Autotools for PIPS

Serge Guelton

September 2, 2024

Abstract

This document describes the new build infrastructure of PIPS, based on the famous autotools suite and completes the PIPS developer guide.
It describes

- the meaning of each involved file 3;
- the installation process 4 for PIPS users and developers.
- the maintenance processes 5 for PIPS developers;

This new infrastructure allows better portability and quite faster (re)compiling taking full advantage from some caching and multicore processors.

You can get a printable version of this document on http://www.cri.ensmp.fr/pips/auto_pips.htdoc/auto_pips.pdf and a HTML version on http://www.cri.ensmp.fr/pips/auto_pips.htdoc

1 Introduction

Building a large software like PIPS is quite complicated:

1. several source languages;
2. many tools involved;
3. unusual automatic header file generations.

If you want to ensure a good level of portability, you have to rely on portable tools.

If you want to ensure a good level of maintainability, you have to rely on external, asserted tools.

As PIPS targets *nix based systems and is written mainly in C, autotools appear as a *de facto* standard. It is especially known for enforcing good portability between MacOs, Linux and BSD. Through autoconf, it separate configuration step from build step. Through gnulib, it ensures portability of non-standard C functions.

As of now, PIPS compiles on Linux, BSD and MacOs Operating Systems. It can be compiled using either gcc or icc.

You should notice that indeed PIPS has currently 2 different build process that lives in parallel[1], the GNU-make-based traditional infrastructure and this

[1]Of course it is logical for a parallelizer to have 2 build infrastructures that can be used indifferently in parallel! ☺ It is also useful for fault-tolerance.

new one based on autotools. In this way we can develop and improve the new build infrastructure without hurting traditional users.

Besides better portability support, the autotools-based build infrastructure is quite faster that the old one and can exploit available multiprocessors for example on GNU-make when using the --jobs=... option to specify the number of make process to use.

The nasty side effect of having 2 build methods is that, when adding new stuff in PIPS, you should declare them in both build infrastructures to avoid having different contents in PIPS according to the build infrastructure used. So refer to the companion of this guide too, the PIPS Developer Guide http://www.cri.ensmp.fr/pips/developer_guide.htdoc/developer_guide.pdf or http://www.cri.ensmp.fr/pips/developer_guide.htdoc.

2 Prerequisites

In this section, we shortly list all packages needed to use auto-pips. Note that those packages are only needed for developers, not for users:

- autoconf
- automake
- libtool
- pkg-config

3 Infrastructure Organization

In this section, we describe the configuration files used by the several tools involved in PIPS build process.

3.1 autoconf

autoconf manages the configuration of the build process. Involved files are

configure.ac: central place for configuration. Running ‘autoreconf -vi‘ will produce a configure script from it.

makes/m4: auxiliary directory where m4 configuration macros are stored. It is read by ‘autoreconf‘.

Makefile.am: top-level Makefile.am contains a macro variable definition ACLOCAL_AMFLAGS where ‘autoreconf‘ will find its additional m4 sources. Running ‘autoreconf -vi‘ will produce a Makefile.in for each Makefile.am

configure: is the portable configuration script generated by ‘autoreconf‘. Running it will turn each Makefile.in into a regular Makefile.

config.status: is a script generated by configuration step, that memorizes configuration parameters.

For in-depth documentation of autoconf, feel free to read http://www.gnu.org/software/autoconf/manual.

3.2 automake

automake manages the set of makefiles involved in the build process.

Each Makefile.am in source repository describes the build process for this repository. It follows the 'make' syntax, without needing GNU-make extensions for the sake of portability.

For in-depth documentation of automake, feel free to read http://www.gnu.org/software/automake/manual.

3.3 gnulib

gnulib manages portability of C functions across *nix flavors. Its whole configuration is stored in src/Libs/gnulib and src/Libs/gnulib/m4. A few lines have been added in configure.ac to manage gnulib configuration.

For in-depth documentation of gnulib, feel free to read http://www.gnu.org/software/gnulib/manual.

