xml* and writes the result in *a.xml*.

***Precondition*:** no precondition.

***Specification*:**

The automaton *a.xml* is the 'standard automaton' of the expression *e.xml*, computed by the recursive use of the operations on automata, as described at Section 2.1.3 and as specified at Section **??**.

For the specification of the expression formats, *cf.* Section 1.3.3.2.

***Caveat*:** (i) For technical reasons, the `exp-to-aut` function *is not implemented* for the *fmp* instances, that is, for transducers, in TAF-Kit 1.4.
(ii) The actual implementation of `exp-to-aut` carries out first a 'letterization' of the expression, which is not necessary in principle. As it is, it is completely synonymous to the `standard` function (*cf.* Section 2.2.3). This is one of the reasons for which it is not implemented for the *fmp* instances.

***Example*:** The `exp-to-aut` function is not implemented for transducers, but is for weigted automata, as shown at Figure 2.6, result of the following command (*cf.* [7, Exer. III.2.24]).

```
$ vcsn-char-q -aab exp-to-aut '(1/6a* + 1/3b*)*' \| display -
```

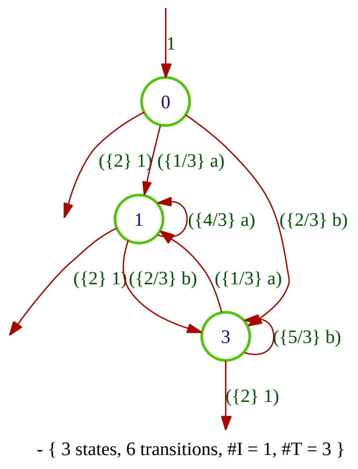


Figure 2.6: A standard $\mathbb{Q}$-automaton built by `exp-to-aut`

2.1.5.3 `expand`

```
$ vcsn -ixml -oxml expand e.xml > f.xml
$
```

Expands the expression *e.xml* and writes the result in *a.xml*.

***Specification*:**

Distribute product over addition. For the specification of the expression formats, *cf.* Section 1.3.3.2.

Caveat: Not implemented for the *fmp* instances, that is, for expressions over a direct product of free monoids.

Example:

```
$ vcsn-char-b -aabc expand '(a+b+1)((a+ba)(ca+cc))*'
a.(aca+acc+baca+bacc)*+b.(aca+acc+baca+bacc)*+(aca+acc+baca+bacc)*
```

Comment for the VAUCANSON *Group* (110626): *Fonction de service, pour présenter certains résultats de façon plus naturelle et lisible.*

Dans mon souvenir, la distributivité s'effectuait récursivement de gauche à droite, jusqu'à la première sous-expression étoilée, et s'arrêtait alors, sans franchir cette sous-expression et sans rentrer dedans.

Je pense que cette fonction peut avoir différents comportements, l'actuel, celui décrit précédemment, et peut-être d'autres, commandés par un paramètre. Il faudrait aller y voir de plus près.

2.2 Weighted automata and expressions over free monoids

The following functions concern automata over a free monoid — as opposed to automata over a direct product of free monoids. *A priori*, there is no assumption on the multiplicity semiring. However, in Vaucanson 1.4, TAF-Kit gives access to automata with weight in 'numerical' *commutative* semirings only.

The next two sections, Section 2.3 and Section 2.4, will describe functions that are special to automata with multiplicity in a field ($\mathbb{R}$ and Q) and in $\mathbb{B}$ respectively.

1. Properties and transformations of automata

   (1.1) `transpose` <*aut*>

   (1.2) `is-realtime` <*aut*>, `realtime` <*aut*>

   (1.3) `is-unambiguous` <*aut*>

   (1.4) `quotient` <*aut*>

   (1.5) `partial-identity` <*aut*>

2. Behaviour of automata

   (2.1) `eval` <*aut*> <*word*>

   (2.2) `eval-S` <*aut*> <*word*>

3. From expressions to automata

   (3.1) `standard` <*exp*>

4. Operations on automata and their behaviour

   (4.1) `product` <*aut1*> <*aut2*>

   (4.2) `power` <*aut*> <*n*>

   (4.3) `shuffle` <*aut1*> <*aut2*>, `infiltration` <*aut1*> <*aut2*>

2.2.1 Properties and transformations of automata

The following function is not implemented. It is just convenient to *describe the specification* of `realtime`.

```
$ vcsn letterize a.xml > b.xml
$
```

Computes from *a.xml* an equivalent automaton whose transitions are all labelled by letters or the empty word, by cutting the label of every transition into letters and writes the result in *b.xml*.

2.2.1.1 `transpose`

```
$ vcsn transpose a.xml > b.xml
$
```

Computes the transposition of the automaton *a.xml* and writes the result in *b.xml*.

***Specification*:**

Builds the transposition of the underlying graph, and *exchanges* the initial and final functions (that is, realises the function `reverse` (*cf.* Section 2.1). Finally, transposes the labels as well, that is, takes the *mirror* image of the words that label the transitions *and* in the initial and final functions.[6]

***Comments*:** (i) The behaviour of $\mathcal{A}^{\mathrm{t}}$, the tranpose of $\mathcal{A}$, is the transpose of the behaviour of $\mathcal{A}$.
(ii) There exists a `transpose` function for transducers (`fmp`) as well, that will be redefined explicitely for them (*cf.* Section 2.5.1.2).

2.2.1.2 `is-realtime`, `realtime`

```
$ vcsn is-realtime -v a.xml
Input is realtime
```

Tells whether or not the automaton *a.xml* is realtime.

***Specification*:**

An automaton (over a free monoid) is realtime if it is both letterized and proper.

```
$ vcsn realtime a.xml > b.xml
$
```

Computes from *a.xml* an automaton by eliminating the spontaneous transitions from the letterized version of *a.xml* and writes the result in *b.xml*.

***Specification*:**

`realtime(a.xml)` = `proper(letterize(a.xml))`

***Comments*:** (i) The problem with `realtime` is the same as the one of `proper` and has been mentioned at Section 2.2.4.1.
(ii) `letterize(proper(a.xml))` is another realtime automaton, which has potentially many more states and transitions than `realtime(a.xml)`.

2.2.1.3 `is-unambiguous`

```
$ vcsn -v is-unambiguous a.xml
Input is unambiguous
```

Tells whether or not the automaton *a.xml* is unambiguous.

***Precondition*:** *a.xml* is a *realtime* automaton.

***Specification*:**

An automaton is *unambiguous* if every word accepted by the automaton is the label of *only one* successful computation.

***Comments*:** *An automaton $\mathcal{A}$ is ambiguous if, and only if, the trim part of the product $\mathcal{A}\times\mathcal{A}$ contains a state outside of the diagonal.*

[6]Such automata can be built by using the `edit` or the `image` functions for instance.

2.2.1.4 quotient

```
$ vcsn quotient a.xml > b.xml
$
```

Computes the quotient of *a.xml* and writes the result in *b.xml*.

Precondition: *a.xml* is a *realtime* automaton.

Specification:

The `quotient` function implements an iterative refinement of equivalences over states (by a variant of Hopcroft's algorithm) *cf.* Section C.2.1.1.

```
$ vcsn-char-z -T power c1.xml 10 \| quotient - \| display -
Taf-kit command bench
...
[Task list:]
Charge  id:        <name>        total     self      calls   self avg. total avg.
100.0%   0:            _program   20.15s   20.15s         1     20.15s     20.15s
 93.2%   7:         eps_removal   18.78s   18.78s         1     18.78s     18.78s
  1.0%  10:            quotient    0.23s    0.21s         1      0.21s      0.23s
  1.5%   6:              cut_up    0.31s    0.31s         1      0.31s      0.31s
  1.5%   3:             product    0.31s    0.30s         9     32.89ms    33.89ms
  1.2%   5:  CMD[1]: quotient   19.61s    0.25s         1      0.25s     19.61s
  1.1%   1:     CMD[0]: power    0.54s    0.23s         1      0.23s      0.54s
  0.2%   8: accessible_states    0.04s    0.04s         1     36.56ms    36.56ms
  0.2%   4:is_realtime (autom    0.03s    0.03s        19      1.62ms     1.62ms
  0.0%   9:       sub_automaton    0.01s    0.01s         1      6.97ms     6.97ms
  0.0%   2:     automaton input    0.01s    0.01s         1      5.12ms     5.12ms
  0.0%  11:    CMD[2]: display    0.00s    0.00s         1      0.95ms     1.46ms
  0.0%  12:    automaton output    0.00s    0.00s         1      0.52ms     0.52ms
```

2.2.1.5 partial-identity

```
$ vcsn partial-identity a.xml > t.xml
$
```

Transforms the automaton *a.xml* over A^* into an automaton over $A^* \times A^*$ (a `fmp-transducer`) which realises the identity on the behaviour of *a.xml* and writes the result in *t.xml*.

Precondition: no precondition.

Specification:

Every transition of *t.xml* is obtained from a transition of *a.xml* by keeping the same weight and by replacing the label f by the pair (f, f).

Example:

```
$ vcsn-char-z partial-identity c1.xml > c1pi.xml
$ vcsn-char-fmp-z display c1pi.xml
```

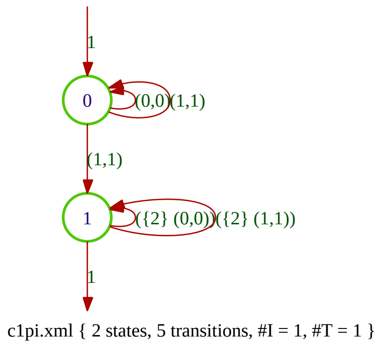


Figure 2.7: A weighted partial identity

***Caveat*:** (i) The `partial-identity` function is implemented for the TAF-Kit instances `vcsn-char-b`, `vcsn-int-b`, `vcsn-char-z`, et `vcsn-int-z` only, so that the type of the result matches an implemented instance for *fmp*.
(ii) As the type of the result is different from the type of the input, it is not possible to use the intern pipe to chain the functions.

2.2.2 Behaviour of automata

The function `aut-to-exp` (*cf.* Section 2.1.5.1) applies to these automata.

2.2.2.1 `eval`

```
$ vcsn eval a.xml 'word'
<value>
```

Computes the coefficient of the word *`word`* in the series realized by *`a.xml`*.

***Precondition*:** (i) *`a.xml`* is realtime.
(ii) *`word`* is a sequence of letters in the input alphabet of *`a.xml`* (the generators of A^*).

***Example*:**

```
$ vcsn-char-z power c1.xml 10 > c10.xml
$ vcsn-char-z eval c10.xml '10'
1024
```

***Caveat*:** The parameter *`word`* must be a sequence of letters, and not an expression which denotes a word.

```
$ vcsn-char-z eval c10.xml '1 0'
FATAL: Cannot parse 1 0
```

***Comments*:** Not so trivial algorithm (*cf.* Section **??**).

2.2.2.2 eval-S

```
$ vcsn eval-S a.xml 'word'
<value>
```

Computes the coefficient of the word *word* in the series realized by *a.xml*.

***Precondition*:** (i) No condition on *a.xml*.
(ii) As for `eval`, *word* is a sequence of letters in the input alphabet of *a.xml*.

***Specification*:**

`eval-S(a.xml,`*word*`)` = `eval(realtime(a.xml),`*word*`)`.

2.2.3 From expressions to automata

2.2.3.1 standard

```
$ vcsn standard e.xml > a.xml
$
```

Computes *the* standard automaton of *e.xml* and writes the result in *a.xml*.

***Specification*:**

We call *standard automaton* what is often called in the literature *Glushov automaton* or *position automaton* of the expression that is thus understood to be 'letterized' (even if it not necessarily so in VAUCANSON 1.4).

