Why Do We Build Our Software in a Different Way Than Everybody Else ?

G.Gaycken

Laboratoire Leprince-Ringuet - École polytechnique

Cambridge, 4-6 April 2006

Outline

1. Library dependencies
2. A portable, standardised build system, GNU autotools
3. Standardising the installation

How We Build and Install Software

- Handcrafted GNU make files
  $\longrightarrow$ no standard enforced.
  $\longrightarrow$ functionality varies from package to package.
- Each package resides in its own directory
  (source code, header files, executables and libraries)
  $\longrightarrow$ deviation from UNIX philosophy
  $\longrightarrow$ tedious to find libraries, header files.

(Software in Europe: LCIO, Marlin, Mokka, etc.)

How We Build and Install Software

- Handcrafted GNU make files
  $\longrightarrow$ no standard enforced.
  $\longrightarrow$ functionality varies from package to package.
- Each package resides in its own directory
  (source code, header files, executables and libraries)
  $\longrightarrow$ deviation from UNIX philosophy
  $\longrightarrow$ tedious to find libraries, header files.

(Software in Europe: LCIO, Marlin, Mokka, etc.)

Dependencies of The Libraries

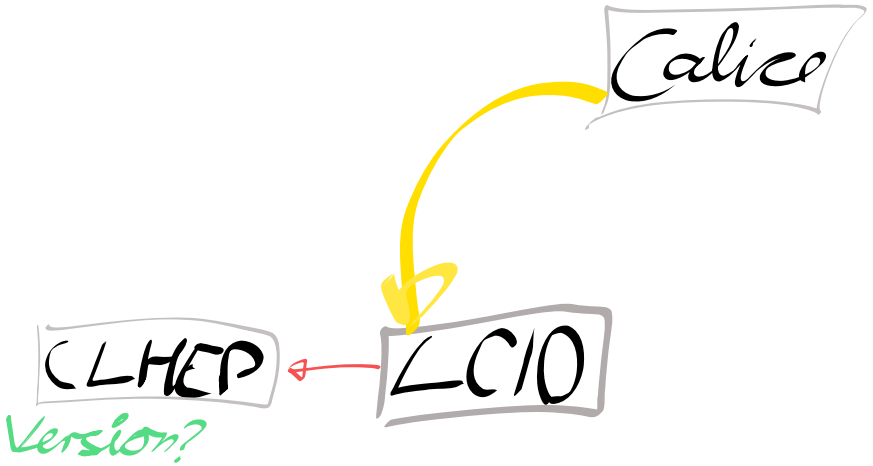
Example: The LCIO based Calice software:

Dependencies of The Libraries

Example: The LCIO based Calice software:

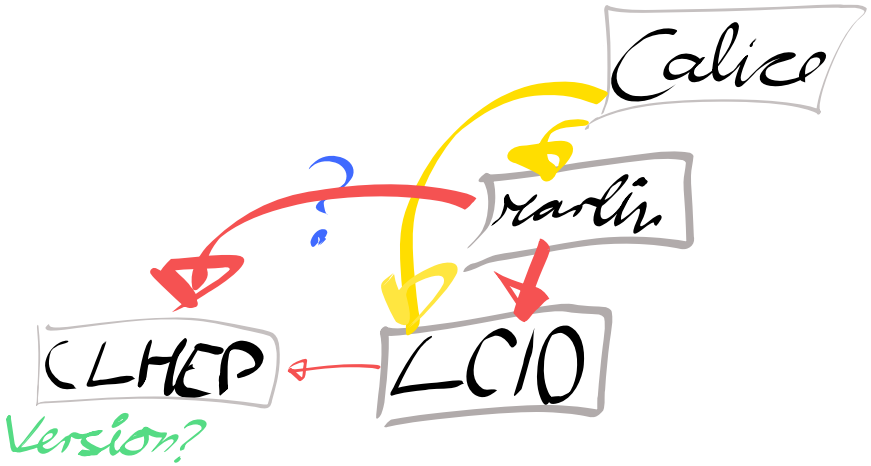
Dependencies of The Libraries

Example: The LCIO based Calice software:



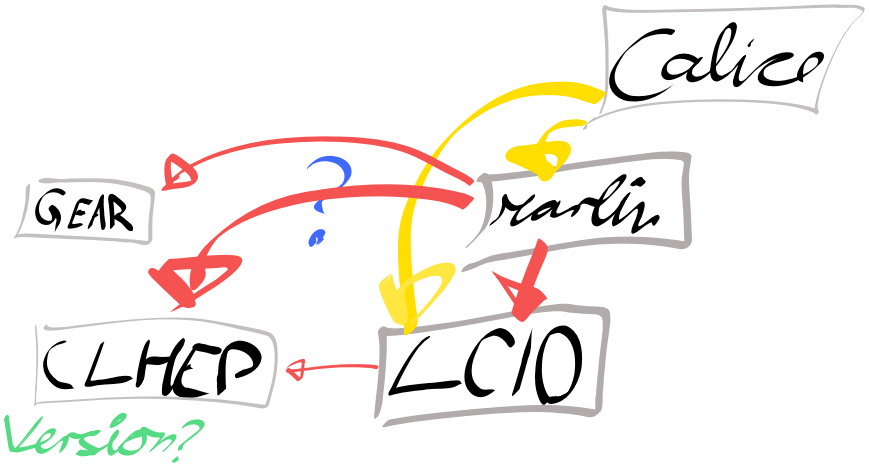
Dependencies of The Libraries

Example: The LCIO based Calice software:



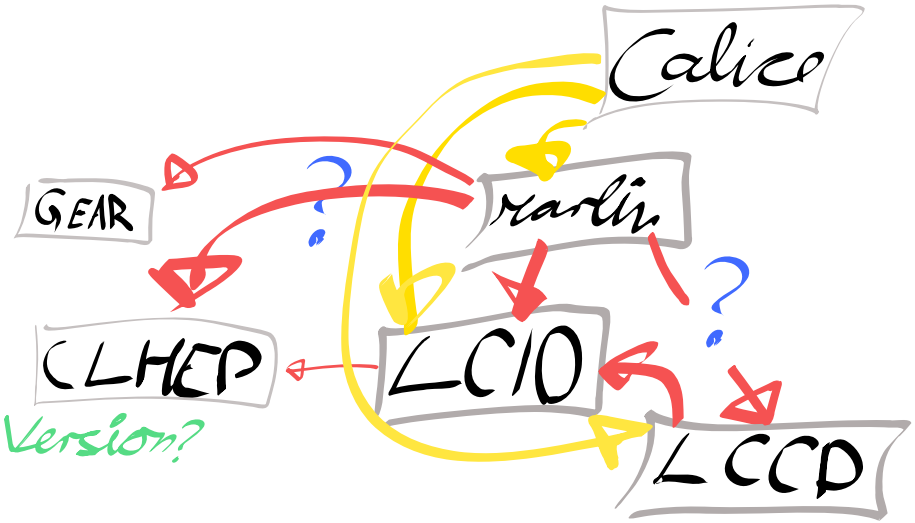
Dependencies of The Libraries

Example: The LCIO based Calice software:



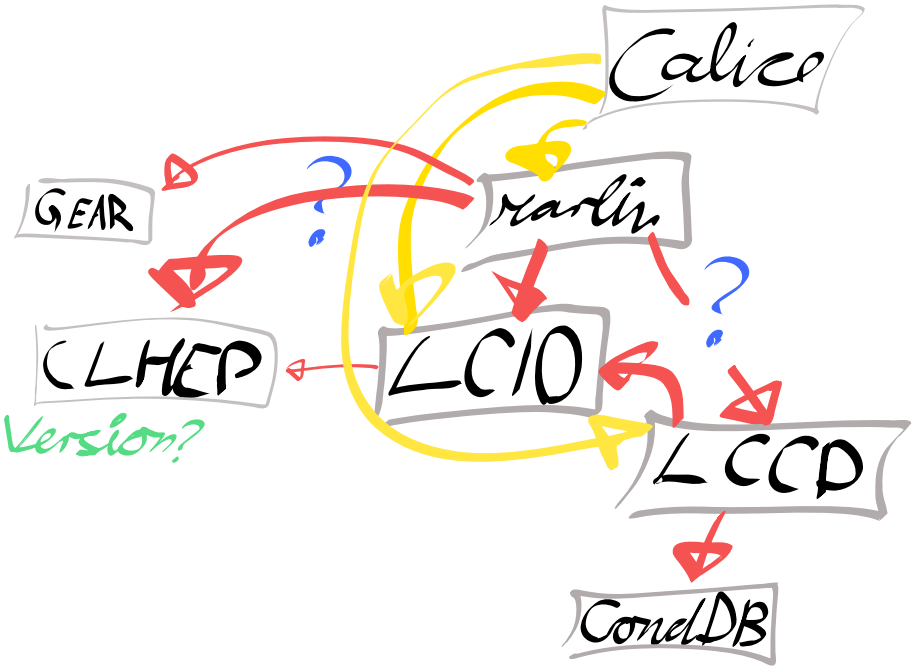
Dependencies of The Libraries

Example: The LCIO based Calice software:



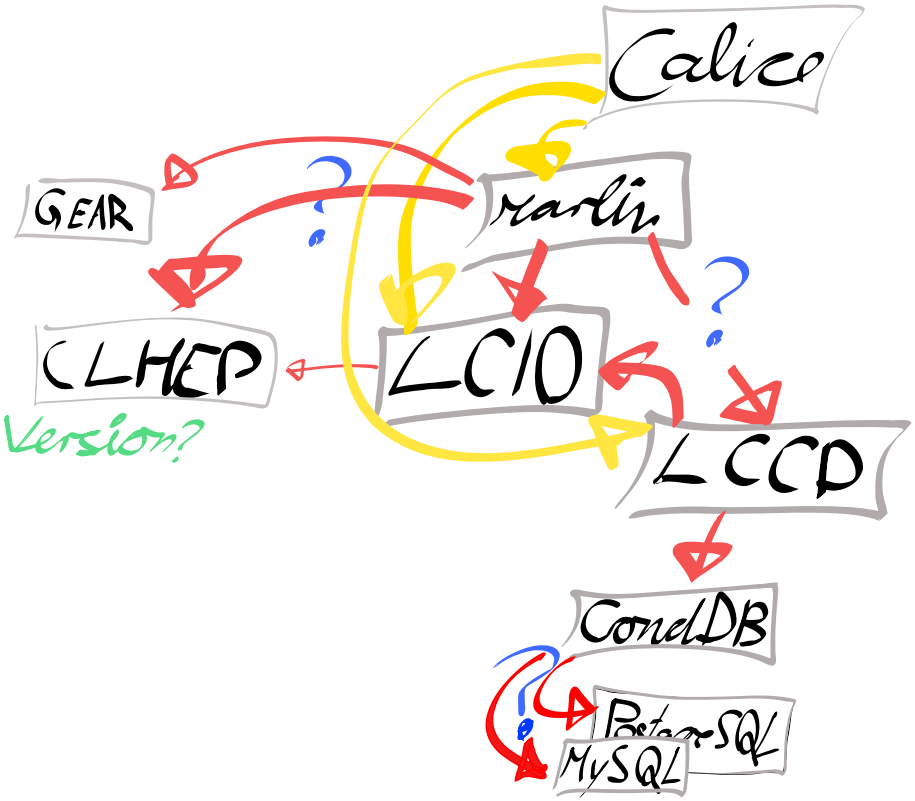
Dependencies of The Libraries

Example: The LCIO based Calice software:



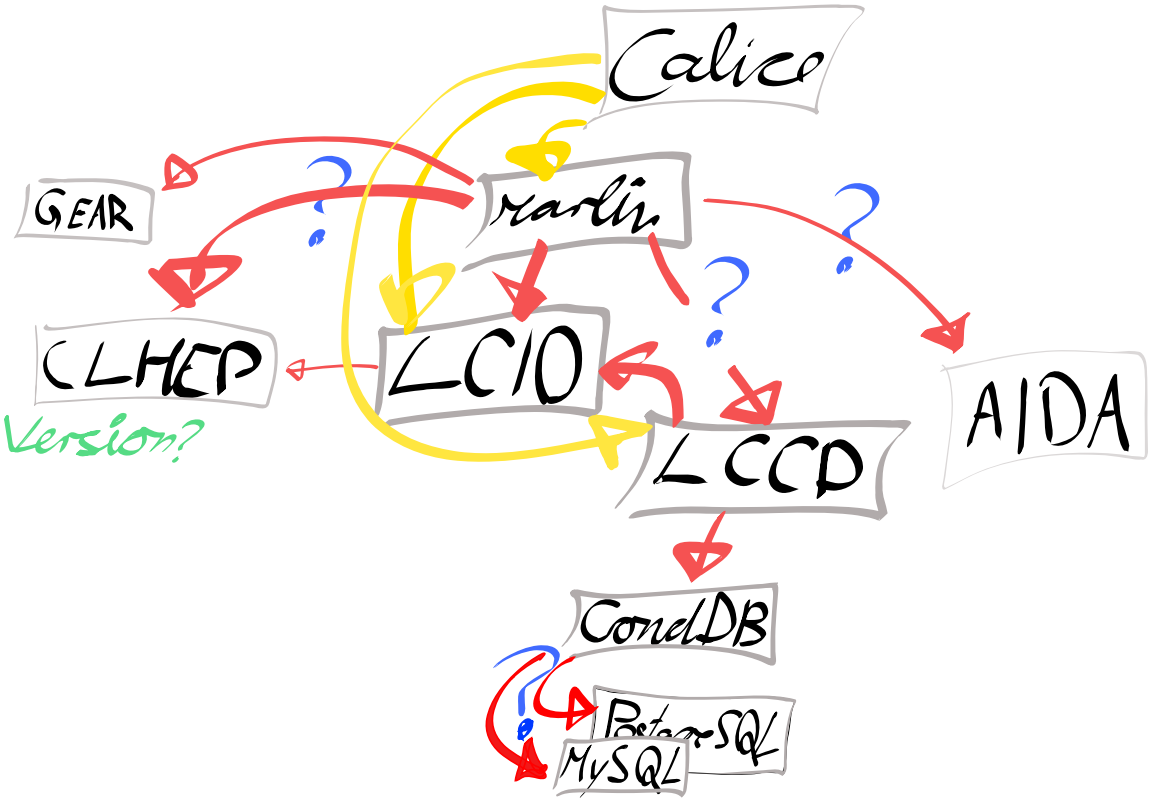
Dependencies of The Libraries

Example: The LCIO based Calice software:



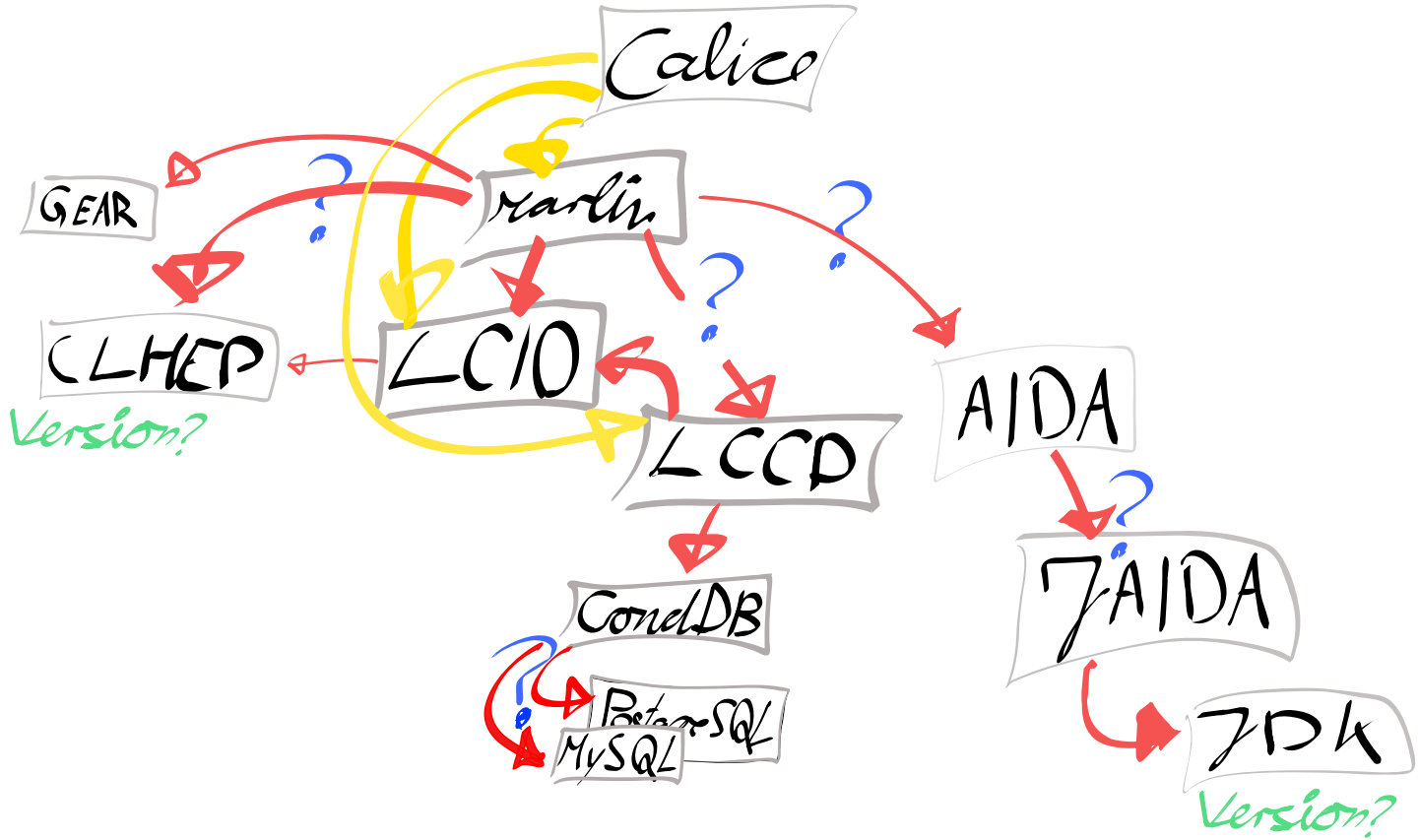
Dependencies of The Libraries

Example: The LCIO based Calice software:



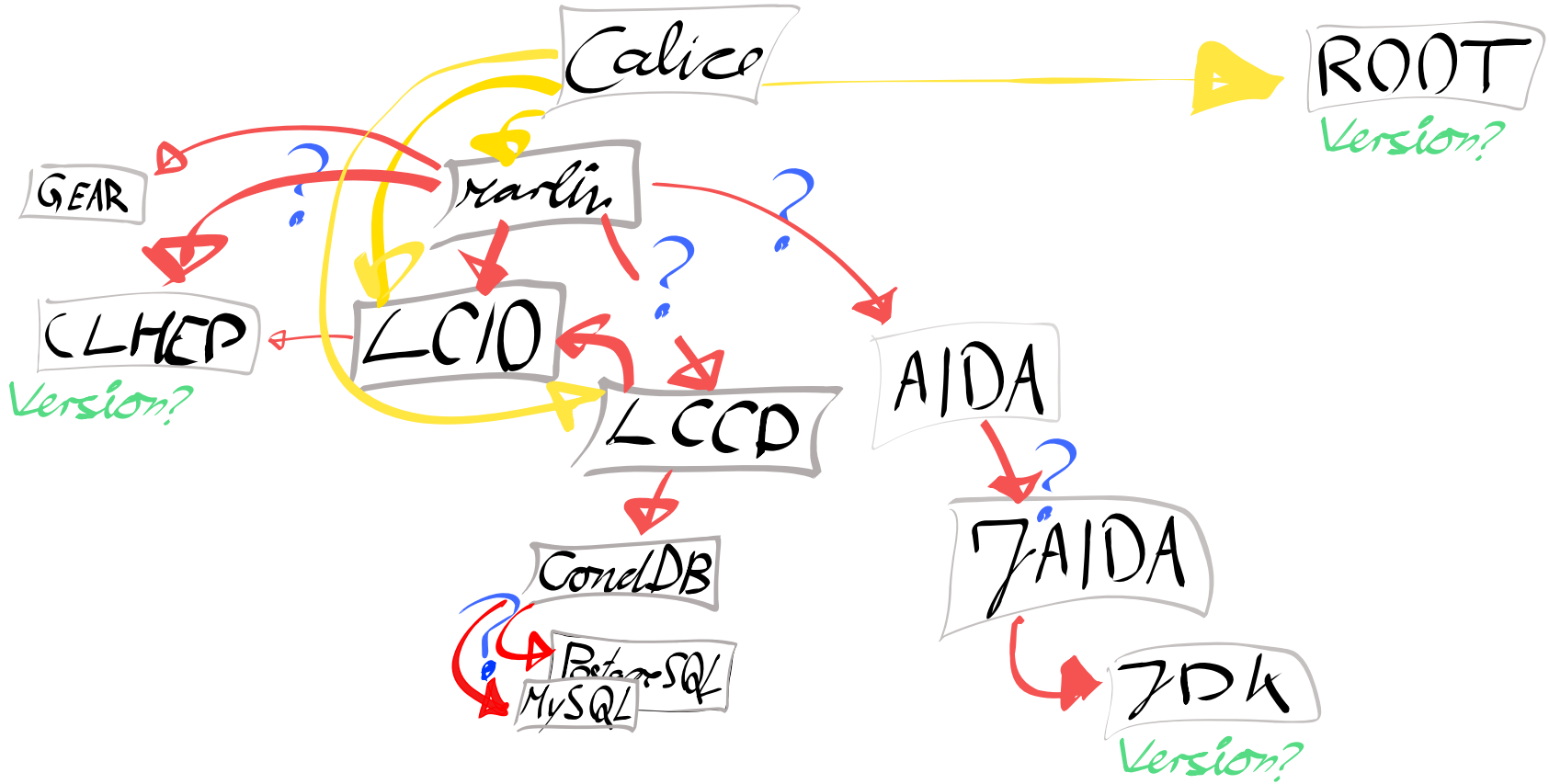
Dependencies of The Libraries

Example: The LCIO based Calice software:



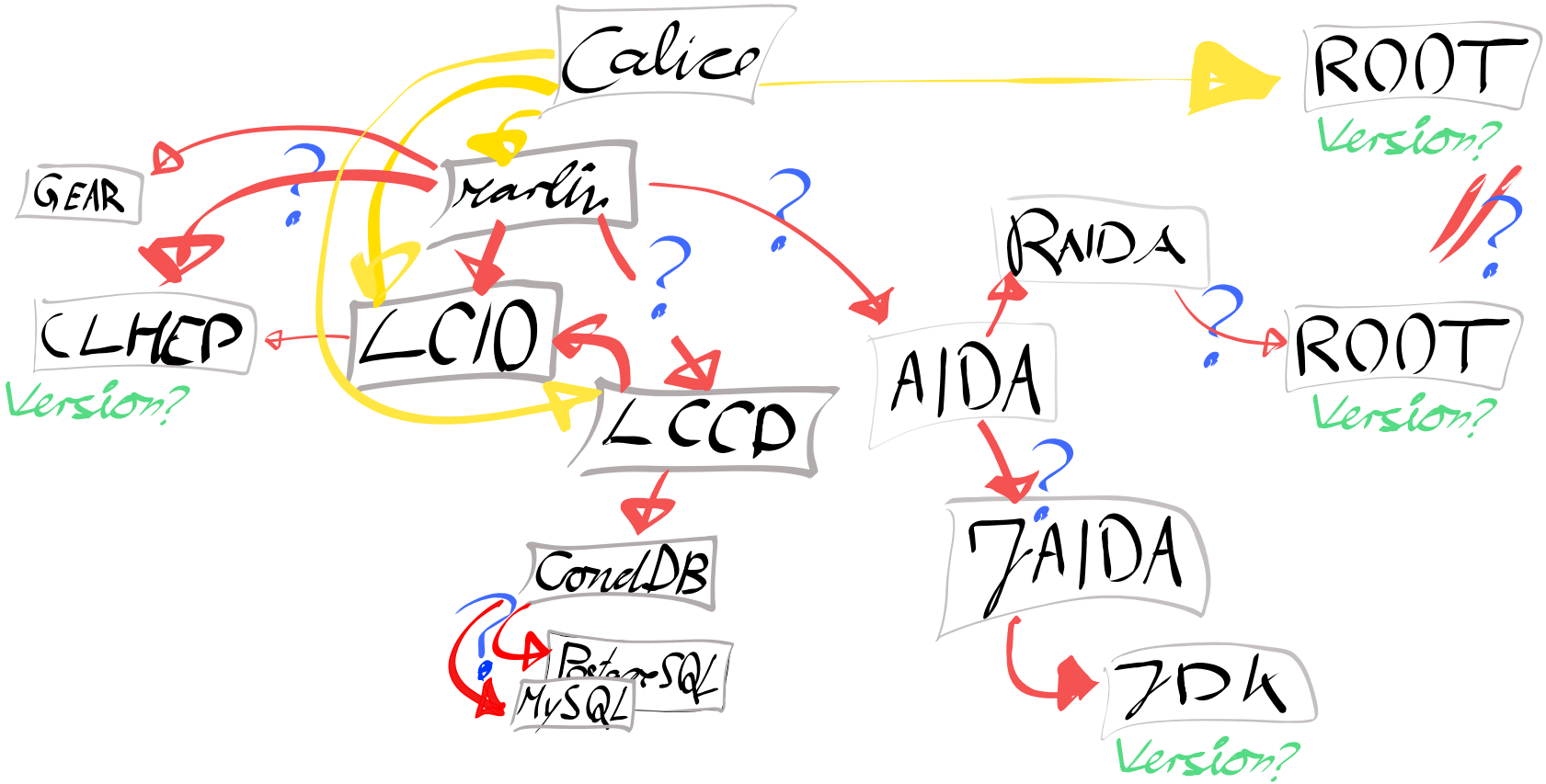
Dependencies of The Libraries

Example: The LCIO based Calice software:



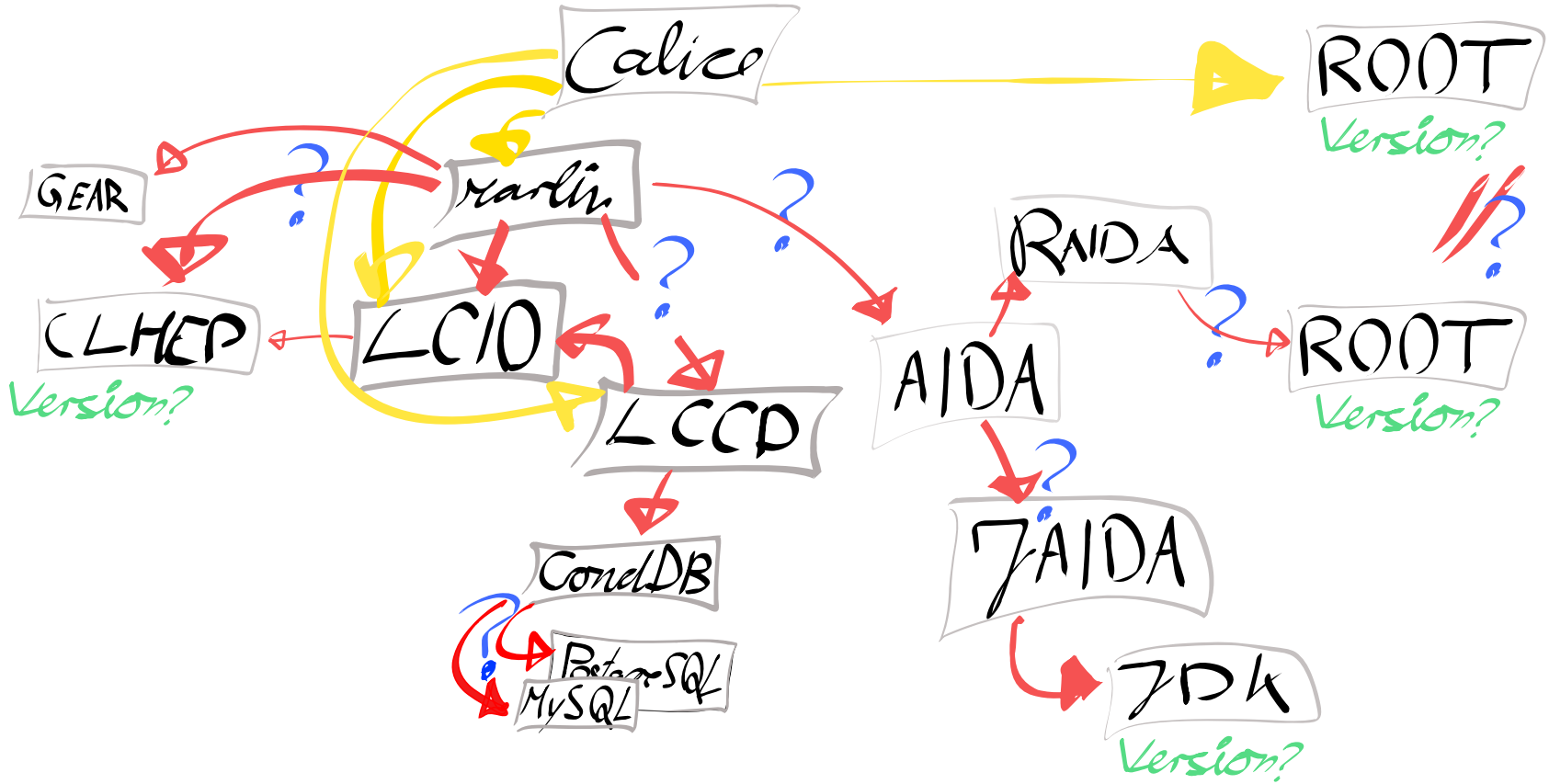
Dependencies of The Libraries

Example: The LCIO based Calice software:



Dependencies of The Libraries

Example: The LCIO based Calice software:



How to find out which libraries need to be linked ?

Bookkeeping of Library Dependencies – pkg-config

Solution: each library provides a file which describes preprocessor/linker flags and dependencies:

```
prefix=/ILC-soft
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include

Name: Marlin
Description: Modular Analysis and Reconstruction ...
Requires: lcio lccd jaida
Version: 0.9.5
Libs: -L${libdir}  \
 -Wl,-whole-archive,-lMarlin,-no-whole-archive
Cflags: -I${includedir}/marlin -DUSE_LCCD
```

Compiling/Linking using pkg-config

```
g++ -c MyMarlin.cc `pkg-config --cflags Marlin` \
   -o MyMarlin.o

g++ MyMarlin.o `pkg-config --libs Marlin`
```

pkg-config resolves dependencies. It returns all necessary preprocessor and linker flags.

- Do we need portability ?
  Or, do we have a monoculture ?
- Is the current way of building standardised enough ?
- Or should we adopt more automatised solutions ?

- Do we need portability ?
  Or, do we have a monoculture ?
- Is the current way of building standardised enough ?
- Or should we adopt more automatised solutions ?

- Do we need portability ?
  Or, do we have a monoculture ?
- Is the current way of building standardised enough ?
- Or should we adopt more automatised solutions ?

- Do we need portability ?
  Or, do we have a monoculture ?
- Is the current way of building standardised enough ?
- Or should we adopt more automatised solutions ?

GNU autotools would provide a standardised, portable build system.

GNU Autotools

- Automatic configuration:using results of feature tests :
  (tests use the compiler/linker)
  - OS and CPU,
  - standard conformance and features of compiler,
  - availability of functions/libraries,
  - function interfaces,
  - ...

  $\longrightarrow$ Portability.
- Portable generation of shared/static libraries.
- Standardised Makefiles.
  - Work with every implementation of make.
  - Scale from simple to complex projects.
  - Can install/uninstall, produce distributable source packages.
  - Track changes of source/headers and compile accordingly.
  - Include/exclude sources in/from compilation according to test results.

GNU Autotools

- Automatic configuration:using results of feature tests :
  (tests use the compiler/linker)
  - OS and CPU,
  - standard conformance and features of compiler,
  - availability of functions/libraries,
  - function interfaces,
  - ...