4 Installation Processes

An automated script does all the nasty things for you if you want to install a *production* version, that is to say copies of Linear, Newgen and Pips trunks. Just begin by typing the following lines in the directory where you want your production version to be installed (for instance ~/Pips4u/prod)

```
wget http://ridee.enstb.org/pips/get-pips4u.sh
chmod u+x get-pips4u.sh
./get-pips4u.sh --help
```

The last command displays all the options available for get-pips4u.sh. In particular, check the default value for the --prefix command (~/pips4u-0.1). This is the directory where the libraries and executables are installed. If you also want to work on development branches (strongly recommended), this may not be convenient as you may want to have several versions available at the same time. For that purpose, you can specify another installation directory with:

```
./get-pips4u.sh --devel --prefix ~/Pips4u/prod/auto-root
```

or whatever location pleases you. Launching this script does (almost) everything for you, from checking out PIPS sources to compiling and installing the libraries and executables.

After installation, if you want to activate some extra PIPS modules such as hpfc or pyps for instance, you can invoke:

```
cd ~/Pips4u/prod/pips4u-0.1/src/pips-0.1/_build
./config.status -V
```

The last commant gives you the options previously used for the configure. Then, from the very same directory execute:

```
../configure ...the same options... --enable-hpfc --enable-
    pyps
```

If you want to be able to run the validation, use the --enable-devel-mode option. Beware that it also sets the compilation flags to -Wall -Werror -O0. After all this you have to recompile and install by typing

```
make; make install
```

in the `_build` directory.

Now, you may also want a *development* branch. This is not automated, and you have to do it by yourself. Here are some guidelines to achieve this.

First get your development area from svn (in `~/Pips4u/dev` for instance), and create a development branch:

```
svn co http://svn.cri.ensmp.fr/svn/pips/branches/luther dev
cd dev
svn cp http://svn.cri.ensmp.fr/svn/pips/trunk my-branch-name
svn commit my-branch-name
cd my-branch-name
```

Then you have to get the `PATH`, `PKG_CONFIG_PATH` and `LD_LIBRARY_PATH` values used for the configure in your production building directory:

```
pushd  ~/Pips4u/prod/pips4u-0.1/src/pips-0.1/_build
./config.status -V
popd
```

And perform the configure:

```
autoreconf -vi
mkdif _build
cd _build
../configure --disable-static --prefix=~/Pips4u/dev/my-
    branch-name/auto-root PATH=... PKG_CONFIG_PATH=...
    LD_LIBRAY_PATH=... --enable-devel-mode
```

where the ... stand for the values retrieved from the production environment, and where you can add whatever –enable options you want.

At last, don't forget to compile and then install in you development installation directory (here `~/Pips4u/dev/my-branch-name/auto-root`).

In some cases, `PIPS` may be included in another distribution and you may build `PIPS` differently. For example in Par4All, where only this `autotools` build method is used, this is done by the Par4All installation process and you do not need to care about the previous installation script.

5 Maintenance Processes

This section describes the process to follow when changing build infrastructure.

5.1 Adding a `C` source file in an existing `PIPS` library

Let us assume you want to add the file `pips.c` into library `src/Libs/ri-util`. First make sure your source file includes pips configuration header, by adding

```
#ifdef HAVE_CONFIG_H
    #include "pips_config.h"
#endif
```

at the top of your source file, before any other include.

The only thing you have to do then is to add your file in the macro variable suffixed `_SOURCES` in `src/Libs/ri-util/Makefile.am` That is

```
libri_util_la_SOURCES=eval.c  ... size.c
```

Becomes

```
libri_util_la_SOURCES=eval.c  ... size.c pips.c
```

5.2 Adding a C header file in an existing PIPS library

Let us assume you want to add the file `pips.h` into library `src/Libs/ri-util`. You will have to modify `src/Libs/ri-util/Makefile.am` Ask yourself the question: Do I want to install the header file with the distribution ?

- If the answer is yes, add your file to the `include_HEADERS` macro variable in, or create it if it does not exist.
- If the answer is no, add your file to the `dist_noinst_HEADERS` macro variable, or create it if it does not exist.

That is write something like this

```
include_HEADERS=pips.h
```

`automake` provides a fine grain control over what gets installed and distributed.

