Vaucanswig: a dynamic wrapper around Vaucanson

Author: Raphael Poss
Contact: raph@lrde.epita.fr
Version: Id
Date: January 2005

Vaucanswig is a set of SWIG definitions which allow to use Vaucanson in a high-level, dynamic, language such as Python, Perl, PHP or Ruby.

Contents

Introduction

Usage

What is provided?

- Glossary
- What is in a category?
- Algebra
  - Algebra usage
- Automata
- Algorithms

Adding new algorithms

- Example
- Limitations

Python support

Licence

Contact

Introduction

Vaucanson is a C++ library that uses static genericity.

SWIG is an interface generator for C and C++ libraries, that allow their use from a variety of languages: CHICKEN, C#, Scheme, Java, O'Caml, Perl, Pike, PHP, Python, Ruby, Lisp and TCL.

Unfortunately, running SWIG directly on the Vaucanson library does not work: most of Vaucanson features are expressed using C++ meta-code, which means that basically there is no real code in Vaucanson for SWIG to work on.

Vaucanswig comes in between SWIG and Vaucanson: it describes to SWIG some explicit Vaucanson types and algorithms implementations so that SWIG can generate the inter-language interface.

Usage

For any SWIG-supported language, using Vaucanswig requires the following steps:

1. generation of the language interface from SWIG input sources (`.i` files) provided by Vaucanswig,
2. compilation of the interface into extensions to the language library (e.g. dynamically loadable shared package module for Python).
3. loading the extension into the target language.

Vaucanswig provides no material nor tools to achieve these two steps, except for the Python language target (see below). Refer to the SWIG documentation for information about generating language extensions from SWIG input files for other languages.

What is provided?

Glossary

In the next sections, the name "category" will refer to the set of features related to a particular algebraic configuration in Vaucanson.

The following categories are predefined in Vaucanswig:

Category	**Semiring values**	**Monoid values**	**Series**	**Series values**	**Expression values**
`usual`	`bool`	`string`	B<<A*>>	`polynom`	`exp`
`numerical`	`int`	`string`	Z<<A*>>	`polynom`	`exp`
`tropical_min`	`int`	`string`	Z(min,+)<<A*>>	`polynom`	`exp`
`tropical_max`	`int`	`string`	Z(max,+)<<A*>>	`polynom`	`exp`

These are the standard contexts defined in Vaucanson. They are defined in Vaucanswig in the file `expand.sh`.

What is in a category?

For a given category *D*, Vaucanswig defines the following **modules**:

`vaucanswig_D_context`**:** Algebra and algebraic context.

`vaucanswig_D_automaton`**:** Automata types (standard and generalized).

`vaucanswig_D_alg_...`**:** Algorithm wrappers.

`vaucanswig_D_algorithms`**:** General wrapper for all algorithms.

Each of these modules becomes an extension package/module/namespace in the target language.

Algebra

For a given category *D*, the module `vaucanswig_D_context` contains the following **classes**:

`D_alphabet_t`**:** Alphabet element with constructor from a string of generator letters:

```
(constructor): string -> D_alphabet_t
```

`D_monoid_t`: Monoid structural element with the following members:

- standard Vaucanson constructors and operators,
- method to construct a word element from a simple string:

```
make: string -> D_monoid_elt_t
```

- method to generate the identity value:

```
identity: -> D_monoid_elt_t
```

`D_monoid_elt_t`: Word (monoid element) with standard Vaucanson constructors and operators.

`D_semiring_t`: Semiring structural element with the following members:

- standard Vaucanson constructors and operators,
- method to construct a weight element from a number:

```
make: int -> D_semiring_elt_t
```

- methods to generate the identity and zero values:

```
identity: -> D_semiring_elt_t
zero: -> D_semiring_elt_t
```

`D_semiring_elt_t`: Weight (semiring element) with standard Vaucanson constructors and operators.

`D_series_set_t`: Series structural element with the following members:

- standard Vaucanson constructors and operators,
- methods to construct a series element from a number or string:

```
make: int -> D_series_set_elt_t
make: string -> D_series_set_elt_t
```

- methods to generate the identity and zero values as polynoms or expressions:

```
identity: -> D_series_set_elt_t
zero: -> D_series_set_elt_t
exp_identity: -> D_exp_t
exp_zero: -> D_exp_t
```

`D_series_set_elt_t`, `D_exp_t`: Polynom and expressions (series elements with polynom and expression implementations) with standard Vaucanson constructors and operators.

`D_automata_set_t`: Structural element for automata. Include standard Vaucanson constructors.

`D_context`: Convenience class with utility methods. It provides the following members:

- constructors:

```
(constructor): D_automata_set_t -> D_context
(copy constructor): D_context -> D_context
```

- accessors for structural elements:

```
automata_set: -> D_automata_set_t
series: -> D_series_set_t
monoid: -> D_monoid_t
semiring: -> D_semiring_t
alphabet: -> D_alphabet_t
```

- shortcut constructors for elements:

```
semiring_elt: int -> D_semiring_elt_t
word: string -> D_monoid_elt_t
series: int -> D_series_set_elt_t
series: word -> D_series_set_elt_t
series: D_exp_t -> D_series_set_elt_t
exp: D_series_set_elt_t -> D_exp_t
exp: string -> D_expt_t
```

In addition to these classes, the module `vaucanswig_D_context` contains the following **function**:

```
make_context: D_alphabet_t -> D_context
```

