Vaucanson's Developer's Guide

This document does not target users of the Vaucanson library. It is meant for developers and those who wish to contribute code to Vaucanson.

Contents

Tools

Maintainer Tools

We use a number of tools during development which we call *maintainer tools*, because they are not required by the end user. You should have these tools installed on your build machine in order to build a fresh checkout of the Vaucanson repository.

Autoconf, Automake, and Libtool Generate the GNU Build System.

Doxygen Is used to build a reference documentation from comments in the source code.

rst2latex Is used to convert reStructuredText into LaTex. This document is written using reStructuredText. `rst2latex` is often distributed in a package called `python-docutils` or `py-docutils`.

Note about some developer tools

Some tools help to improve the code. Use them liberally!

Valgrind and the C++ standard library

Valgrind help catching incorrect memory usage: double deletes, memory leaks, uninitialized memory readings, and so forth. Usually, to optimize speed, implementations of the C++ library don't free all the memory they allocated unless asked. You should `export GLIBCXX_FORCE_NEW=1` to force GCC's C++ library to free allocated structure.

See the C++ Library FAQ for help for details.

To check memory leaks use `valgrind --leak-check=yes`. To get more details, add `--num-callers=20 --leak-resolution=high --show-reachable=yes`.

Using `gdb` or `valgrind` on TAF-Kit

The executables that are built in `taf-kit/src/` are `libtool` scripts that call the true executables (usually hidden under `taf-kit/src/.libs/`). You cannot run `gdb` directly on these scripts, you should always ask `libtool` to do it for you:

```
% cd taf-kit/src
% export VCSN_DATA_PATH=$PWD/../../data
% libtool --mode=execute gdb ./vcsn-int-b
(gdb) run determinize x.xml
...
```

It's often more convenient to run the scripts from taf-kit/tests because they export `VCSN_DATA_PATH` and run the corresponding executable from `taf-kit/src` for you. In that case you have to use the `PREVCSN` environment variable to specify these `libtool` options:

```
% cd taf-kit/tests
% PREVCSN='libtool --mode=execute gdb' ./vcsn-int-b
(gdb) run determinize x.xml
...
```

The same commands can of course be used to run other tools like Valgrind. Here is how to run TAF-Kit under Valgrind and attach a debugger on the first error:

```
% cd taf-kit/tests
% PREVCSN='libtool --mode=execute valgrind --db-attach' ./vcsn-int-b
```

A note for Darwin users: because your system comes with another tool called `libtool`, GNU `libtool` is usually installed as `glibtool`. Alternatively, you may want to use the copy of `libtool` output by `configure` at the root of Vaucanson's build tree.

Debugging STL

The GNU standard C++ library comes with some useful debugging features. Just add `-D_GLIBCXX_DEBUG` to your `CPPFLAGS` when compiling Vaucanson.

You are likely to encounter issues with *singular* iterators. It refers to default-constructed iterators, which is not the same thing as being the `end()` of a given container. They are write-only according to the STL, and copying them is not allowed.

As a result, the following code is incorrect:

```
std::vector<std::list<int>::iterator> is(10);
```

This is because this vector constructor builds a single default object (here a default `std::list<int>::iterator`, then *copies* it ten times, which is (ten times) forbidden. As there is no clear means to rewrite the code to avoid this violation, you might want to use `end()` iterators, which can be used in comparisons, can be copied, but cannot be dereferenced:

```
std::vector<std::list<int>::iterator> is(10, std::list<int>().end());
```

Version control

Access to the Git repository

The master Git repository is on `git.lrde.epita.fr`. It can be cloned anonymously using:

```
git clone git://git.lrde.epita.fr/vaucanson
```

For write access, send your public SSH key to adl@lrde.epita.fr, then use the following command instead:

```
git clone git@git.lrde.epita.fr:vaucanson
```

Layout of the Git repository

The repository contains the following important branches

- branch `master` holds the latest release as well as small bugfixes and straightforward patches. It should be a stable branch.
- the `exp/*` branches contain any development that need to mature before inclusion in a release. Use one branch per feature. Any experimental branch of development should be based off `master` or another experimental branch.
- branch `hive` merges all the above experimental branches. It should contain only merge commits. Never base another branch off `hive`.