***Comments*:** In TAF-KIT 1.4, the `standard` function is synonymous to `exp-to-aut`, or to be more precise, the `exp-to-aut` function is synonymous to `standard` (*cf.* Section 2.1.5.2).

2.2.4 Operations on automata and their behaviour

2.2.4.1 product

```
$ vcsn product a.xml b.xml > c.xml
$
```

Computes the product of *a.xml* and *b.xml* and writes the result in *c.xml*.

***Precondition*:** (i) *a.xml* and *b.xml* are *realtime* automata.
(ii) This operation requires, to be meaningful, that the weight semiring be *commutative*, and this is the case for all the instances implemented in TAF-KIT 1.4.

***Specification*:**

The product of *a.xml* and *b.xml* is, by definition, the *accessible part* of the cartesian product of the two automata whose transitions are defined by

$$\forall p,q \in \mathcal{A}\,,\ \forall r,s \in \mathcal{B} \qquad p \xrightarrow[\mathcal{A}]{a|k} q \quad \text{and} \quad r \xrightarrow[\mathcal{B}]{a|h} s \qquad \Longrightarrow \qquad (p,r) \xrightarrow[\mathcal{A}\times\mathcal{B}]{a|kh} (q,s)$$

and the initial and final functions by

$$\forall p \in \mathcal{A}\,,\ \forall r \in \mathcal{B} \qquad I(p,r) = I(p)\,I(r) \quad \text{and} \quad T(p,r) = T(p)\,T(r)\ .$$

***Comments*:** (i) The result *c.xml* is a realtime automaton.
(ii) In terms of *representations*, the representation of the product is the *tensor product* of the representations of the operands.

2.2.4.2 power

```
$ vcsn power a.xml n > d.xml
$
```

Computes the product of *a.xml* by itself *n* times and writes the result in *d.xml*.

***Precondition*:** (i) *a.xml* is *realtime.*
(ii) This operation requires, to be meaningful, that the weight semiring be *commutative*, and this is the case for all the instances implemented in the TAF-Kit 1.4.

2.2.4.3 shuffle

```
$ vcsn shuffle a.xml b.xml > c.xml
$
```

Computes the shuffle of *a.xml* and *b.xml* and writes the result in *c.xml*.

***Precondition*:** (i) *a.xml* and *b.xml* are *realtime* automata.
(ii) This operation requires, to be meaningful, that the weight semiring be *commutative.*

***Specification*:**

The shuffle of *a.xml* and *b.xml* is, by definition, the *accessible part* of the automaton whose set of states is the cartesian product of the sets of states of the two automata and whose transitions are defined by

$$\forall p,q \in \mathcal{A}\,,\ \forall r \in \mathcal{B} \qquad p \xrightarrow[\mathcal{A}]{a|k} q \qquad \Longrightarrow \qquad (p,r) \xrightarrow[\mathcal{A}\between\mathcal{B}]{a|kh} (q,r)$$

$$\forall p \in \mathcal{A}\,,\ \forall r,s \in \mathcal{B} \qquad r \xrightarrow[\mathcal{B}]{a|h} s \qquad \Longrightarrow \qquad (p,r) \xrightarrow[\mathcal{A}\between\mathcal{B}]{a|kh} (p,s)$$

and the initial and final functions by

$$\forall p \in \mathcal{A}\,,\ \forall r \in \mathcal{B} \qquad I(p,r) = I(p)\,I(r) \quad \text{and} \quad T(p,r) = T(p)\,T(r)\,.$$

2.2.4.4 infiltration

```
$ vcsn infiltration a.xml b.xml > c.xml
$
```

Computes the infiltration of *a.xml* and *b.xml* and writes the result in *c.xml*.

***Precondition*:** (i) *a.xml* and *b.xml* are *realtime* automata.
(ii) This operation requires, to be meaningful, that the weight semiring be *commutative.*

***Specification*:**

The infiltration of *a.xml* and *b.xml* is, by definition, the *accessible part* of the automaton whose set of states is the cartesian product of the sets of states of the two automata and whose transitions are those of the product and of the shuffle.

2.3 Automata and rational expressions on free monoids with weights in a field

Three instances of TAF-Kit 1.4 implement a weight semiring which is a *field*: `vcsn-char-q`, `vcsn-char-r`, and `vcsn-char-f2`, for which the weight semiring is $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$ respectively (*cf.* Section 1.2.2). In addition to all the functions of the preceding section which obviously apply, a function `reduce` is specific those automata whose weight semiring is a field. It then easily allows to test the *equivalence* of two automata or expressions.

1. Operations on automata

   (1.1) `reduce` <*aut*>

   (1.2) `are-equivalent` <*aut1*> <*aut2*>

2. Operations on expressions

   (2.1) `are-equivalent-E` <*exp1*> <*exp2*>

2.3.1 Operations on automata

2.3.1.1 `reduce`

```
$ vcsn reduce a.xml > b.xml
$
```

Computes from *a.xml* an equivalent automaton of minimal dimension and writes the result in *b.xml*.

Precondition: *a.xml* is realtime.

Comments: Implements Schützenberger algorithm for reduction of representations (*cf.* Section C.3).

```
$ vcsn-char-r power c1.xml 5 \| reduce -  \| data -
States: 7
Transitions: 56
Initial states: 1
Final states: 1
$ vcsn-char-q power c1.xml 5 \| reduce -  \| data -
States: 7
Transitions: 49
Initial states: 1
Final states: 1

$ vcsn-char-r power c1.xml 2 \| quotient - \| reduce - > c2qr.xml
$ vcsn-char-q display c2qr.xml
$ vcsn-char-q eval c2qr.xml 'bbba'
196
$ vcsn-char-q eval c2qr.xml 'bbbb'
-15/286331153
```

```
$ vcsn-char-q eval c2qr.xml 'baaaaaaaaaaaaaaa'
1073741824
$ vcsn-char-q eval c2qr.xml 'baaaaaaaaaaaaaaaa'
0
```

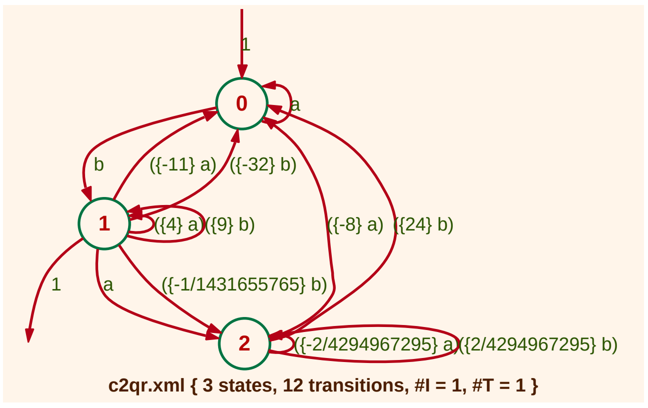


Figure 2.8: L'automate `c2qr.xml`

2.3.1.2 `are-equivalent`

```
$ vcsn -v are-equivalent a.xml b.xm
Automata are not equivalent
```

Tells whether or not the automata *a.xml* and *b.xml* realize the same series.

Precondition: no precondition.

Specification:

`are-equivalent(a.xml,b.xml)` =
  `is-useless(reduce(sum(realtime(a.xml),left-mult(realtime(b.xml),-1))))`

```
$ vcsn-char-r power c1.xml 3 \| quotient - > c3q.xml
$ vcsn-char-r power c1.xml 3 \| transpose - \| quotient - \| transpose - > c3cq.xml
$ vcsn-char-q -v are-equivalent c3q.xml c3cq.xml
Automata are not equivalent
```

2.3.2 Operations on expressions

2.3.2.1 `are-equivalent-E`

```
$ vcsn -v -ixml are-equivalent-E e.xml f.xml
Expressions are equivalent
```

Tells whether or not the expressions *e.xml* and *f.xml* denote the same language.

Specification:

`are-equivalent-E(e.xml,f.xml)` = `are-equivalent(standard(e.xml),standard(f.xml))`

Caveat: The specifications for the input format of rational expressions apply for this function.

Example:

```
$ vcsn-char-q -aab -v are-equivalent-E 'b*((2a).b*)*' '((2a)*b)*(2a)*'
Expressions are equivalent
```

2.4 Boolean automata and rational expressions on free monoids

The classical theory of automata has been developed for automata with no weight, that is, with weight taken in the Boolean semiring. All functions of Section 2.1 and Section 2.2 obviously apply. But a number of other functions, very important ones indeed, are specific to Boolean automata.

1. Operations on automata

   (1.1) `is-complete` <*aut*>, `complete` <*aut*>

   (1.2) `is-deterministic` <*aut*>, `determinize` <*aut*>

   (1.3) `complement` <*aut*>

   (1.4) `minimize` <*aut*>

   (1.5) `prefix` <*aut*>, `suffix` <*aut*>, `factor` <*aut*>

2. Operations on behaviours of automata

   (2.1) `enumerate` <*aut*>

   (2.2) `shortest` <*aut*>

   (2.3) `intersection` <*aut1*> <*aut2*>

   (2.4) `are-equivalent` <*aut1*> <*aut2*>

3. Operations on expressions

   (3.1) `derived-term` <*exp*>

   (3.2) `are-equivalent-E` <*exp1*> <*exp2*>

***Comments*:** For clarifying specifications, we make use of some specific automata:
- $\mathcal{V}$ is the empty automaton (no state);
- $\mathcal{W}$ is the one-state automaton, where the unique state is initial but not final, and is both the source and the target of a transition labeled by every letter of the alphabet.

2.4.1 Operations on automata

2.4.1.1 `is-complete`, `complete`

```
$ vcsn -v is-complete a.xml
Input is complete
```

Tells whether or not the automaton `a.xml` is complete.

***Precondition*:** `a.xml` is realtime.

***Specification*:**

A realtime automaton `a.xml` over the alphabet A is *complete* if
(a) it has at least one initial state;

(b) every state of *a.xml* is the origin of at least one transition labelled by a, for every a in A.

***Comments*:** As a consequence of the specifications, every word of A^* is the label of at least one computation in *a.xml* (characteristic property which makes (a) necessary), possibly a not successful one.
(i) The definition thus depends not only on *a.xml* itself, but also on the alphabet on which *a.xml* is constructed. Or, to tell it in another way, not only on the *value* of the automaton, but also on its *type*.
(ii) The empty automaton $\mathcal{V}$ is *not complete*.
(iii) Once the definition is written down, it appears that it could be taken for automata over a free monoid in general, and not only for Boolean automata. It is the *function* `complete()` which would be meaningless, or, at least, artifical.
(iv) One must acknowledge that the definition is rather artifical also for automata which are not *accessible*.

```
$ vcsn complete a.xml > b.xml
$
```

Computes from *a.xml* an equivalent complete automaton and writes the result in *b.xml*.

***Precondition*:** *a.xml* is realtime.

***Specification*:**

If *a.xml* is not complete,
(a) add a new state z to *a.xml*;
(b) for every state p of *a.xml* (including z), and for every a in A, if there exist no transition (p, a, q) in *a.xml*, add a transition (p, a, z) to *a.xml*;
(c) if there exist no initial state in *a.xml*, make z initial.

***Comments*:** `complete(`$\mathcal{V}$`)` $= \mathcal{W}$.

2.4.1.2 `is-deterministic`, `determinize`

```
$ vcsn is-deterministic -v a.xml
Input is not deterministic
```

Tells whether or not the automaton *a.xml* is deterministic.

***Precondition*:** *a.xml* is realtime.

***Specification*:**

A realtime automaton *a.xml* over the alphabet A is *deterministic* if
(a) it has at most one initial state;
(b) every state of *a.xml* is the origin of at most one transition labelled by a, for every a in A.