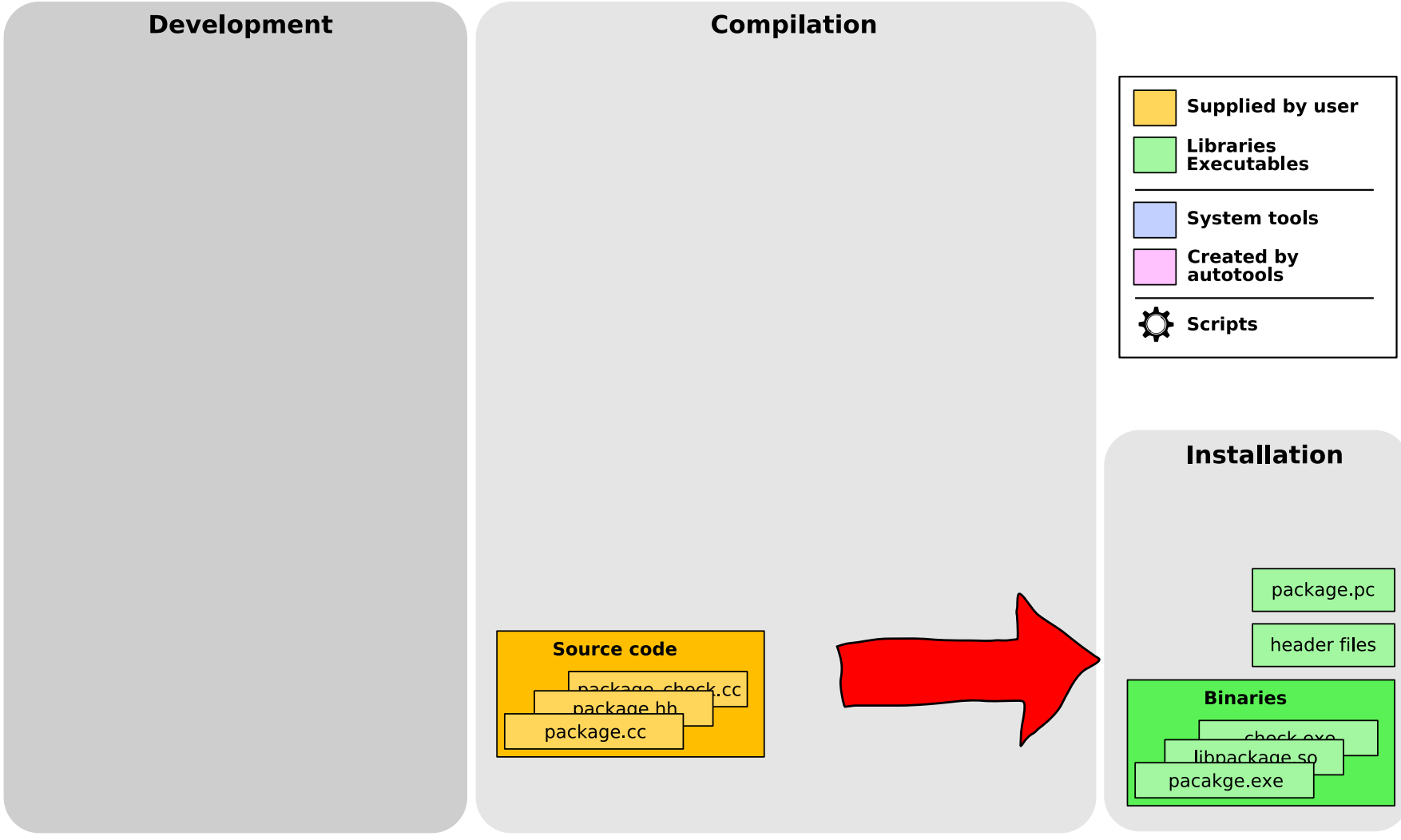
  $\longrightarrow$ Portability.
- Portable generation of shared/static libraries.
- Standardised Makefiles.
  - Work with every implementation of make.
  - Scale from simple to complex projects.
  - Can install/uninstall, produce distributable source packages.
  - Track changes of source/headers and compile accordingly.
  - Include/exclude sources in/from compilation according to test results.

GNU Autotools

- Automatic configuration:using results of feature tests :
  (tests use the compiler/linker)
  - OS and CPU,
  - standard conformance and features of compiler,
  - availability of functions/libraries,
  - function interfaces,
  - ...

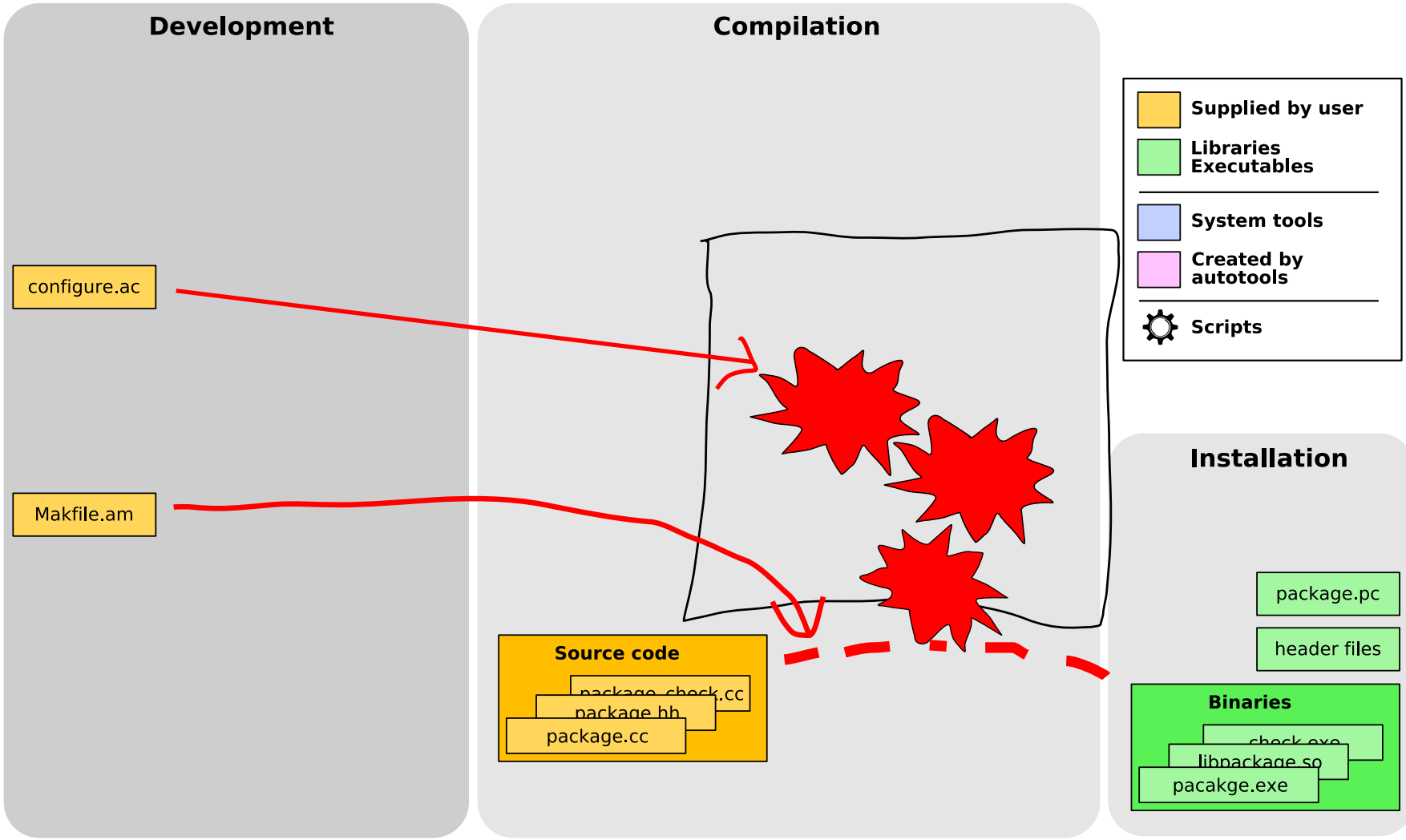
  $\longrightarrow$ Portability.
- Portable generation of shared/static libraries.
- Standardised Makefiles.
  - Work with every implementation of make.
  - Scale from simple to complex projects.
  - Can install/uninstall, produce distributable source packages.
  - Track changes of source/headers and compile accordingly.
  - Include/exclude sources in/from compilation according to test results.

