Vaucanson I/O

Date: January 2005

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

Contents

Introduction

As usual, the structure of the data representing an automaton in a flat file is called the file format. There are several input and output formats for Vaucanson automata. Obviously:

- input formats are those that can be read from, i.e. from which an automaton can be loaded.
- output formats are those that can be written to, i.e. to which an automaton can be dumped.

Given these definitions, here is the meat:

- Vaucanson supports Graphviz (dot) as an output format. Most kinds of automata can be dumped as dot-files. Through the library this format is simply called `dot`.
- Vaucanson supports XML as an input and output format. Most kinds of automata can be read and written to and from XML streams, which Vaucanson does by using the Xerces-C++ library. Through the library this format is simply called `xml`.

- Vaucanson supports the FSM toolkit I/O format as an input and output format. This allows for basic FSM interaction. Only certain kinds of weighted automata can be meaningfully input and output with this format. Through the library this format is simply called `fsm`.
- Vaucanson supports a simple informative textual format as an input and output format. Most kinds of automata can be read and written to and from this format. Through the library this format is simply called `simple`.

Dot format

This format provides an easy way to produce a graphical representation of an automaton.

Output using this format can be given as input to the Graphviz `dot` command, which can in turn produce graphical representations in Encapsulated PostScript, PNG, JPEG, and many others.

It uses Graphviz' "directed graph" subformat.

If you want to see what it looks like go to the `src/demos/automaton_library` subdirectory, build the examples and run them with the "dot" argument.

For Graphviz users:

Each graph generated by Vaucanson can be named with a string that also prefixes each state name. If done so, several automata can be grouped in a single graph by simply concatenating the Vaucanson outputs.

XML format

This format is intended to be an all-purpose strongly typed input and output format for automata.

Using it requires:

- that the Xerces-C++ library is installed and ready to use by the C++ compiler that is used to compile Vaucanson.
- configuring Vaucanson to use XML.
- computer resources and time.

What you gain:

- support for the Greater and Better I/O format. See documentation in the `doc/xml` subdirectory for further information.

If you want to see what it looks like go to the `src/demos/automaton_library` subdirectory, build the examples and run them with the `xml` argument.

FSM format

This format is intended to provide a basic level of compatibility with the FSM tool kit. (FIXME: references needed)

Like FSM, support for this format in Vaucanson is limited to deterministic automata. It probably does not work with transducers, either.

It is not meant to be used that much apart from performance comparison with FSM. Some code exists to simulate FSM, in `src/demos/utilities/fsm`.

If you want to see what it looks like go to the `src/demos/automaton_library subdirectory`, build the examples and run them with the `fsm` argument.

Simple format

Initially intended to be a quick and dirty debugging input and output format, this format actually proves to be a useful, compact and efficient textual representation of automata.

Advantages over XML:

- does not require additional 3rd party software,
- simple and efficient (designed to be read and written to streams with very low memory footprint and minimum complexity),
- less bytes in file,
- not strongely typed (can be dumped from one automaton type and loaded to another).

Drawbacks from XML:

- not strongely typed (one cannot know what automaton type to build by only looking at the raw data).
- currently does not (probably) support transducers.

If you want to see what it looks like go to the `src/demos/automaton_library subdirectory`, build the examples and run them with the `simple` argument.

Using input and output

The library provides an infrastructure for generic I/O, which (hopefully) will help supporting more formats in the future.

The basis for this infrastructure is the way a developer C++ using the library will use it:

```
#include <vaucanson/tools/io.hh>

/* to save an automaton */
output_stream << automaton_saver(automaton, converter, format)

/* to load an automaton */
input_stream >> automaton_loader(automaton, converter, format, merge_states)
```

Where:

- `automaton` is the automaton undergoing input or output. Note that the object must already be constructed, even to be read into.
- `converter` is a helper class that is able to convert automaton transitions to character strings and possibly vice-versa.
- `format` is a helper class that is able to convert the automaton to (and possibly from) a character string, using the converter as an argument.
- `merge_states` is an optional argument that should be omitted in most cases. For advanced users, it allows loading a single automaton from several different streams that share the same state set.

About converters

The `converter` argument is mandatory. There are several converter types already available in Vaucanson. See below.

An I/O converter is a function object with one or both of the following:

- an operation that takes an automaton, a transition label and converts the transition label to a character string (std::string). This is called the output conversion.
- an operation that takes an automaton, a character string and converts the character string to a transition label. This is called the input conversion.

Vaucanson already provides these converters:

`vcsn::io::string_out`**, bundled with** `io.hh`**.** Provides the output conversion only. Uses the C++ operator << to create a textual representation of transition labels. Should work with all label types.

`vcsn::io::usual_converter_exp`**, defined in** `tools/usual_io.hh`**.** Provides both input and output conversions. Uses the C++ operator << to create a textual representation of transition labels, but requires also that algebra::parse can read back that representation into a variable of the same type. It is mostly used for generalized automata where transitions are labeled by rational expressions, hence the name.