***Comments*:** As a consequence, every word of A^* is the label of at most one computation in *a.xml* (characteristic property which makes (a) necessary).
(i) The result depends indeed only on *a.xml* itself, not on its *type*.
(ii) The empty automaton $\mathcal{V}$ is *deterministic*.

```
$ vcsn determinize a.xml > b.xml
$
```

Computes the ‘determinisation’ of *a.xml* and writes the result in *b.xml*.

***Precondition*:** *a.xml* is realtime.

***Specification*:**

Computes the accessible part of the ‘subset automaton’, an algorithm sometimes refered to as ‘the subset construction’. The result is thus *accessible* and *complete*.

determinize($\mathcal{V}$) $= \mathcal{W}$

2.4.1.3 complement

```
$ vcsn complement a.xml > b.xml
$
```

Computes the ‘complement automaton’ of *a.xml* and writes the result in *b.xml*.

***Precondition*:** *a.xml* is complete (thus realtime) and deterministic.

***Specification*:**

Swap terminal for non-terminal states in *a.xml*.

***Comments*:** Thanks to the preconditions, the language accepted by `complement(a.xml)` is the complement of the language accepted by *a.xml*.

2.4.1.4 minimize

```
$ vcsn minimize a.xml > b.xml
$
```

Computes the ‘minimized automaton’ of *a.xml* and writes the result in *b.xml*.

***Precondition*:** *a.xml* is complete (thus realtime) and deterministic.

***Specification*:**

`minimize(a.xml)` = `quotient(a.xml)`.

***Comments*:** Thanks to the preconditions,
(a) `minimize(a.xml)` is *the minimal automaton* of the language accepted by *a.xml*.
(b) a variant of the quotient algorithm (known as *Hopcroft algorithm*) can be used and is indeed implemented in TAF-Kit (*cf.* Section C.4.1.1)

2.4.1.5 prefix, suffix, factor

```
$ vcsn prefix a.xml > b.xml
$
```

Makes every state of *a.xml* final and writes the result in *b.xml*.

***Precondition*:** *a.xml* is *realtime* and *trim*.

***Comments*:** Thanks to the preconditions, *b.xml*= `prefix(a.xml)` is an automaton which accepts all prefixes of words in the language accepted by *a.xml*.

```
$ vcsn suffix a.xml > b.xml
$
```

Makes every state of *a.xml* initial and writes the result in *b.xml*.

Precondition: *a.xml* is *realtime* and *trim*.

Comments: Thanks to the preconditions, *b.xml*= `suffix(a.xml)` is an automaton which accepts all suffixes of words in the language accepted by *a.xml*.

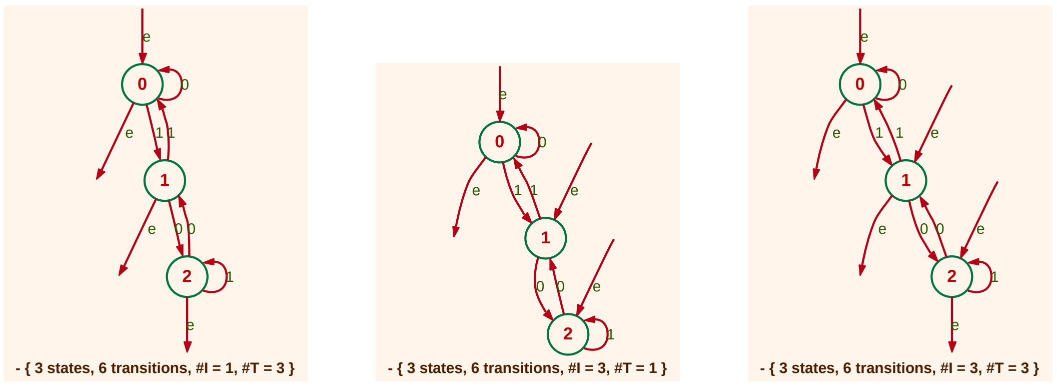
```
$ vcsn factor a.xml > b.xml
$
```

Makes every state of *a.xml* initial and final and writes the result in *b.xml*.

Precondition: *a.xml* is *realtime* and *trim*.

Comments: Thanks to the preconditions, *b.xml*= `factor(a.xml)` is an automaton which accepts all factors of words in the language accepted by *a.xml*.

Example: Figure 2.9 shows the automata for the prefixes, suffixes, and factors of `div3base2.xml`.



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

Figure 2.9: Automata for the prefixes, suffixes, and factors of `div3base2.xml`

2.4.2 Behaviour of automata

2.4.2.1 `enumerate`

```
$ vcsn enumerate a.xml  n
< list of words >
```

Computes the list of the words of length less than or equal to *n* in the support of the series realized by *a.xml*.

Precondition: *a.xml* is realtime.

Specification:

(i) The words are enumerated in the radix ordering, and output as one word per line.
(ii) If `is-useless(a.xml)`, then the list is empty.

Example: The next command enumerates the words with an even number of `a`'s.

```
$ vcsn enumerate apair.xml 3
1
b
aa
bb
```

```
aab
aba
baa
bbb
```

2.4.2.2 `shortest`

```
$ vcsn shortest a.xml
< word >
```

Computes the shortest word in the support of the series realized by *a.xml*.

***Precondition*:** *a.xml* is realtime.

***Specification*:**

If `is-useless(a.xml)`, exits with a nonzero `exit` code.

2.4.3 Operations on behaviours of automata

2.4.3.1 `intersection`

```
$ vcsn intersection a.xml b.xm > c.xml
$
```

Computes from *a.xml* and *b.xml* an automaton which accepts the intersection of the languages accepted by *a.xml* and *b.xml* and writes the result in *c.xml*.

***Precondition*:** no precondition.

***Specification*:**

`intersection(a.xml,b.xml)` = `product(realtime(a.xml),realtime(b.xml))`

2.4.3.2 `are-equivalent`

```
$ vcsn are-equivalent -v a.xml b.xm
Automata are not equivalent
```

Tells whether or not the automata *a.xml* and *b.xml* accept the same language.

***Precondition*:** no precondition.

***Specification*:**

```
are-equivalent(a.xml,b.xml) =
  is-useless(intersection(a.xml, complement(determinize(realtime(b.xml)))))
    ∧ is-useless(intersection(complement(determinize(realtime(a.xml))),b.xml))
```

2.4.4 Operations on expressions

2.4.4.1 `derived-term`

```
$ vcsn derived-term e.xml > a.xml
$
```

Computes the derived term automaton of *e.xml* and writes the result in *a.xml*.

***Precondition*:** no precondition.

***Specification*:**

The precise specification of `derived-term` is to be found elsewhere.

***Caveat*:** The specifications for the input format of rational expressions apply for this function.

***Example*:** As shown with the next commands and Figure 2.10, the automaton `div3base2.xml` yields again a good example (*cf.* [7, Exer. I.5.5]).

```
$ vcsn-char-b aut-to-exp-SO div3base2.xml
0*.1.(1.0*.1)*.0.(0.(1.0*.1)*.0+1)*.0.(1.0*.1)*.1.0*+0*.1.(1.0*.1)*.1.0*+0*
$ vcsn-char-b aut-to-exp-SO div3base2.xml \| derived-term - \| display -
```

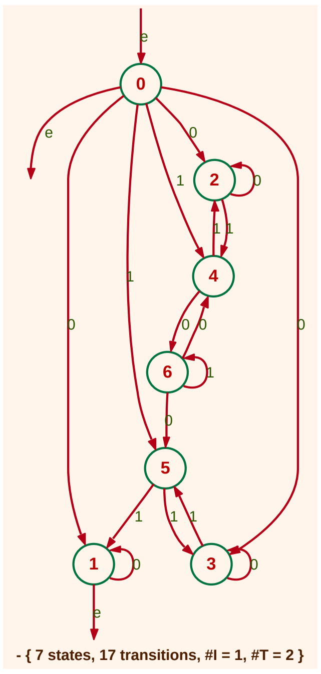


Figure 2.10: The derived term automaton of an expression computed from `div3base2.xml`

***Comments*:** The definition of the derived term automaton of an expression in the Boolean case is due to Antimirov [1]. The computation of the derived terms of an expression in VAUCANSON 1.4 implements the ideas introduced in [2] (*cf.* Section C.4.3.1).

The derived term automaton of an expression can be defined for weighted expressions as well and not only for Boolean expressions (*cf.* [5]). This is not implemented in VAUCANSON 1.4 (but will be in subsequent versions of VAUCANSON).

2.4.4.2 `are-equivalent-E`

```
$ vcsn -v -ixml are-equivalent-E e.xml f.xm
Expressions are equivalent
```

Tells whether or not the expressions *e.xml* and *f.xml* denote the same language.

***Precondition*:** no precondition.

***Specification*:**

`are-equivalent-E(e.xml,f.xml)` = `are-equivalent(standard(e.xml),standard(f.xml))`

***Caveat*:** The specifications for the input format of rational expressions apply for this function.

2.5 Weighted automata over a product of two free monoids

Automata over a product of (two) free monoids are called *transducers* in the literature and `fmp-transducers` in VAUCANSON, '`fmp`' stands for *free monoid product.* Their behaviours are series over $A^*\times B^*$, that is, weighted subsets of $A^*\times B^*$, or weighted relations from A^* (input monoid) to B^* (output monoid) but looked at rather symmetrically.

Transducers can also be considered as automata over the input alphabet with multiplicity in the semiring of (rational) series over the output alphabet (the equivalence between the two points of view is asserted by the Kleene-Schützenberger Theorem). These would be called `rw-transducers` in VAUCANSON, '`rw`' stands for *rational weights* and are not implemented in TAF-KIT VAUCANSON 1.4 (*cf.* Section 1.2.2). They will be implemented in subsequent versions of VAUCANSON.

In the sequel, we denote the input monoid by A^*, the output monoid by B^* — in TAF-KIT 1.4, they are both alphabets of characters or both alphabets of integers — and the weight semiring (numerical, and commutative) by $\mathbb{K}$ — in TAF-KIT 1.4, $\mathbb{B}$ or $\mathbb{Z}$. We denote the transducers by ***tdc*** rather than by ***aut***.

In theory, all functions of Section 2.1 should apply. But few, which involve reading rational expressions: `cat-E`, `exp-to-aut`, are not implemented in TAF-KIT 1.4. A number of functions are specific to transducers, and described in this section.

1. Transformations of transducers

   (1.1) `inverse` <***tdc***>

   (1.2) `transpose` <***tdc***>

   (1.3) `is-subnormalized` <***tdc***>, `subnormalize` <***tdc***>

   (1.4) `is-ltl` <***tdc***>

   (1.5) `ltl-to-pair` <***tdc***>

2. Operations on transducers

   (2.1) `domain` <***tdc***>, `image` <***tdc***>

   (2.2) `composition` <***tdc1***> <***tdc2***>

   (2.3) `evaluation` <***tdc***> <***aut***>

   (2.4) `eval` <***tdc***> <***word***>

3. Operations on behaviours of transducers

   (3.1) `composition-R` <***tdc1***> <***tdc2***>

   (3.2) `evaluation-S` <***tdc***> <***aut***>

Comment for the VAUCANSON *Group*: *Bilan de la réunion du 2/12/10:*

(i) *Pas implémentées, et ne le seront pas rapidement parce que pas urgent ou important:* ***W-image;***

(ii) *Pas encore implémentées mais le seront peut-être:* ***eval, composition-R, evaluation-S;***

2.5.1 Transformations of transducers

2.5.1.1 inverse

```
$ vcsn inverse t.xml > u.xml
$
```

u.xml realizes what is called the *inverse relation* of the relation realized by *t.xml*

Precondition: no precondition.

Specification:

Swaps the first for the second component in the labels of the transitions of the transducer *t.xml* and writes the result in the transducer *u.xml*.

