Generating and extending Vaucanswig sources

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

The file `build-process.txt` describes how to use Vaucanswig to create a wrapper for Vaucanson in a scripting language. (read it first)

This document instead describes how Vaucanswig itself is generated, currently using the infamous `expand.sh` script.

Contents

The list of Vaucanswig modules

Once generated, Vaucanswig is a set of SWIG modules. This list of modules is algorithmically generated. The overall process to build the list of module names is as follows:

1. put "`core`" in the `MODULES` list.
2. create an auxiliary list `ALGS` of algorithm families.
   (detailed below, gives `alg_sum`, `alg_complete`, ...)
3. create an auxiliary list `KINDS` of algebra contexts
   (contains `boolean`, `z`, `z_max_plus`, ...)
4. extend `ALGS` with "`context`", "`algorithms`" and "`automaton`".
5. make the cross product of `KINDS` and `ALGS` putting a "_" between the two parts of each generated name.
6. add the results of this cross product to the `MODULES` list.

The list of algorithm families (`ALGS` in step 2 above)

In Vaucanswig, an "algorithm family" is the set of algorithms declared in a single Vaucanson header file. Most families declare only one algorithm, but usually with several forms (using overloading). In Vaucanswig, each algorithm family is related to a SWIG source file: `src/vaucanswig_alg_NAME.i` where `NAME` is the name of the algorithm family.

Each family source file contains the following items:

- a link to its C++ header.
- the definition of a bunch of SWIG macros which are able to instanciate the algorithm *declarations* for the type set given as parameters.
- the definition of a bunch of SWIG macros which are able to instanciate algorithm *wrappers* for the set of types given as parameters.

To create the list of algorithm families and associated SWIG sources, the geneeration script proceeds as follows:

1. Find all files in the Vaucanson includes that declare algorithms using the "`// INTERFACE:`" construct.
2. For each such include file, proceed as follows:
    a. Prepend the base name of the file with "`alg_`" to make a "family name".
    b. Create `src/vaucanswig_(family_name).i` containing the relevant SWIG code
    c. Put the generated family name (with prefix) in the `ALGS` list.

The cross-product of contexts and generic code (step 5 above)

This is where you find all the magic. :)

This is the step where *real* code (i.e. non-template) is produced.

The goal of this step is to build the list of SWIG modules names *and* the source file for each SWIG module. The basic idea is simple. It relies on the following two facts:

1. each algorithm family defined above defines macros that take types as parameters and produce non-template declarations and definitions.
2. each algebra context defines a set of types, that fit as parameters in the macros for algorithm families.

Now the rest is quite simple. Since we have two lists `KINDS` (contexts) and `ALGS` (algorithm families), proceed as follows:

```
for each K of KINDS, do:
  for each A of ALGS, do:

    # Step 5.1
    instanciate macros...
    ... from src/vaucanswig_alg_${A}.i
    ... using ${K}
    ... into src/vaucanswig_${K}_${A}.i

    # Step 5.2
```

```
  add "${K}_${A}" to the MODULES list.

# the following step is not fundamental, but required for later
# compilation:

# Step 5.3 (still in the K loop)
add "${K}_context" to src/${K}_automaton.deps

for each algorithm family F, do:

   # Step 5.4
   add "${K}_automaton" to src/${K}_${F}.deps

   # Step 5.5
   add "${K}_${F}" to src/${K}_algorithms.deps
```

The result of steps 5.3, 5.4 and 5.5 above can later be used to create dynamic link dependencies between object code for modules (see `build-process.txt`). It creates the following dependency graph:

```
core -> K1_context -> K1_automaton -> K1_F1 -> K1_algorithms
                                   -> K1_F2 ->
                                   -> K1_F3 ->

     -> K2_context -> K2_automaton -> K2_F1 -> K2_algorithms
                                   -> K2_F2 ->
                                   -> K2_F3 ->

(and so on)
```

The transparency property

At every level, a property can be recognized. If an algorithm `foo()` is declared (C++) in `bar.h`, then:

- "`bar`" is the "algorithm family" of `foo()`
- for each selected context K, exactly one SWIG module exists and is called called "`K_bar`".
- the goal is that at the end of the compilation, in the target scripting language you can write:

```
K_bar.foo()
# (or equivalent)
```

What is *not* automatic

Some work is required from the part of the developer:

- keeping "`// INTERFACE:`" tags in Vaucanson headers.
- deciding a list of contexts to instanciate in Vaucanswig.
- running the generator for Vaucanswig generic code whenever the Vaucanson library is updated.
- distributing the generated generic sources and building rules afterwards.

Things not easy to change *yet*

In this section, K stands for any algebra context.

The set of K -dependent types available in wrapper code in the "`// INTERFACE:`" tags is not yet easily configurable, because it involves a huge piece of hand-written dedicated code.

For the moment, the following types are available for each context `K`:

Name of type	**Description**
Automaton	the automaton type labeled by series
GenAutomaton	the corresponding type labeled by expressions
Series	the type of series in K
Exp	the type of expressions in K
HList	a type for lists of unsigned integer (to be used as automaton handlers where required)

Adding more of these is not difficult, but very tedious. It involves adding a new argument in various argument list in various SWIG macros in the code. These will be documented later.

But still, it remains **very difficult** to bind in Vaucanswig any algorithm that operates on more than one algebra context at the same time. "Very difficult" here means that some major work is required to change Vaucanswig to support this case.