`vcsn::io::usual_converter_poly<ExpType>`**, defined in** `tools/usual_io.hh`**.** Provides both input and output conversions. Converts transition labels to and from ExpType before (after) doing I/O. The implementation is meant to be used when labels are polynoms, and using the generalized (expression) type as ExpType.

Notes about XML and converters

When the XML I/O format was implemented, the initial converter system was not used. Instead a specific converter system was re-designed specifically for this format.

(FIXME: explain why!)

(FIXME: why hasn't the generic converter for XML been ported back to fsm and simple formats?)

Because of this, when using XML I/O the "converter" argument is completely ignored by the format processor. Usually you can see `vcsn::io::string_output` mentioned.

(FIXME: this is terrible! it must be patched to use an empty vcsn::io::xml_converter_placeholder or something like it).

About formats

The `format` argument is mandatory. It specifies an instance of the object in charge of the actual input or output.

A format object is a function object that provides one or both the following operations:

- an operation that takes an output stream, the caller `automaton_saver` object, and the `converter` object. This is called the output operation.
- an operation that takes an input stream and the caller `automaton_loader` object. This is called the input operation. Note that this operation does not uses the `converter` object, because it should call back the `automaton_loader` object to actually perform string to transition label conversions.

Format objects may require arguments to be constructed, such as the title of the automaton in the output.

Format objects for a format should be defined in a `tools/xxx_format.hh` file.

Vaucanson already provides the following format objects:

`vscn::io::dot(const std::string& digraph_title)`, **in** `tools/dot_format.hh`. Provides an output operation for the Graphviz `dot` subformat. The title provided when buildint the `dot` object in Vaucanson becomes the title of the graph in the output data and a prefix for state names. Therefore the title must contain only alphanumeric characters or the underscore (_), and no spaces.

`vcsn::io::simple()`, **in** `tools/simple_format.hh`. Provides both input and output operations for a simple text format.

`vcsn::xml::XML(const std::string& xml_title)`, **in** `xml/XML.hh`. Provides both input and output operations for the Vaucanson XML I/O format.

(FIXME: why not tools/xml_format.hh with proper includes of headers in xml/?)
(FIXME: really the FSM format should have a format object too.)

Examples

Create a simple dot output for an automaton a1:

```
std::ofstream fout("output.dot");
fout << automa-
ton_saver(a1, vcsn::io::string_output(), vcsn::io::dot("a1"));
fout.close()
```

Output automaton a1 to XML, read it back into another automaton a2 (possibly of another type):

```
std::ofstream fout("file.xml");
fout << automaton_saver(a1, NULL, vcsn::xml::XML());
fout.close()

std::ifstream fin("file.xml");
fin >> automaton_loader(a2, NULL, vcsn::xml::XML());
fin.close()
```

Do the same, but this time using the simple format. The automata are generalized, i.e. labeled by expressions:

```
std::ofstream fout("file.txt");
fout << automa-
ton_saver(a1, vcsn::io::usual_converter_exp(), vcsn::io::simple());
fout.close()

std::ifstream fin("file.txt");
fin >> automa-
ton_loader(a2, vcsn::io::usual_converter_exp(), vcsn::io::simple());
fin.close()
```

Internal scenario

What happens in Vaucanson when you write:

```
fin >> automaton_loader(a1, c1, f1)
```

?

1. function `automaton_loader` creates an object AL1 of type `automaton_loader_` that memorizes its arguments.

2. `automaton_loader()` returns AL1.

3. `operator>>(fin, AL1)` is called.

4. `operator>>` says to format object f1: "hi, please use fin to load something with AL1".

5. f1 scans input stream fin. Things may happen then:

   - f1 finds a state numbered N. Then it says to AL1: "hey, make a new state into the output automaton, keep its handler s1 for yourself and remember it is associated to N". (callback `AL1.add_state`)
   - f1 finds a transition from state numbered N to state P, labeled with character string S. Then it says to AL1: "hey, create a transition with N, P, and S." (callback `AL1.add_transition`). Then:
     - AL1 remembers handler for state N (s1)
     - AL1 remembers handler for state P (s2)
     - AL1 says to converter c1: "hey, make me a transition label from S"
     - AL1 creates transition from s1 to s2 using converted label into output automaton.

6. when f1 is finished, it returns control to `operator>>` and then calling code.

Of course since everything is statically compiled using templates there is no performance drawback due to the intensive use of callbacks.

Convenience utilities

For most formats the (relatively) tedious following piece of code:

```
output_stream << automaton_saver(a, CONVERTER(), FORMAT(...))
```

is also available as:

```
FORMAT_dump(output_stream, a, ...)
```

If available, this convenience utility is defined in `tools/XXX_dump.hh`.
Conversely, the following piece of code:

```
input_stream >> automaton_loader(a, CONVERTER(), FORMAT(...))
```

is usually also available as:

```
FORMAT_load(input_stream, a, ...)
```

If available, this convenience utility is defined in `tools/XXX_load.hh`.
(FIXME: move fsm_load away from fsm_dump.hh!)
As of today (2006-03-17) the FSM format is only available using the fsm_load() and fsm_dump() interface.