Comments: `inverse(t.xml)` is kind of pivotal function and will have an influence on the specification of other functions.

2.5.1.2 transpose

```
$ vcsn transpose t.xml > u.xml
$
```

Computes the transposition of the transducer *t.xml* and writes the result in the transducer *u.xml*.

Precondition: no precondition.

Specification:

(i) Builds the transposition of the underlying graph.
(ii) Transposes the labels of the transitions thanks to the extension of the function `transpose()` from words to pair of words:
`transpose((f,g))= (transpose(f),transpose(g))`.

2.5.1.3 is-subnormalized, subnormalize

```
$ vcsn is-subnormalized -v t.xml
Input is not subnormalized
```

Tells whether or not the transducer *t.xml* is subnormalized.

Specification:

A transducer is *subnormalized* if it is

1. *proper*;
2. weakly 'letterized', *in the sense* that the labels of transitions are either in $(A \times 1_{B^*})$ or in $(1_{A^*} \times B)$, or in $(A \times B)$;
3. initial and final functions are *scalar*, that is, take values in the weight semiring.

Comments: The terminology 'subnormalized' is new and comes from 'normalized', which means that the labels of transitions are either in $(A \times 1_{B^*})$ or in $(1_{A^*} \times B)$. The terminology 'normalized' is not so good, as it collides with the notion of normalized automata, but is widely accepted and used. Once 'normalized' is accepted, 'subnormalized' is not so bad. Other suggestions are still welcome: no established terminology exists.

```
$ vcsn subnormalize t.xml > u.xml
$
```

Computes from *t.xml* a subnormalized transducer and writes the result in *u.xml*.

***Precondition*:** no precondition.

***Specification*:**

1. As for `proper` above, one wants to 'letterize' first, and then eliminate the spontaneous transitions.
2. We are to 'letterize' monomials such as $\mathtt{m} = \{\mathtt{k}\}(\mathtt{f},\mathtt{g})$ with f in A^* and g in B^*.

   A monomial of the form $\{\mathtt{k}\}(\mathtt{abc},\mathtt{xy})$ will be decomposed in the product of $n = \mathtt{sup}(|f|,|g|)$ 'generators' in the following way:

   $$(\{\mathtt{k}\}(\mathtt{a},\mathtt{x}))\,(\mathtt{b},\mathtt{y})\,(\mathtt{c},\mathtt{1})$$

3. create $n-1$ states between the origin and the end of the transition labeled by the monomial and the n transitions such that each of them is labeled by one of the generators we have computed in the above decomposition, the first one being possibly weighted.
4. eliminate the spontaneous transitions with a 'backward' procedure.

***Comments*:** The function `subnormalize` is only a 'decomposition' algorithm; it does not attempt to make the automaton more compact: this would be the task of other, and more sophisticated, algorithms.

2.5.1.4 `is-ltl`

```
$ vcsn is-ltl -v t.xml
Input is letter-to-letter
```

Tells whether or not the label of every transition of *t.xml* is in $A{\times}B$.

2.5.1.5 `ltl-to-pair`

```
$ vcsn ltl-to-pair t.xml > a.xml
$
```

Transforms *t.xml* into an automaton over $(A{\times}B)^*$ with weight in $\mathbb{K}$ and writes the result in *a.xml*.

***Precondition*:** *t.xml* is letter-to-letter.

***Specification*:**

The label of every transition of *t.xml* becomes *a letter* in the alphabet $(A{\times}B)$ and the weight of the transition is kept unchanged.

2.5.2 Operations on transducers

2.5.2.1 `domain`, `image`, `W-image`

```
$ vcsn domain t.xml > a.xml
$
```

Forgets the second component of the label *and the weight* of the transitions of the transducer *`t.xml`* and writes the result in the *Boolean automaton* *`a.xml`* on A^*.

***Precondition*:** no precondition.

```
$ vcsn image t.xml > b.xml
$
```

Forgets the first component of the label *and the weight* of the transitions of the transducer *`t.xml`* and writes the result in the *Boolean automaton* *`b.xml`* on B^*.

***Precondition*:** no precondition.

***Comments*:** The specification for `image` is taken so that the following identities hold:
`image(t.xml)` = `domain(inverse(t.xml))` and
`domain(t.xml)` = `image(inverse(t.xml))`.

```
$ vcsn W-image t.xml > c.xml
$
```

Forgets the first component of the label (and *not* the weight) of the transitions of the transducer *`t.xml`* and writes the result in the weighted automaton *`c.xml`* on B^*.

***Precondition*:** no precondition.

***Comments*:** Mostly ancillary: needed for `evaluation`.

Comment for the Vaucanson *Group* (101205): *Pas implémentée.*

2.5.2.2 `composition`

```
$ vcsn composition t.xml u.xml > v.xml
$
```

Realizes the composition algorithm on *`t.xml`* and *`u.xml`* and writes the result in *`v.xml`*.

***Precondition*:** *`t.xml`* and *`u.xml`* are subnormalized, with matching monoids (output of *`t.xml`* = input of *`u.xml`*) and same weight semirings.

***Specification*:**

The composition algorithm used in TAF-Kit is described at Section C.5.2.2.

***Comments*:** When the weight semiring is not *complete*, it may be the case that the composition is not defined, in which case the call to `composition` will produce an error.

2.5.2.3 `evaluation`

```
$ vcsn evaluation t.xml a.xml > b.xml
$
```

Computes an automaton which realizes the image of the series realized by *a.xml* by the relation realized by *t.xml* and writes the result in *b.xml*.

***Precondition*:** *t.xml* is subnormalized, *a.xml* is a realtime automaton over the input monoid of *t.xml*, *t.xml* and *a.xml* have the same weight semiring.

***Specification*:**

`evaluation(t.xml, a.xml)` = `W-image(composition(partial-identity(a.xml),t.xml))`

***Comments*:** When the weight semiring is not *complete*, it may be the case that the evaluation is not defined, in which case the call to `evaluation` will produce an error.

2.5.2.4 `eval`

```
$ vcsn eval t.xml w > b.xml
$
```

Computes an automaton which realizes the image of the word *w* by the relation realized by *t.xml* and writes the result in *b.xml*.

***Precondition*:** *t.xml* is subnormalized, *w* is a word over the input monoid of *t.xml*.

***Specification*:**

`eval(t.xml, w)` = `evaluation(t.xml, standard(w))`

***Comments*:** Just a wrapper for `evaluation`.

Comment for the Vaucanson *Group* (101205): *Pas implémentée.*

2.5.3 Operations on behaviours of transducers

2.5.3.1 `composition-R`

```
$ vcsn composition-R t.xml u.xml > v.xml
$
```

Computes a transducer that realizes the composition of the relations realized by *t.xml* and *u.xml* and writes the result in *v.xml*.

***Precondition*:** *t.xml* and *u.xml* have matching monoids (output of *t.xml* = input of *u.xml*) and the same weight semiring.

***Specification*:**

`composition-R(t.xml, u.aml)` = `composition(subnormalize(t.xml), subnormalize(u.xml))`

Comment for the Vaucanson *Group* (101205): *Pas implémentée.*

2.5.3.2 `evaluation-S`

```
$ vcsn evaluation-S t.xml a.xml > b.xml
$
```

Computes an automaton which realises the series which is the image of the series realized by *`a.xml`* by the relation realized by *`t.xml`* and writes the result in *`b.xml`*.

Precondition: *`t.xml`* is any transducer, *`a.xml`* is any automaton over the input monoid of *`t.xml`*, *`t.xml`* and *`a.xml`* have the same weight semiring.

Specification:

`evaluation-S(t.xml, a.xml)` = `evaluation(subnormalize(t.xml),realtime(a.xml))`

Comment for the VAUCANSON *Group* (101205): *Pas implémentée.*

2.6 Weighted automata on free monoids over alphabets of pairs

An alphabet of pairs A is defined by a pair of alphabets B and C and letters in A are pairs (b,c) with b in B and c in C. A is thus a subset of $B{\times}C$, $(B{\times}C)^*$ is easily identified with a subset of $B^*{\times}C^*$ and in this way some functions apply to automata over A^* that correspond to functions on automata over $B^*{\times}C^*$.

The alphabets of pairs are the key to several constructions on automata and transducers. One example is when letters within an expression or an automaton are *indexed*; another one is the treatment of letter-to-letter transducers as automata on a free monoid. In TAF-Kit of Vaucanson 1.4 there are not many functions special to automata over such alphabets. There will be more in subsequent versions. At this stage, what is more important is the mere existence of this type of automata whithin TAF-Kit, which already allows to demonstrate the usefulness of going forth and back between the class of transducers and the one of automata.

1. Transformations of automata

   (1.1) `first-projection` <*aut*>, `second-projection` <*aut*>

   (1.2) `pair-to-fmp` <*aut*>

Comment for the Vaucanson *Group*: *Bilan de la réunion du 2/12/10:*

Aucune de ces commandes n'a encore été testée en dehors des tests associés à la génération–compilation de la tarball.

2.6.1 Transformations of automata

2.6.1.1 `first-projection`, `second-projection`

```
$ vcsn first-projection a.xml > b.xml
$
```

yields an automaton over B^* (resp. C^*), by keeping the first (resp. second) component of every letter.

2.6.1.2 `pair-to-fmp`

```
$ vcsn pair-to-fmp a.xml > t.xml
$
```

yields *fmp-transducer* over $B^*\times C^*$ every letter (b,c) to the corresponding element of $B^*\times C^*$.

***Specification*:**

A transition labelled by $(a,x)(b,x)(a,y)$ becomes a transition labelled by (aba,xxy).

Appendix A

Automata repository and factory

The VAUCANSON 1.4 distribution contains a folder `data/automata/` where a number of automata and of VAUCANSON programs which build automata are ready for use by the TAF-KIT commands.

A.1 $\mathbb{B}$-automata

A.1.1 Repository

The following automata files are stored '`data/automata/char-b/`' directory (and accessible by the command `vcsn-char-b cat`).

A.1.1.1 '`a1.xml`' for $\mathcal{A}_1$

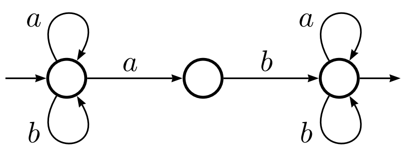


Figure A.1: The Boolean automaton $\mathcal{A}_1$ over $\{a, b\}^*$ (*cf.* Figure 1.1).

A.1.1.2 '`b1.xml`' for $\mathcal{B}_1$

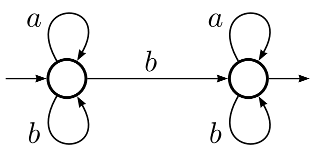


Figure A.2: The Boolean automaton $\mathcal{B}_1$ over $\{a, b\}^*$.

A.1.2 Factory

The following programs are in the '`data/automata/char-b/`' directory.

A.1.2.1 Program '`divkbaseb`'

```
$ divkbaseb 4 3 > div4base3.xml
$
```

Generates an automaton over the digit alphabet $\{0,\ldots,b-1\}$ that recognises the writing in base *b* of the numbers divisible by the integer *k*.

***Comments*:** The 'divisor' '`div3base2.xml`' (Figure A.3) is already in the repository.

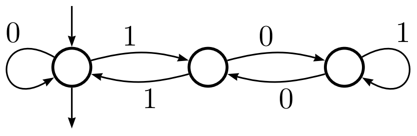


Figure A.3: The 'divisor' '`div3base2.xml`' over $\{0,1\}^*$.