Git Workflow

The suggested workflow bellow applies only when you are developing a new feature, or anything that will take time. Any straightforward patch (like a quick bug fix) can of course be applied directly to `master`. Other work should be done on an experimental branch, then merged into `hive`. Once they have matured, experimental branches will eventually be merged into `master` when preparing a release.

- Initially, you should work locally. There is no need to show your early developments (unless you seek comments). By working locally it's easier for you to reorganize your work or even start over, and you often need to do that in the early stages of development.

- Create a branch on your local clone with:

  ```
  git checkout -b exp/my-new-feature origin/master
  ```

  You may need to base your branch off another experimental branch if there is some dependency, but please never base your branch off `hive`, it would make it difficult to merge your feature in a release without merging in all the other features that are in `hive`.
- Work on that branch locally until you have something reasonably stable that can be made public. Occasionally, update your branch to the latest upstream changes with:

  ```
  git pull --rebase
  ```

  (This will `fetch` all recent revisions from the server, and rebase your branch against the new version of `origin/master`.)

- Once you are ready to publish your branch, review your changes with `git log`, `gitk`, or any other interface. If needed, cleanup and reorder your history using `git rebase -i origin/master`.
  - Push the branch with:

    ```
    git push origin exp/my-new-feature
    ```

    If you want to publish your branch under a different name, the syntax is:

    ```
    git push origin private-name:exp/public-name
    ```

    Please push only one branch at a time. There is a script on the server that post a news for each push, and it will behave strangely if you push two branches that share some new patches.
  - Your local branch is still tracking `origin/master`, but now that a copy of that branch is public and that other people can write to it, it is probably more sensible to change your local branch to track `origin/exp/my-new-feature`:

    ```
    git config branch.exp/my-new-feature.remote origin
    git config branch.exp/my-new-feature.merge refs/heads/exp/my-new-feature
    ```

- You may now continue to work locally in your `exp/my-new-feature` branch to prepare another set of patches. At any point you may run `git pull --rebase` to rebase your work on top of any recent changes pushed to `origin/exp/my-new-feature`.

  It is better to avoid repeated merges when working on a feature branch, but if for instance you absolutely need to retrieve some fixes from `master`, you can run:

  ```
  git merge origin/master
  ```

  You may also want to perform such a merge after a release (that does not include your branch).

- You can push any new development made into your branch using the same syntax as above:

  ```
  git push origin exp/public-name
  ```

  Again, please do not use `git push origin` because that will push all your local branches that match a branch on the server, and we only want to push one branch at a time.

- After you have pushed new patches to a public branch, it is often a good idea to update the `hive`. (It's obviously a bad idea if that branch is known to be in a sorry state.)

  :: git checkout -b hive origin/hive git merge exp/my-new-feature git push origin hive

  Git can merge several branches at once, so you can also write `git merge exp/feature1 exp/feature2 exp/feature3`. This will result in one merge commit (with 4 parents) instead of three separate merge commits.

  The `hive` branch should contain only merge commits. So if at any time you have to fix a bug on `hive`, please fix it in the appropriate `exp/*` branch and merge it back into `hive`.

- When preparing a release we will want to pick *some* of the mature `exp/*` branches and merge them into `master`. (This is why you should never fix `hive` directly.) Once a branch has been merged into `master`, it can be erased.

Check list for feature branches

The experimental branches (`exp/*` and `hive`) obviously do not need to be perfect. Ideally we would like any public branch to compile and pass all its tests. Please keep your broken branches on your local repositories. As time goes we expect each developed branch to mature and improve in quality.

Here is a non-exhaustive list of items you should consider to assess the maturity of a branch.

- Does the branch compiles and passes tests?
- Is the feature documented? This includes the Doxygen documentation (any public C++ interface should be documented), the LaTeX documentation (for instance new TAF-Kit features should be documented), any README file around, but also comments in the code.

  Documentation does not concern only new feature. When working on an existing feature you should probably also update and maybe improve its documentation.
- When documenting an algorithm, it is important to
  - Define precisely what is being computed.