Algebra usage

All classes are equipped with a `describe` method for textual representation of values. Example use (Python):

```
>>> from vaucanswig_usual_context import *
>>> c = make_context(usual_alphabet_t("abc"))

>>> c.exp("a+b+c").describe()
'usual_exp_t@0x81a2e60 = ((a+b)+c)'

>>> (c.exp("a")*c.exp("a+b+c")).star().describe()
'usual_exp_t@0x81a20f8 = (a.((a+b)+c))*'

>>> from vaucanswig_tropical_min_context import *
>>> c = make_context(tropical_min_alphabet_t("abc"))

>>> c.series().identity().describe()
'tropical_min_serie_t@0x81ad8b8 = 0'
>>> c.series().zero().describe()
'tropical_min_serie_t@0x81a6de8 = +oo'
```

Automata

For a given category D, the module `vaucanswig_D_automaton` contains the following **classes**:

`D_auto_t`: The standard automaton type for this category.

`gen_D_auto_t`: The generalized (with expression labels) automaton type for this category.

These class provides the following constructors:

```
(constructor): D_context -> D_auto_t
(constructor): D_context -> gen_D_auto_t
(copy constructor): D_auto_t -> D_auto_t
(copy constructor): gen_D_auto_t -> gen_D_auto_t
(constructor): D_auto_t -> gen_D_auto_t
```

For convenience purposes, a `gen_D_auto_t` instance can be constructed from a `D_auto_t` (generalization). The opposite is not possible, of course.

In addition to the standard Vaucanson methods, these classes have been augmented with the following operators:

`describe()`: Give a short description for the object.

`save(filename)`: Save data to a file.

`load(filename)`: Load data from a file. The automaton must be already defined (empty) and its structural element must be compatible with the file data.

`dot_run(tmpf, cmd)`: Dump the automaton to file named `tmpf`, then run command `cmd` on file `tmpf`. The file is in dot format compatible with Graphviz.

Example use:

```
>>> from vaucanswig_usual_automaton import *
>>> a = usual_auto_t(c)
>>> a.add_state()
0
>>> a.add_state()
1
>>> a.add_state()
2
>>> a.del_state(1)
>>> for i in a.states():
...   print i
...
0
2
>>> a.dot_run("tmp", "dot_view")

>>> a.save("foo")
>>> a2 = usual_auto_t(c)
>>> a2.load("foo");
>>> a2.states().size()
2
```

Algorithms

As a general rule of thumb, if some algorithm `foo` is defined in the source file `vaucanson/algorithms/bar.hh` then:

- the module `vaucanswig_D_alg_bar` contains a function `foo`,
- the module `vaucanswig_D_algorithms` contains `D.foo`.

Adding new algorithms

The Vaucanswig generator automatically build Vaucanswig modules from definitions found in the Vaucanson source files.

You can add a new algorithm to vaucanswig simply by adding declarations of the form:

```
// INTERFACE: ....
```

to the Vaucanson headers.

Example

Let's consider the Vaucanson header `foo.hh` in `include/vaucanson/algorithms`, which contains the following code:

```
// INTERFACE: Exp foo1(const Exp& other) { return vcsn::foo1(other); }
template<typename S, typename T>
Element<S, T> foo1(const Element<S, T>& exp);

// INTERFACE: Exp foo1(const Exp& other1, const Exp& other2) { return vcsn::foo2(other1, other2
template<typename S, typename T>
Element<S, T> foo1(const Element<S, T>& exp);
```

Then, after running `expand.sh` (the Vaucanswig generator) for category D, the module `vaucanswig_D_alg_foo` becomes available:

```
foo1: D_exp_t -> D_exp_t
foo2: (D_exp_t, D_exp_t) -> D_exp_t
```

In addition, the special algorithm class `D`, defined in `vaucanswig_D_algorithms`, also contains 'foo1' and 'foo2'.

Limitations

When writing `// INTERFACE:` comments, the following notes must be taken into consideration:

- The comment must stand on a single line. Indeed, `expand.sh` does not currently support multi-line interface declarations.
- The following special macro names are available:

  `Exp:` The expression type for the category.

  `Serie:` The polynom/serie type for the category.

  `Automaton, GenAutomaton:` The automaton types for the category.

  `HList:` A list of state or edge handlers (integers). This type is `std::list<int>` in C++ and a standard sequence of numbers in the target language.

- When accessing automata, a special behavior stands. Instead of writing:

  ```
  // INTERFACE: void foo(Automaton& a) { return vcsn::foo(a); }
  // INTERFACE: void foo(GenAutomaton& a) { return vcsn::foo(a); }
  ```

  one should write instead:

  ```
  // INTERFACE: void foo(Automaton& a) { return vcsn::foo(*a); }
  // INTERFACE: void foo(GenAutomaton& a) { return vcsn::foo(*a); }
  ```

  Indeed, `Automaton` and `GenAutomaton` do not expand to Vaucanson automata types, but to a wrapper type. The real automaton can be reached by means of operator*().

Python support

For convenience purposes, Python interfaces for Vaucanswig are included in the distribution. They are automatically compiled and installed with Vaucanson if enabled. To enable these modules, run the `configure` script like this:

```
configure --enable-vaucanswig
```

Licence

Vaucanswig is part of Vaucanson, and is distributed under the GNU Lesser General Public Licence. See the file `COPYING` for details.

Contact

For any comments, requests or suggestions, please write mail to `vaucanson@lrde.epita.fr`.