A.1.2.2 Program '`double_ring`'

```
$ double_ring 6 1 3 4 5 > double-6-1-3-4-5.xml
$
```

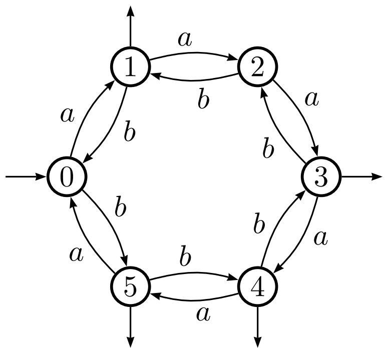
Generates an *n* state automaton over the alphabet $\{a,b\}$ that consists in a double ring of transitions: a counter clockwise ring of transitions labelled by *a* and a clockwise ring of transitions labelled by *b*.

***Specification*:**

The states are labelled from *0* to *n-1*. State *0* is initial. The number of states *n* is the first parameter and the next parameters give the list of final states. Figure A.5 shows the automaton built by the above command.

***Comments*:** The double-ring automata are closely related to the star-height problem. Schützenberger used them to give the first example of automata over a 2 letter alphabet that have arbitrary large loop complexity and McNaughton to give the simplest example of minimal automata which do not achieve the minimal loop complexity for the language the recognize. This was then reinterpreted in terms of *universal automata* (*cf.* [7, Sec. II.8]).

The automaton '`double-3-1.xml`' (Figure A.5) is already in the repository.



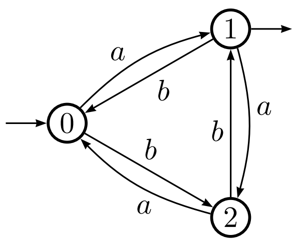


Figure A.4: The 'double rings' $\mathcal{H}_6$ and '`double-3-1.xml`'

A.1.2.3 Program '`ladybird`'

```
$ ladybird 6  > ladybird-6.xml
$
```

Generates an *n* state automaton over the alphabet $\{a, b, c\}$ whose equivalent minimal deterministic automaton has 2^n states.

***Specification*:**

The states are labelled from *0* to *n-1*. State *0* is initial and final. The number of states *n* is the first parameter and the next parameters give the list of final states. Figure A.5 shows the automaton built by the above command.

***Comments*:** The determinisation of '`ladybird-n`' has 2^n states and is minimal as it is co-deterministic.

'`ladybird-n`' is used in the benchmarking of Vaucanson.

The automaton '`ladybird-6.xml`' (Figure **??**) is already in the repository.

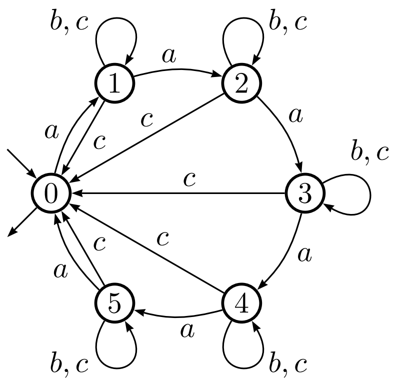


Figure A.5: The 'ladybird' $\mathcal{L}_6$

A.1.2.4 Function '`universal`'

To be completed

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

A.2.1 Repository

The following automata files are stored '`data/automata/char-z/`' directory (and accessible by the command `vcsn-char-z cat`).

A.2.1.1 '`b1.xml`'

The chacteristic automaton of the automaton $\mathcal{B}_1$ (*cf.* Figure A.2). The different outcomes of functions such as `power n b1.xml \quotient -` on the automaton '`b1.xml`' illustrate well the influence of the weights.

A.2.1.2 ‘c1.xml’ for $\mathcal{C}_1$

***Comments*:** The series realized by $\mathcal{C}_1$ associated with every word w of $\{a,b\}^*$ its value in the binary notation, when a is interpreted as 0 and b as 1.

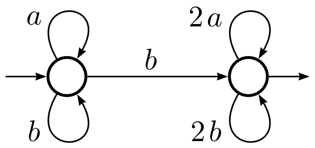


Figure A.6: The $\mathbb{Z}$-automaton $\mathcal{C}_1$ over $\{a,b\}^*$.

A.2.2 Factory

The following program is in the ‘data/automata/char-z/’ directory.

A.2.2.1 Program ‘rem_divkbaseb’

```
$ rem_divkbaseb 4 3 > rem-div4base3.xml
$
```

Generates an automaton over the digit alphabet $\{0,\ldots,b-1\}$ that computes the remainder of the division by the integer k of the numbers written in base b (*cf.* Figure A.7).

***Comments*:** The ‘divisor’ ‘rem-div3base10.xml’ is already in the repository.

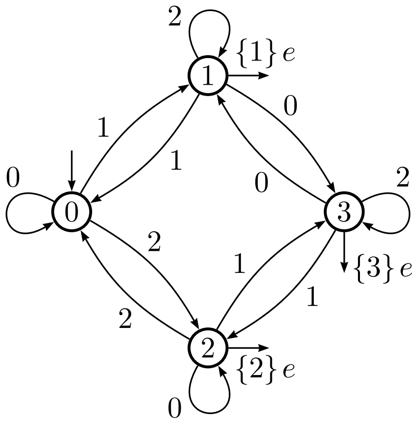


Figure A.7: The ‘divisor’ ‘rem-div4base3.xml’ over $\{0,1,2\}^*$.

A.3 $\mathbb{B}$-*fmp-transducers*

A.3.1 Repository

The following automata files are stored ‘data/automata/char-z/’ directory (and accessible by the command vcsn-char-z cat).

A.3.1.1 ‘`t1.xml`’ for $\mathcal{T}_1$

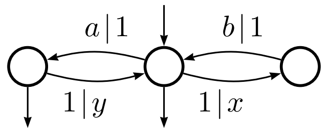


Figure A.8: The *fmp-transducer* $\mathcal{T}_1$ over $\{a,b\}^* \times \{x,y\}^*$ (*cf.* Section C.5.2.2).

A.3.1.2 ‘`u1.xml`’ for $\mathcal{U}_1$

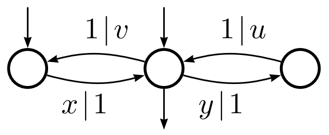


Figure A.9: The *fmp-transducer* $\mathcal{U}_1$ over $\{x,y\}^* \times \{u,v\}^*$ (*cf.* Section C.5.2.2).

A.3.2 Factory

The following program is in the ‘`data/automata/char-fmp-b/`’ directory.

A.3.2.1 Program ‘`quotkbaseb`’

```
$ quotkbaseb 3 2 > quot3base2.xml
$
```

Generates an *fmp-transducer* over the digit alphabets $\{0,\ldots,b-1\}$ that computes the *integer quotient* of the division by the integer *k* (first parameter) of the numbers written in base *b* (second parameter).

***Comments*:** The ‘divisor’ ‘`quot3base2.xml`’ (*cf.* Figure A.10) is already in the repository.

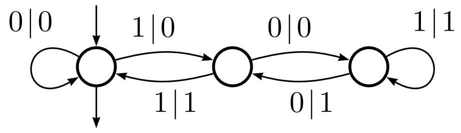


Figure A.10: The ‘divisor’ ‘`quot3base2.xml`’ over $\{0,1,2\}^*$.

Appendix B

FSM XML, an XML format for automata

Comment: *This appendix gives the description of the format* FSM XML *that had been given in May 2008 under a layout that is lighter than the reference card that can be found on the web page of format.*

This description is not up to date anymore and has to be updated.

FSM XML is an XML format proposal for the description of weighted automata, transducers, and regular expressions.

The aim of this XML format is to make possible, and hopefully easy, the communication between the various programs and systems that deal with weighted automata and transducters. FSM XML is part of the VAUCANSON Project. In particular, FSM XML is the input/output format of TAF-KIT, the command line interface to the VAUCANSON library.

This document gives a pseudo-formal description of the format.

All tags of the format are listed, with their children, and attributes.

1 The root tag

0. `<fsmxml/>`

   The unique possible root of an FSM XML file, which can contain any number of automata and standalone rational expressions.

```
<fsmxml xmlns="" version="">
   <automaton/>        0 or more occ.
   <regExp/>           0 or more occ.
</fsmxml>
```

2 The value type tags

Both `<automaton/>` and `<regExp/>` have a child in common: the `<valueType/>` tag which describes the 'type' of the behaviour of the automaton or of the series denoted by the expression.

1. `<valueType/>`

```
<valueType>
   <writingData/>      opt.
   <semiring/>         req.
   <monoid/>           req.
</valueType>
```

2. `<semiring/>`

   · Pivotal att.: `type` = *numerical*|*series*   token, req.

   2.1. `type` = *numerical*

```
<semiring type=numerical set='' operation='' >
   <writingData/>      opt.
</semiring>
```

   · Att: `set` = *B*|*N*|*Z*|*Q*|*R*|*C*   token, req.
   `operation` = *classical*| *minPlus*| *maxPlus*   token, req.

   · Tag `<writingData identitySymbol='' zeroSymbol=''/>`
   · Att.: `identitySymbol` = ' '   string, req.
   `zeroSymbol` = ' '   string, req.

   2.2. `type` = *series*

```
<semiring type=series>
   <writingData/>      opt.
   <semiring/>         req.
   <monoid/>           req.
</semiring>
```

   · Constraint: `<monoid/>` should not be of `type` = *unit*

3. `<monoid/>`

· Pivotal att.: `type` = *`unit`*|*`free`*|*`product`*   token, req.

3.1. `type` = *`unit`*

This type means 'no monoid' and gives the possibility of describing within the format graphs valued by numerical semirings only.

```
<monoid type=unit/>
```

· Constraint: not allowed in `<semiring type=series/>` (*cf.* 2.2)
nor in `<monoid type=product/>` (*cf.* 3.3)

3.2. `type` = *`free`*

· Pivotal att.: `genKind` = *`simple`*|*`tuple`*   token, req.
· Pivotal att.: `genDescrip` = *`enum`*|*`range`*|*`set`*   token, req.

This attribute `genDescrip` is put for further development. No alternative value is described here.

3.2.1. `genKind` = *`simple`*

```
<monoid type=free genKind=simple genDescrip='enum' genSort=''>
   <writingData/>              opt.
   <monGen/>             1+ occ. req.
</monoid>
```

· Att.: `genSort` = *`letter`*|*`digit`*|*`alphanum`*|*`integer`*   token, req.
· Tag `<writingData identitySymbol=''/>`
`identitySymbol` = ' '   string,
Tells how the identity of the monoid should be written when output (*e.g.* in expressions)

3.2.2. `genKind` = *`tuple`*

```
<monoid type=free genKind=tuple genDim='' genDescrip='enum'>
   <writingData/>                        opt.
   <genSort>                             req.
      <genCompSort/>     "genDim"   occ. req.
   </genSort>
   <monGen/>                      1+   occ. req.
</monoid>
```

· Att.: `genDim` = ' '   integer strictly larger than 1, req.
· Tag `<genSort/>` holds the *`"genDim"`* `<genCompSort/>` tags
· Tag `<genCompSort value=''/>`
`value` = ' '   has the same role as the attribute `genSort` in 3.2.1 for the corresponding coordinate of the monoid generator and can take the same token values.

3.3. `type` = *`product`*

```
<monoid type=product prodDim=''>
   <writingData/>                      opt.
   <monoid/>          "prodDim"  occ. req.
</monoid>
```

· Att.: `prodDim` = ' '   integer strictly larger than 1, req.

– Constraint: no children `<monoid/>` can be of `type` = *`unit`*

4. `<monGen/>`

Describes a monoid generator for a free monoid.
Its form will depend on the pivotal attribute `genKind`.
(The only case considered here is when `genDescrip` = *enum*.)

4.1. `genKind` = *`simple`*

```
<monGen value=''/>
```

· Att.: `value`   must be consistent with `genSort` req.