    For instance the Doxygen documentation for the function `realtime()`, that converts an automaton to a *realtime automaton*, is a good place to define what a *realtime automaton* is. Other places that use realtime automata should point to this definition.
  - Explain how the computation is performed if that is easy, or refer to a place (preferably give the full references to a paper) where we can learn how the algorithm works.

    Note that citing a paper does not exempt you from documenting your work in other ways. The reader should not have to read a paper to understand what a function does and how to use it.
  - Explicit any assumptions on the input (*preconditions*).
- It's often a good idea to document the algorithm before actually implementing it.
- Testing is important, and difficult. One error students often do is to write a small program to test an algorithm on a simple case, and then commit the algorithm without their test program. This is bad in many ways:
  - The test is not automated (in the `make check` sense), so your are wasting some time testing things by hand.
  - There is only one test, and it is unlikely to cover a large class of inputs. Please consider basic automata as well as extreme cases that often cause problems (two examples are the empty automata, and automata whose initial or final functions have a nonempty support). Please also check how your algorithm behave on erroneous input (if you have the right preconditions that should not be a problem).
  - The test is not public, so other people may not warn you that the test is so weak that it means almost nothing. An example of weak test is running an algorithm on an automaton without checking the output in any way: the only thing you learn here is that the algorithm is not broken to the point that it would segfault the given automaton. (There many tests like this in Vaucanson, please don't add any more.)

  - Because the test is not public, other people can break the algorithm without noticing. This has occurred many times in the history of Vaucanson, with algorithms that where either not tested, or had very weak tests. You should consider testing as a kind of defensive programming: you want to add as much tests as needed to guarantee, no only that your algorithm works perfectly in all situations, but also to ensure that this will still be true in the future, regardless of how Vaucanson is changed.
  - It's a good idea to test a generic algorithm in different contexts, but do not go overboard. There is often little point in testing an algorithm with different kinds of letters (characters, pairs, integers...), unless that algorithm deals with the properties of these letters.
  - It's often a good idea to write some tests *before* even writing the new feature, so that you know what your goal is.
  - For serious bug fixes, you should write a test case that reproduce the bug before attempting to fix that bug. Then commit the test case along with the fix to ensure that this bug will never have to be fixed again.

- If your branch changes the name of a symbol, use `git grep` to make sure you caught all occurrences.
- Please remove any cruft like disabled code that has been commented out or files that are no longer used. Keeping a trace of this history is the job of Git.

Development

Faster builds

Vaucanson takes a long time to build, but the time can be reduced dramatically with a few simple measures.

Fast machine, plenty of memory

Use a fast machine with plenty of memory. 2GB seems to be a minimum for an optimized build; any less causes severe swapping.

Local disk

Build on a local disk, not in an NFS mount.

Non-optimized build

Use:

```
./configure ... CCFLAGS="-g -ggdb -Wall" CXXFLAGS="-g -ggdb -Wall"
```

N.B. The variables go at the end of the line.

Parallel `make`

If you have N CPU cores, use `make -jN` to build up to N targets at the same time. Under Linux you can find the number of cores by looking at `/proc/cpuinfo`.

ccache

At the time of writing (Vaucanson 1.2.95a) running `make CC='ccache gcc' CXX='ccache g++'` requires 100MB of cache, and running `make CC='ccache gcc' CXX='ccache g++' check` requires 500MB of cache.

However if you are switching between multiple GIT branches (and you should), you will may want to multiply these values by your number of branches. A cache size of 4GB can be setup with:

```
ccache -M 4G
```

Note the because Vaucanson usually compiles a lot of header files into a single program, it's likely that a change to a *common* include file will invalidate most of your cache.

distcc

The following instructions are for LRDE users, but may be adapted to other places.

1. Wake up as much machines as you want with `lrde-wakeonlan`. Use `lrde-wakeonlan .` to wake up all hosts.

2. Configure with:

   ```
   ./configure CC=gcc-4.2 CXX=g++-4.2
   ```

   You need to specify the GCC version number to make sure all machines use the same compiler.