5.3 Adding a TEX file in an existing PIPS directory

Let us assume you want to add the file `pips.tex` into library `src/Libs/ri-util`. You will have to modify `src/Libs/ri-util/Makefile.am`

First, beware that documentation is not built by default. It is only built when user activates configure flags `--enable-doc`.

So everything you do in a makefile that is relevant to documentation must be guarded by `WITH_DOC` The `automake` variable for documentation is `dist_noinst_DATA` for sources and `doc_DATA` for output. That is

```
if WITH_DOC
dist_noinst_DATA=pips.tex
doc_DATA=pips.pdf
endif
```

In addition to this, you have to supply `automake` rules to build PDF from TEX files, using the directive

```
include $(top_srcdir)/makes/latex.mk
```

if it is not already there.

5.4 Adding a library

This one is a bit more difficult. In the following, we assume you want to add `mylib` into `src/Libs`.

There are many steps involved, follow them carefully:

1. create a directory `mylib` into `src/Libs`;
2. add `mylib` to the `PIPS_SUBDIRS` macro variable in `src/Libs/Makefile.am`;

3. add `mylib/libmylib.la` to the `libpipslibs_la_LIBADD` macro variable in `src/Libs/Makefile.am`;

4. add following template in `src/Libs/mylib/Makefile.am`

```
TARGET  = mylib
include_HEADERS = $(TARGET).h
BUILT_SOURCES=$(TARGET).h
include $(top_srcdir)/makes/cproto.mk
noinst_LTLIBRARIES=libmylib.la
lib_mylib_la_SOURCES= src0.c src1.c ... srcn.c

include $(srcdir)/../pipslibs_includes.mk
```

   Where

   `TARGET` is used to avoid redundancy and to communicate with `cproto.mk`.

   `include_HEADERS` specifies that you want to distribute the header generated by ‘`cproto`‘.

   `BUILT_SOURCES` specifies that ‘`cproto`‘ generated header must be built before anything else.

   `include $(top_srcdir)/makes/cproto.mk` specifies how to use ‘`cproto`‘.

   `noinst_LTLIBRARIES` specifies the name of local libraries.

   `lib_mylib_la_SOURCES` specifies the sources of your library.

   `include $(srcdir)/../pipslibs_includes.mk` sets preprocessor include path correctly.

5. add `src/Libs/mylib/Makefile` to the `AC_CONFIG_FILES(...)` macro function parameters in `configure.ac`;

5.5 Adding a program check

For uncommon build, one may need to depend on an extra program. Then comes the distribution issue: how can we assert the program is installed on user/developers machines ? First you have to ask yourself: Is the new feature that depends on this program critical or not ? If not, you will add an optional dependency. Otherwise it is a mandatory dependency.

You will basically add your check by filling a call to macro function `AX_CHECK_PROG(`*`prog_name`*`)` in `configure.ac`. This will perform the check for the program, trying to find it in current `$PATH` or in env variable `${`*`PROG_NAME`*`}`.

The macro variable `$(`*`PROG_NAME`*`)` will be available in your `Makefile.am`.

The last step is to attach the result of the check to a dependency. That way, the configure will fail or not depending on the result of the check. To do so, you will use the macro function `AX_DEPENDS(`*`feature`*`,`*`list-of-dependencies`*`)`. If you have a mandatory dependency, add the name of the program to the `AX_DEPENDS([minimum],[...])` line. Otherwise, add it to the optional `AX_DEPENDS(...)` of your choice. To fully understand usage of `AX_DEPENDS(...)`, please read section on passes5.6.

5.6 Adding a pass

2bedone

6 Additional Checks

`automake` generates a `check` rule for ‘`make`‘, but this rule is not used (yet). Instead you can try one of the following, at the top of your build directory:

- ‘`make check-includes`‘: checks if a source file does not include useless pips headers. It is based on the ‘`pipslibsdeps.py`‘ script which has some extra features, try ‘`pipslibsdeps.py --help`‘ !
- ‘`make check-properties`‘: check if a property is defined in `pipsmake-rc.tex` but never referenced;
- ‘`make inspect-symbols`‘: checks exported but unused symbols for each pips library.