4.2. `genKind` = *`tuple`*

```
<monGen>
   <monCompGen/>       "genDim" occ. req.
</monGen>
```

· Tag `<monCompGen value=''/>`

– Constraint: each `value` must be consistent with the corresponding `genCompSort`

3 The rational (regular) expressions

5. `<regExp/>`

   ```
   <regExp name="">
      <valueType/>        req.
      <typedRegExp/>      req.
   </regExp>
   ```

   · Att.: `name=` ’ ’   string, opt.

   – Constraint: the `<monoid/>` cannot be of `type` = *`unit`*

   – At this stage, one could think of a `<writingData/>` tag which would contains writing options for the expressions: a dot or nothing for the product, delimitors for the weight, etc.

6. `<typedRegExp/>`

   ```
   <typedRegExp>
      {Body::typedRegExp}     plays the role of a non terminal in a grammar.
   </typedRegExp>
   ```

   ```
   {Body::typedRegExp}= <sum/>|<product/>|<star/>|
                        <rightExtMul/>|<leftExtMul/>|
                        <zero/>|<one/>|<monElmt/>
   ```

7. `<sum/>`

   ```
   <sum>
      {Body::typedRegExp}
      {Body::typedRegExp}
   </sum>
   ```

8. `<product/>`

   ```
   <product>
      {Body::typedRegExp}
      {Body::typedRegExp}
   </product>
   ```

9. `<star/>`

   ```
   <star>
      {Body::typedRegExp}
   </star>
   ```

10. `<rightExtMul/>` or `<leftExtMul/>`

```
<xxxExtMul>
   <weight/>
   {Body::typedRegExp}
</xxxExtMul>
```

11. `<zero/>` and `<one/>` "final" tags

12. `<monElmt/>`

    Depends on the pivotal attribute `type` of the `<monoid/>` in the `<valueType/>`.

    12.1. `type` = *`free`*

    ```
    <monElmt>
       <monGen/>        1+ occ. req.
    </monElmt>
    ```

    12.2. `type` = *`product`*

    ```
    <monElmt>
       <one/>|<monElmt/>        "prodDim" occ. req.
    </monElmt>
    ```

13. `<weight/>`

    Depends on the pivotal attribute `type` of the `<semiring/>` in the `<valueType/>`.

    13.1. `type` = *`numerical`*

    ```
    <weight value=''/>
    ```

    · Att.: `value=` ' '   string, that will be interpreted according to the attribute `set`

    – If `set` =*`Q`* , one can think of having 2 integers values.

    13.2. `type` = *`series`*

    ```
    <weight>
       {Body::typedRegExp}        of the <valueType/> defined by <semiring/>
    </weight>
    ```

4 The automata

14. `<automaton/>`

```
<automaton name=’’ readingDir=’’>
   <geometricData/>        opt.
   <drawingData/>          opt.
   <valueType/>            req.
   <automatonStruct/>      req.
</automaton>
```

· Att.: `name` = ’ ’ string, opt.
`readingDir` = *left*|*right* token,
· Tag `<geometricData x=’’ y=’’/>` gives relative origin
· Tag `<drawingData drawingClass=’’ />` or something more complicated

15. `<automStruct/>`

```
<automStruct>
   <states/>          req.
   <transitions/>     req.
</automStruct>
```

16. `<states/>`

```
<states>
   <state/>       0 or more occ.
</states>
```

17. `<state/>`

```
<state id=’’ name=’’ key=’’ >
   <geometricData/>      opt.
   <drawingData/>        opt.
</state>
```

· Att.: `id` = ’ ’ string, req. must be unique in the whole automaton.
`name` = ’ ’ string, opt.
`key` = ’ ’ integer opt. may be used to pass an ordering on the states.
· Tag `<geometricData x=’’ y=’’/>` coordinates of the state
· Tag `<drawingData drawingClass=’’ />` or something more complicated

18. `<transitions/>`

```
<transitions>
   <transition/>        0 or more occ.
   <initial/>           0 or more occ.
   <final/>             0 or more occ.
</transitions>
```

19. `<transition/>`

```
<transition source='' target=''>
   <geometricData/>       opt.
   <drawingData/>         opt.
   <label/>               req.
</transition>
```

· Att.:`source` = ' '   string, req. must be a valid `id`
`target` = ' '   string, req. must be a valid `id`

· Tag `<geometricData transitionType='' labelPos='' labelDist='' loopDir=''/>`

· Pivotal att.: `transitionType` = *EdgeL*|*EdgeR*|*ArcL*|*ArcR*   token, req.
`source` must be different from `target`
`transitionType` = *Loop*   `source` must be equal to `target`

· Att.: `loopDir` = *N*|*S*|*E*|*W*|*NE*|*NW*|*SE*|*SW* token, req.
or integer between 0 and 360 (*E* 0)
but only if `transitionType` = *Loop*
`labelPos` = ' ' float opt.
`labelDist` = ' ' float opt.

· Tag `<drawingData drawingClass='' />` or something more complicated

20. `<initial/>`

```
<initial state=''>
   <geometricData/>       opt.
   <drawingData/>         opt.
   <label/>               req.
</initial>
```

· Att.: `state` = ' '   string, req. must be a valid `id`

· Tag `<geometricData initialDir='' labelPos='' labelDist=''/>`
(*cf.* 19)

· Tag `<drawingData drawingClass='' />` or something more complicated

It is assumed that initial states are marked with an incoming arrow.

21. `<final/>`

```
<final state=''>
   <geometricData/>       opt.
   <drawingData/>         opt.
   <label/>               req.
</final>
```

· Att.: `state` = ' '   string, req. must be a valid `id`

· Tag `<geometricData finalMod='' finalDir='' labelPos='' labelDist=''/>`
   · Pivotal att.: `finalMod` = *circle*|*arrow*   token, req.
   · Other att: *cf.* 19.

· Tag `<drawingData drawingClass='' />` or something more complicated

22. `<label/>`

```
<label>
   {Body::typedRegExp}
</label>
```

Appendix C

Algorithm specification, description and discussion

C.1 General automata and rational expressions functions

C.1.1 Graph functions

C.1.1.0 `reverse`

This is a hidden (and ancillary) graph function, not accessible to the user through TAF-Kit (because it would be somewhat confusing with `transpose`. It builds the transpose of the graph including the initial and final function that can be seen as labels of arcs from subliminal to real states, but leaves the labels untouched.

C.1.1.1 `accessible`, `coaccessible`, `trim`

Graph traversal. Implemented by depth-first, or breadth-first search?

Should be in-place by default in the library.

C.1.2 Transformations of automata

C.1.2.1 `proper`

From a theoretical point of view, the algorithm `proper` cannot be described, nor understood, before addressing the problem of the star in a semiring of series.

(1) If M is graded, then $\mathbb{K}\langle\langle M\rangle\rangle$, equipped with the Cauchy product, is a semiring as well.[1]

If $\mathbb{T}$ is a semiring, and t is in $\mathbb{T}$, by definition

$$t^* = \sum_{n\in\mathbb{N}} t^n$$

and as infinite sums are not always defined, t^* is not always defined. Hence a semiring should be equipped with two supplementary methods (supplementary to the defining operations of the semiring) `is-starable()` and `star()`, with obvious meaning and result.

[1]If M is not graded, this may not be the case anymore, but is out of the scope of Vaucanson for the time being, and for certain while.

If s is a series in $\mathbb{K}\langle\langle M\rangle\rangle$, we denote by $\mathsf{c}(s)$ its *constant term*, that is, the coefficient of 1_M. Thus, a series s is proper if its constant term is nul: $\mathsf{c}(s) = 0_{\mathbb{K}}$. And the *proper part* of an arbitrary series s is the proper series s_{p} such that $s = \mathsf{c}(s)\,1_M + s_{\mathsf{p}}$. Under a natural, and not restrictive, hypothesis on $\mathbb{K}$ (*cf.* [7, 8]), the following property holds.

Property 1 *A series s in $\mathbb{K}\langle\langle M\rangle\rangle$ is starable if, and only if, $\mathsf{c}(s)$ is starable and it holds: $s^* = (\mathsf{c}(s))^*\,(s_{\mathsf{p}}\,(\mathsf{c}(s))^*)^*$*

As a conclusion to this paragraph, we can say that star is not always defined in $\mathbb{K}$, and thus in $\mathbb{K}\langle\langle M\rangle\rangle$.

(2) Let $\mathcal{A}$ be an automaton over M with multiplicityin $\mathbb{K}$. We say that the *behaviour of* $\mathcal{A}$, $|\mathcal{A}|$, is defined if, and only if, for every pair of states p and q in $\mathcal{A}$, the family of labels of computations from p to q is *summable.*

Let $\mathcal{A}_0$ be the automaton obtained from $\mathcal{A}$ by retaining the transitions labelled by 1_M only. We then have:

Property 2 *The behaviour of $\mathcal{A}$ is defined if, and only if, the behaviour of $\mathcal{A}_0$ is.*

(3) Let $\mathcal{A} = \langle\, I, E, T\,\rangle$ and $\mathcal{A}_0 = \langle\, I, E_0, T\,\rangle$ be their respective matrix description. We write E_{p} for the proper matrix such that $E = E_{\mathsf{p}} + E_0$.

Property 3 *If the behaviour of $\mathcal{A}$ is defined, we have:*

$$|\mathcal{A}| = I \cdot {E_0}^* \cdot (E_{\mathsf{p}} \cdot {E_0}^*)^* \cdot T \quad .$$

It is important to note that it is not true that $|\mathcal{A}|$ is necessarily defined when ${E_0}^*$, and thus $I \cdot {E_0}^* \cdot (E_{\mathsf{p}} \cdot {E_0}^*)^* \cdot T$ are defined (*cf.* [7, 8] for more details and example).

TO BE COMPLETED

C.1.2.2 `standardize`

An automaton is said to be *standard* if it has a *unique initial state* which is the destination of no transition and whose *initial multiplicity* is equal to the *unit* (of the multiplicity semiring).

Not only every automaton is equivalent to a standard one, but a simple procedure, called 'standardization', transforms every automaton $\mathcal{A}$ in an equivalent standard one. The difficulty in specifying standardization comes from the fact that on the one hand side a standard automaton is not necessarily proper nor accessible and on the other the initial function of a state may a priori be any polynomial.

The procedure goes as follows.

(i) Add a new state s , make it initial, with initial multiplicity equal to the unit of the multiplicity semiring.

(ii) For every initial state i of $\mathcal{A}$, with initial function $I(i)$, add a transition from s to i with label $I(i)$, and set $I(i)$ to $0_{\mathbb{K}}$ (the zero of the multiplicity semiring) — which is equivalent to say that i is *not initial anymore.*

(iii) Suppress by a *backward closure* every *spontaneous transition* that has been created in (ii).

By *convention*, we consider that a transition from s to i is spontaneous if $I(i)$ is *scalar*, that is, if the support of $I(i)$, seen as a polynomial over A^*, is restricted to the identity 1_{A^*}.

(iv) Remove the former initial states of $\mathcal{A}$ that are the destination of no incoming transition.

***Comments*:** (a) Steps (iii) and (iv) are necessary to insure the following property:

The standardization of a standard automaton $\mathcal{A}$ is isomorphic to $\mathcal{A}$.

(b) We say 'by convention' in (iii) as we could have chosen different policies without loosing the above property (which is in the specification of `standardize`).

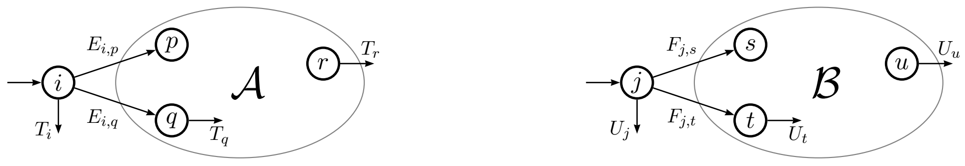
– A non-proper polynomial $I(i)$ could give rise to a spontaneous transition labelled with its constant term. We prefered not to do it in order to change as few things as possible.