3. Update the `.distcc/hosts` files with the list of build hosts available. It should look something like:

   ```
   berville-en-caux.lrde.epita.fr/2,lzo
   marvejols.lrde.epita.fr/2,lzo
   whiteagonycreek.lrde.epita.fr/2,lzo
   --randomize
   ```

   The script `~adl/usr/bin/update-distcc-hosts` can create this file automatically for you.

4. Run `make -jN CC='distcc gcc-4.2' CXX='distcc g++-4.2'` where `N` is the number of available hosts. Beware that preprocessing and linking are still done locally, so you may not want to use more than `-j8` on a single core CPU (or use `make`'s `-l` flag to limit the load).

   If you want to use both `ccache` and `distcc`, type `CCACHE_PREFIX=distcc make -jN CC='ccache gcc-4.2' CXX='ccache g++-4.2'`.

Making a release

Don't do these steps from memory.

- Make sure the last run of the autobuilder was successful.
- Check trac to make sure there are no important pending tickets.
- Run `make maintainer-check` in `doc/manual`.
- Make sure `doc/NEWS.txt` is up-to-date. (Mention important known bugs!)
- Make sure `doc/README.txt` is up-to-date.
- Make sure `doc/HACKING.txt` is up-to-date.
- Make sure AUTHORS is up-to-date.

Make sure your system has up-to-date tools (Autotools, Swig, Doxygen, ...) before continuing.

- Bump the version number in `configure.ac`.
- Run `bootstrap`.
- Write the `ChangeLog` entry for all the above changes (But don't commit it before `distcheck`.)
- Run `make distcheck`.
- Commit all changes on success. Commit suicide otherwise.
- Tag the repository for the release.
- Append a `a` to the version number in `configure.ac` and commit this new change so that the next run of the autobuilder won't create a release.
- Copy the files created by `distcheck` to `/lrde/dload/vaucanson/` don't forget to `chmod a+rX` all files and directories, and to update the `latest` link.
- Create the release page on the LRDE wiki.
- Update the Vaucanson page to point to it.
- Update the Vaucanson download page to point to the release.
- Send an announcement to vaucanson@lrde.epita.fr. The text of the announcement should explain what Vaucanson is (so we can forward the mail to another mailing list) and should include the list of major improvements since the last version (i.e., the top of `NEWS`). Do not assume that people will follow links to get details.
- If the release is a beta release, or an intermediate release before a major release, make it clear in the announcement and on the wiki.
- Install any new major release on `vcsn.enst.fr`.
- Complete and detail this list with what was missing (whatever will help the next guy doing the release).

Template arguments naming convention

Template arguments:

- `A` : Automaton structure.
- `AI` : Automaton implementation.
- `S` : Series.
- `SI` : Series implementation.
- `W` : a Word.

In the case where multiple possibilities could be used, suffix the template argument with the appropriate numbering. For example, to enable the use of two different automaton implementations for each argument of an algorithm:

```
template <typename A, typename AI1, typename AI2>
Element<A, AI1>
algorithm(const Element<A, AI1>& a1, const Element<A, AI2>& a2>);
```

Macros to handle with care

The `VCSN_GRAPH_IMPL` macro must only appear in three locations:

- `include/vaucanson/context`
- `include/vaucanson/automata/generic_contexts`
- `include/vaucanson/misc/usual_macros.hh` (be careful when defining new macros using it)

Any other use is irrelevant and may be very harmful. Moreover this macro must never be used in a file with guards.

How can I choose a specific graph implementation?

Use `configure`'s `--with-default-graph-impl` option to control the default graph implementation of the library. The default is `bmig` (a graph represented using Boost Multi Indexes), the other choice is `listg` (a graph represented using adjacency lists). For instance if you want to compile Vaucanson using `listg` by default, use:

```
./configure --with-default-graph-impl=listg
```

The Vaucanson libraries will be compiled with this default implementation, but if you do not use these libraries you can switch the default graph implementation at any time using the `VCSN_DEFAULT_GRAPH_IMPL` preprocessor macro, for instance:

```
make CPPFLAGS=-DVCSN_DEFAULT_GRAPH_IMPL=listg myprogram
```