– We could have decided to perform no closure as soon as there exists at least one initial function which is not scalar. We have prefered to have a choice which is more local to every intial state, but this is certainly disputable.

C.1.3 Operations on automata

A small sketch is worth a long speech.

Let $\mathcal{A} = <Q, A, E, \{i\}, T>$ and $\mathcal{B} = <R, A, F, \{j\}, U>$ be two standard automata:



C.1.3.1 `union`

Just the union of the two automata. It is a graph function indeed.

C.1.3.2 `sum`

***Precondition*:** *`a.xml`* and *`b.xml`* are *standard* for the sum operation is defined only on standard automata.

***Specification*:**

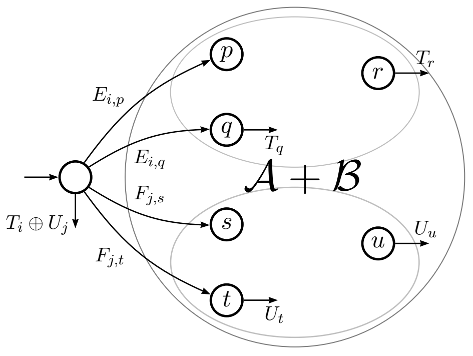
• The standard automaton $\mathcal{A} + \mathcal{B} = <Q \cup R \setminus \{j\}, A, G, \{i\}, V>$ is defined as:

$\forall p, q \in Q \cup R \setminus \{j\}\,,$

$$G_{p,q} = \begin{cases} E_{p,q} & if \quad p, q \in Q \\ F_{p,q} & if \quad p, q \in R \\ F_{j,q} & if \quad p = i \text{ and } q \in R \\ 0_{\mathbb{K}} & otherwise \end{cases}$$

$\forall p \in Q \cup R \setminus \{j\}\,,$

$$V_p = \begin{cases} T_i \oplus U_j & if \quad p = i \\ T_p & if \quad p \in Q \setminus \{i\} \\ U_p & if \quad p \in R \end{cases}$$



C.1.3.3 `concatenate`

***Precondition*:** `a.xml` and `b.xml` are *standard* for the concatenation operation is defined only on standard automata.

***Specification*:**

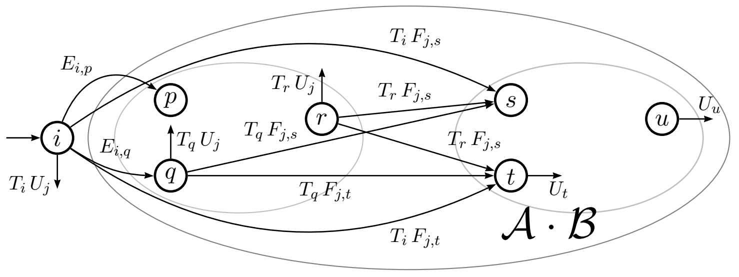
• The standard automaton $\mathcal{A} \cdot \mathcal{B} = \langle Q \cup R \setminus \{j\}, A, G, \{i\}, V \rangle$ is:

$\forall p, q \in Q \cup R \setminus \{j\}\,,$

$$G_{p,q} = \begin{cases} E_{p,q} & if \quad p, q \in Q \\ F_{p,q} & if \quad p, q \in R \\ T_p F_{j,q} & if \quad p \in Q \text{ and } q \in R \\ 0_{\mathbb{K}} & otherwise \end{cases}$$

$\forall p \in Q \cup R \setminus \{j\}\,,$

$$V_p = \begin{cases} U_p & if \quad p \in R \\ T_p U_j & if \quad p \in Q \end{cases}$$



C.1.3.4 `star`

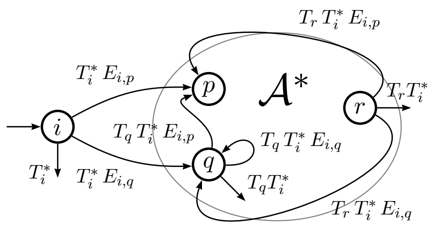
***Precondition*:** `a.xml` is *standard* for the star operation is defined only on standard automata.

***Specification*:**

• The standard automaton $\mathcal{A}^* = {<}Q, A, E^{(*)}, \{i\}, T^{(*)}{>}$ is:

$$\forall p, q \in Q, \quad E^{(*)}_{p,q} = \begin{cases} T_i^* E_{i,q} & if \quad p = i \\ T_p T_i^* E_{i,q} \oplus E_{p,q} & otherwise \end{cases}$$

$$\forall p \in Q, \quad T^{(*)}_p = \begin{cases} T_i^* & if \quad p = i \\ T_p T_i^* & otherwise \end{cases}$$



C.1.3.5 `left-mult`

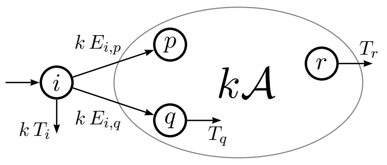
Precondition: `a.xml` is *standard* for the left 'exterior' multiplication operation is defined only on standard automata.

Specification:

• The standard automata $k\mathcal{A} = {<}Q, A, E^{(k.)}, \{i\}, T^{(k.)}{>}$ is defined by:

$$\forall p, q \in Q\,, \qquad E^{(k.)}_{p,q} = \begin{cases} k\, E_{p,q} & if \quad p = i \\ E_{p,q} & otherwise \end{cases}$$

$$\forall p \in Q\,, \qquad T^{(k.)}_p = \begin{cases} k\, T_p & if \quad p = i \\ T_p & otherwise \end{cases}$$



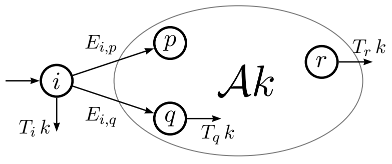
C.1.3.6 `right-mult`

Precondition: `a.xml` is *standard* for the right 'exterior' multiplication operation is defined only on standard automata.

Specification:

• The standard automata $\mathcal{A}k = {<}Q, A, E, \{i\}, T^{(.k)}{>}$ is defined by:

$$\forall p \in Q\,, \qquad T^{(.k)}_p = T_p\, k$$



C.1.4 From automata to expressions

Vaucanson implements the *state elimination method* for computing the rational expression that denotes the behaviour of an automaton. The outcome of the algorithme depends upon the order in which the states are 'eliminated'.

In VAUCANSON library, this order of states is given to the fonction as a second parameter called '`chooser`'. The `chooser` either runs over a list of states that is given explicitely, or implements a heuristics that computes at each step the next state to be suppressed. Two heuristics are implemented in the library: the *'naïve heuristics* which is described below, and a variant of it which takes into account not only the number of transitions incident to every given state, but also the length of the expressions that label these transitions it is due to Delgado and Morais and described in [3].

Note that in any case and for a precise specification (in view of the derivation procedure in particular), one should specify the bracketting:

$$p \xrightarrow{\mathsf{F}} q \xrightarrow{\mathsf{G}} q \xrightarrow{\mathsf{H}} r \quad \text{gives} \quad p \xrightarrow{\mathsf{F}\cdot(\mathsf{G}^*\cdot\mathsf{H})} r \tag{C.1}$$

after the elimination of the state q.

C.1.4.1 The 'naïve' heuristics

(a) Make real the initial and final subliminal states i and t. From i to every initial state p, there is thus a transition with label $I(p)$. Dually, from every final state r to t, there is thus a transition with label $T(r)$.

(b) For every state p (outside i and t) compute a two component index $(l(p), k(p))$:
- $l(p) = 1$ if p is the origin of a loop, $l(p) = 0$ otherwise;
- $k(p) = [i(p) - 1][o(p) - 1]$ where $i(p)$ is the in-degree of p and $o(p)$ its out-degree.
- Lexicographically order the states by their index.

(c) While there remains states,
- choose the state q with smallest index,
- remove it and replace the incoming and outgoing transitions according to (C.1),
- recompute the index for those states that were adjacent to q.

(d) Return the label of the transition from i to t.

Comment for the VAUCANSON *Group* (101206):
(i) *It is to be carefully checked whether the `NHChooser` and the `DMChooser` correctly implement the naïve and the Delgado–Morais heuristics respectively.*
(ii) *I do not see why we put* $k(p) = [i(p) - 1][o(p) - 1]$ *rather than* $k(p) = [i(p)][o(p)]$ *in the description of the 'naïve heuristics.*

C.2 Weighted automata and rational expressions over free monoids

C.2.1 Transformations of automata

C.2.1.1 `quotient`

As announced, a variant of Hopcroft's algorithm.

Comment for the VAUCANSON *Group* (101205): *Mérite d'être décrit, au moins dans les grandes lignes.*

C.2.2 Behaviour of automata

C.2.2.1 `eval`

Comment for the VAUCANSON *Group* (101205): *Mérite d'être décrit, au moins dans les grandes lignes.*

C.3 Automata and rational expressions on free monoids with weights in a field

C.3.1 Operations on automata

C.3.1.1 Reduction of representations over a field

Automata and representation

Any finite automaton over A^* with multiplicity in $\mathbb{K}$ is equivalent to a *realtime* automaton $\mathcal{A}$ with set of states Q: $\mathcal{A} = \langle I, E, T \rangle$ where I and T are vectors in $\mathbb{K}^Q$ and E is a square matrix of dimension Q, whose entries are *linear combination* with coefficients in $\mathbb{K}$ *of letters in* A. One can then write:

$$E = \sum_{a \in A} a\mu a$$

where every $a\mu$ is a square matrix of dimension Q with entries in $\mathbb{K}$. These matices define a morphism μ from A^* into $\mathbb{K}^{Q \times Q}$, and for every w in A^* the coefficient of w in the series s realised by $\mathcal{A}$ is $<s, w> = I \cdot w\mu \cdot T$. The tuple (I, μ, T) is called a *representation* (of dimension Q) of s.

Rational series over a field

If $\mathbb{K}$ is a *field* $\mathbb{F}$, for every $\mathbb{F}$-rational series s, there exists an integer r, called the *rank* of s which is the minimal dimension of any representation of s. A representation of minimal dimension is said to be *reduced*.

Theorem 1 (Schützenberger) *A reduced representation of a* $\mathbb{F}$*-rational series* s *is effectively computable from any representation of* s*.*

A reduced representation of a rational series is an object comparable to the minimal automaton of a rational language, to the extent it is not unique but defined up to a basis transformation within $\mathbb{F}^Q$. The theorem implies two $\mathbb{F}$-automata which realize s and t are equivalent if, and only if the reduced representation of the series $s - t$ is of dimension 0 and this is decidable.

The algorithm

A representation (I, μ, T) of dimension Q being given, the algorithm that underlies the theorem amounts to find a maximal prefix-closed subset P of A^* such that the vectors $I \cdot p\mu$ are independent (in $\mathbb{F}^Q$). Such set of vectors allows in turn to compute a new and equivalent representation, of dimension P. The substance of the theorem is that it is sufficient to perform this algorithm twice in a row, on the given representation and then on its transpose in order to get the reduced representation.

The elementary step in this algorithm is thus to determine whether a given vector belongs to a subspace generated by a set of given independent vectors and in the positive case to compute its coordinates in this system, that is to solve a system of linear equations. In order to reach the optimal complexity, and also to be able to treat the case of non-commutative fields (a case which does not appear in VAUCANSON 1.4), these systems are solved by an iterative method of Gaussian elimination.

C.4 Boolean automata and rational expressions on free monoids

C.4.1 Operations on automata

C.4.1.1 `minimize`

As announced, VAUCANSON implements Hopcroft's algorithm.

Comment for the VAUCANSON *Group* (101205): *Mérite d'être décrit, au moins dans les grandes lignes.*

C.4.3 Operations on expressions

C.4.3.1 `derived-term`

The derived term automaton is constructed (in the Boolean case) by the following algorithm (rédigé par PYA):

(i) L'expression est transformée en `prat_exp` (`include/vaucanson/algorithms/internal/partial_rat_exp.hxx`). Il s'agit d'une liste de noeuds représentant donc une concaténation de sous-expressions. En fait:

- $\texttt{prat}(\mathsf{E}+\mathsf{F}) = \mathsf{E}+\mathsf{F}$
- $\texttt{prat}(\mathsf{E}*) = \mathsf{E}*$
- $\texttt{prat}(\mathsf{E}.\mathsf{F}) = \texttt{prat}(\mathsf{E}), \mathsf{F}$

Les états de l'automate vont être des `prat_exp`, la transformation de l'expression initiale donne l'état initial, l'automate est construit de manière incrémentale.

(ii) A chaque état:

- on vérifie le terme constant de la `prat_exp` et on rend l'état final en fonction
- pour chaque lettre de l'alphabet on dérive la `prat_exp` (`include/vaucanson/algorithms/internal/partial_rat_exp_derivation.hxx`) et on ajoute une transition en fonction.

Les états sont identifiés grace à l'opérateur `==` défini sur les `prat_exp` dans `include/vaucanson/algorithms/internal/partial_rat_exp.hxx` .

C.5.2 Operations on transducers

C.5.2.2 `composition`

Product of normalized transducers

We first consider two proper *normalized* transducers:

$$\mathcal{T} = \langle\, Q, A^* \times B^*, E, I, T \,\rangle \qquad \text{and} \qquad \mathcal{U} = \langle\, R, B^* \times C^*, F, J, U \,\rangle \ ,$$

that is, the transitions of $\mathcal{T}$ are labelled in $A \times 1$ or in $1 \times B$ and those of $\mathcal{U}$ are labelled in $B \times 1$ or in $1 \times C$.

The proof of the Composition Theorem is equivalent to the construction of the transducer

$$\mathcal{T} \bowtie \mathcal{U} = \langle\, Q \times R, A^* \times C^*, G, I \times J, T \times U \,\rangle$$

by the following rules.
(i) If $\big(p, (a,1), q\big) \in E$ then for all $r \in R$ $\big((p,r), (a,1), (q,r)\big) \in G$.
(ii) If $\big(r, (1,u), s\big) \in F$ then for all $q \in Q$ $\big((q,r), (1,u), (q,s)\big) \in G$.
(iii) If $\big(p, (1,x), q\big) \in E$ *and* $(r, (x,1), s) \in F$ then $\big((p,r), (1,1), (q,s)\big) \in G$.

A next possible step is to eliminate the transitions with label $(1,1)$ by means of a classical closure algorithm. Figure C.1 shows an example of such product, before and after the elimination of spontaneous transitions.

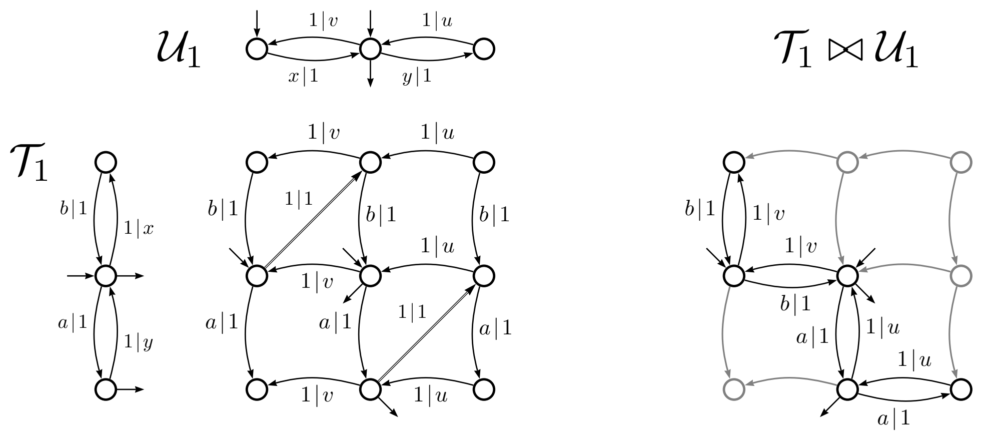


Figure C.1: Composition Theorem on Boolean transducers

Product of subnormalized transducers

This construction can easily be extended to *subnormalized* transducers, which are such that transitions are labelled in $\widehat{A} \times \widehat{B} \setminus (1,1)$ where $\widehat{A} = A \cup \{1\}$. It amounts to replace (iii) by
(iii') If $\big(p, (a',x), q\big) \in E$ with $a' \in \widehat{A}$ *and* $\big(r, (x,u'), s\big) \in F$ with $u' \in \widehat{C}$
then $\big((p,r), (a',u'), (q,s)\big) \in G$.

In this form, it contains as a particular case the composition of letter-to-letter transducers.

Product of subnormalized transducers and composition

It is known that this construction, which works perfectly well for Boolean transducers, does not yields a transducer for the composition if the multiplicities are to be taken into account.

For instance, there is one path labeled (aa, y) in $\mathcal{T}_1$ and one path labeled (y, u) in $\mathcal{U}_1$; and there are *two* paths labeled (aa, u) in $\mathcal{T}_1 \bowtie \mathcal{U}_1$. Hence, $\mathcal{T} \bowtie \mathcal{U}$ does not realize the composition of the weighted relations realized by $\mathcal{T}$ and $\mathcal{U}$.

Preparation of transducers for the composition

In order to have a product of transducers that realises the weighted composition, we performa preliminary operation on both operands that distinguishes between transitions and the take advantage of this supplementary information in order to supress some transitions in the product.

The construction on $\mathcal{T}$ and $\mathcal{U}$ can be described as follows:

(a) Split the states of $\mathcal{T}$ and their *outgoing* transitions in such a way they are labeled either in $(A \times 1)$ — black states — or in $\widehat{A} \times B$ (or the state is final) — white states; the incoming transitions are duplicated on split states. This is transducer $\mathcal{T}'$.

(b) Split the states of $\mathcal{U}$ and their *incoming* transitions in such a way they are labeled either in $(1 \times C)$ — black states — or in $B \times \widehat{C}$ (or the state is initial) — white states; the outgoing transitions are duplicated on split states. This is transducer $\mathcal{U}'$.

(c) Apply the preceeding algorithm [steps (i), (ii) and (iii')] to $\mathcal{T}'$ and $\mathcal{U}'$ in order to build $\mathcal{T}' \bowtie \mathcal{U}'$.

(d) Delete the black-black states (every state in $\mathcal{T}' \bowtie \mathcal{U}'$ is a pair of states).

(e) Trim and eliminate the transitions with label $(1, 1)$ by classical closure.

Figure C.2 shows the construction applied to $\mathcal{T}_1$ and $\mathcal{U}_1$.

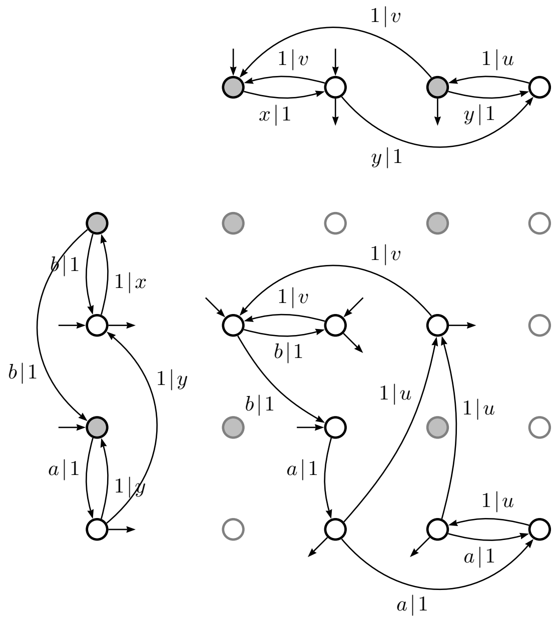


Figure C.2: A composition that preserves multiplicity

Bibliography

[1] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoret. Comput. Sci.* **155** (1996), 291–319.

[2] J.-M. Champarnaud, D. Ziadi. Canonical derivatives, partial derivatives and finite automaton constructions. *Theoret. Comput. Sci.* **289** (2002), 137–163.

[3] M. Delgado, J. Morais. Approximation to the smallest regular expression for a given regular language. *Proc. CIAA 2004*, LNCS 3317, Springer (2004) 312–314.

[4] Ch. Frougny, J. Sakarovitch. Number representation and finite automata, in *Combinatorics, Automata, and Number Theory* (V. Berthé, M. Rigo , eds) Cambridge University Press (2010) 34–107.

[5] S. Lombardy, J. Sakarovitch. Derivatives of rational expressions with multiplicity, *Theoret. Comput. Sci.* **332** (2005), 141–177.

[6] S. Lombardy, J. Sakarovitch. A general spontaneous-transition elimination algorithm and its implementation, *in preparation*.

[7] J. Sakarovitch *Éléments de théorie des automates.* Vuibert, 2003. English corrected edition: *Elements of Automata Theory*, Cambridge University Press, 2009.

[8] J. Sakarovitch Rational and recognisable power series, In *Handbook of Weighted Automata* (M. Droste, W. Kuich and H. Vogler, eds.) Springer (2009) 105–174.

Index

--a, 38

accessible, 43
accessible part, 58
--alphabet, 29, 38
are-equivalent, 61, 67
are-equivalent-E, 61
aut-to-exp, 11, 51

-B, 26
--bench, 26

cat, 37
cat-E, 28, 37, 70
chain, 48
coaccessible, 43
complement, 65
complete, 64
is-complete, 63
composition, 73
composition-R, 74
CONCAT, 35
concatenate, 46

-D, 26
data, 37
derived-term, 67
is-deterministic, 64
determinize, 65
display, 37
divkbaseb-char-b, 17
domain, 73
dot, 23
dotty, 23

edit, 18, 38
enumerate, 66
epsilon transition, 44
eval, 11, 57, 74
eval-S, 58
evaluation, 74
evaluation-S, 75
exp-to-aut, 18, 21, 52, 70
expand, 33, 52
expression
    reduced, 28, 37

factor, 66
field, 60
first-projection, 76
fst, 23

Graphviz, 37

--help, 25

-i, 37
image, 73
W-image, 73
infiltration, 59
intersection, 67
inverse, 71
is-empty, 25, 43
is-ltl, 72
is-unambiguous, 55
is-useless, 44

left-mult, 47
letter-to-letter transducer, 72
letterize, 54
--list-automata, 16
ltl-to-pair, 72

minimize, 65

-o, 37

pair-to-fmp, 76
--parser, 33
partial-identity, 56
predefined alphabets, 20

prefix, 65
product, 58, 59
product-S, 50
proper, 44
is-proper, 44
proper, 45

quotient, 20, 56

radix ordering, 66
rational
  expression, 27
  operator, 27
is-realtime, 55
realtime, 55
reduce, 60
reduced expression, *see* expression
--report-time, 25
right-mult, 47

second-projection, 76
shortest, 67
shuffle, 59
SPACE, 35
spontaneous transition, 44
is-standard, 45
standard, 58
standardize, 45
star, 47
star-S, 50
state elimination method, 50, 95
subnormalize, 72
is-subnormalized, 71
suffix, 65
sum, 46
sum-S, 50

-T, 25, 26
terminal state, 65
TIMES, 34
transducer, 70
transpose, 55
is-trim, 43
trim, 43
trivial identities, 28

union, 46

--verbose, 25

VGI, 37

writing data, 32

-X, 26