The GNU Autotools



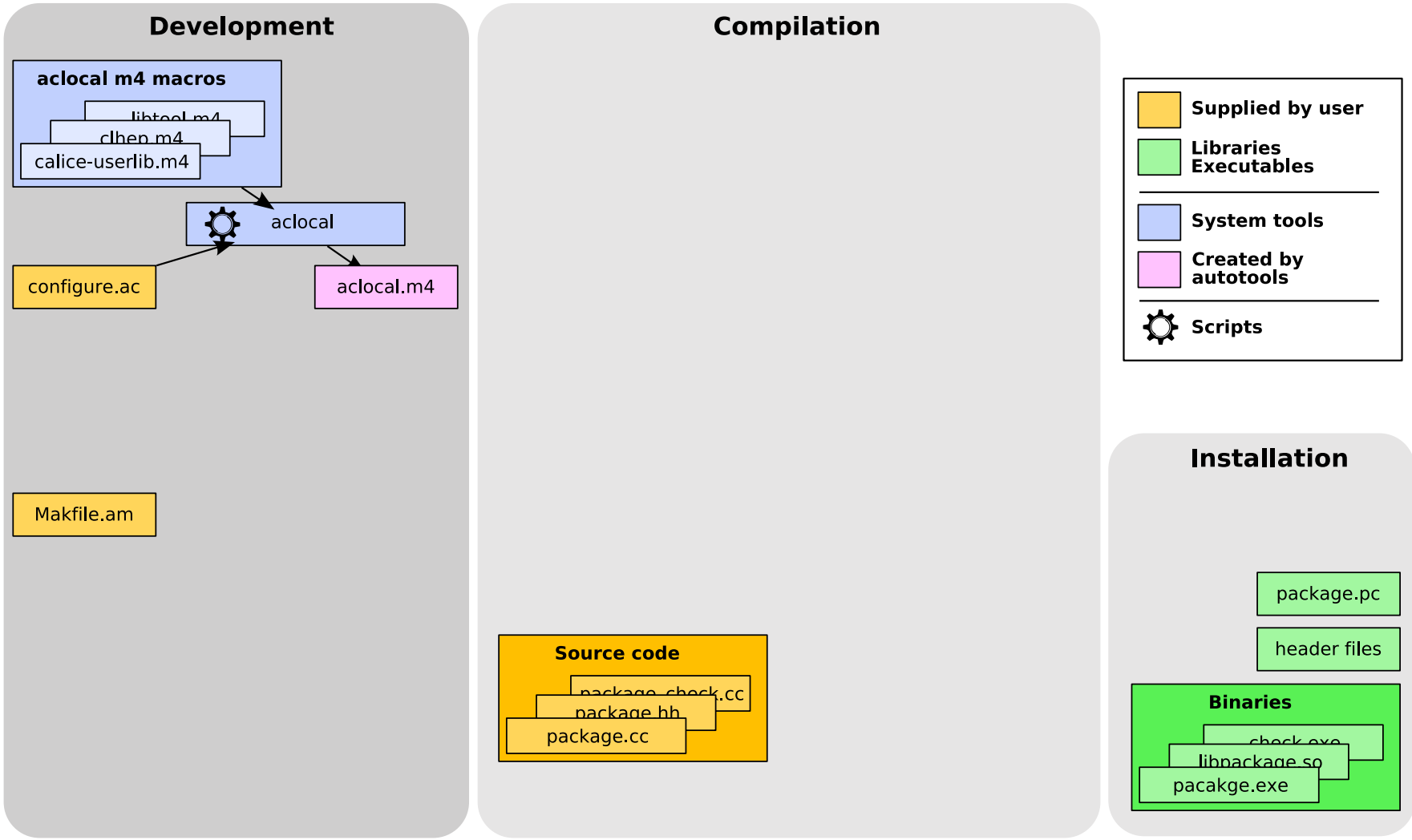
Goal: source $\longrightarrow$ libraries, executable, pkg-config file.

The GNU Autotools



Developer has to provide files which describe the configuration and building.

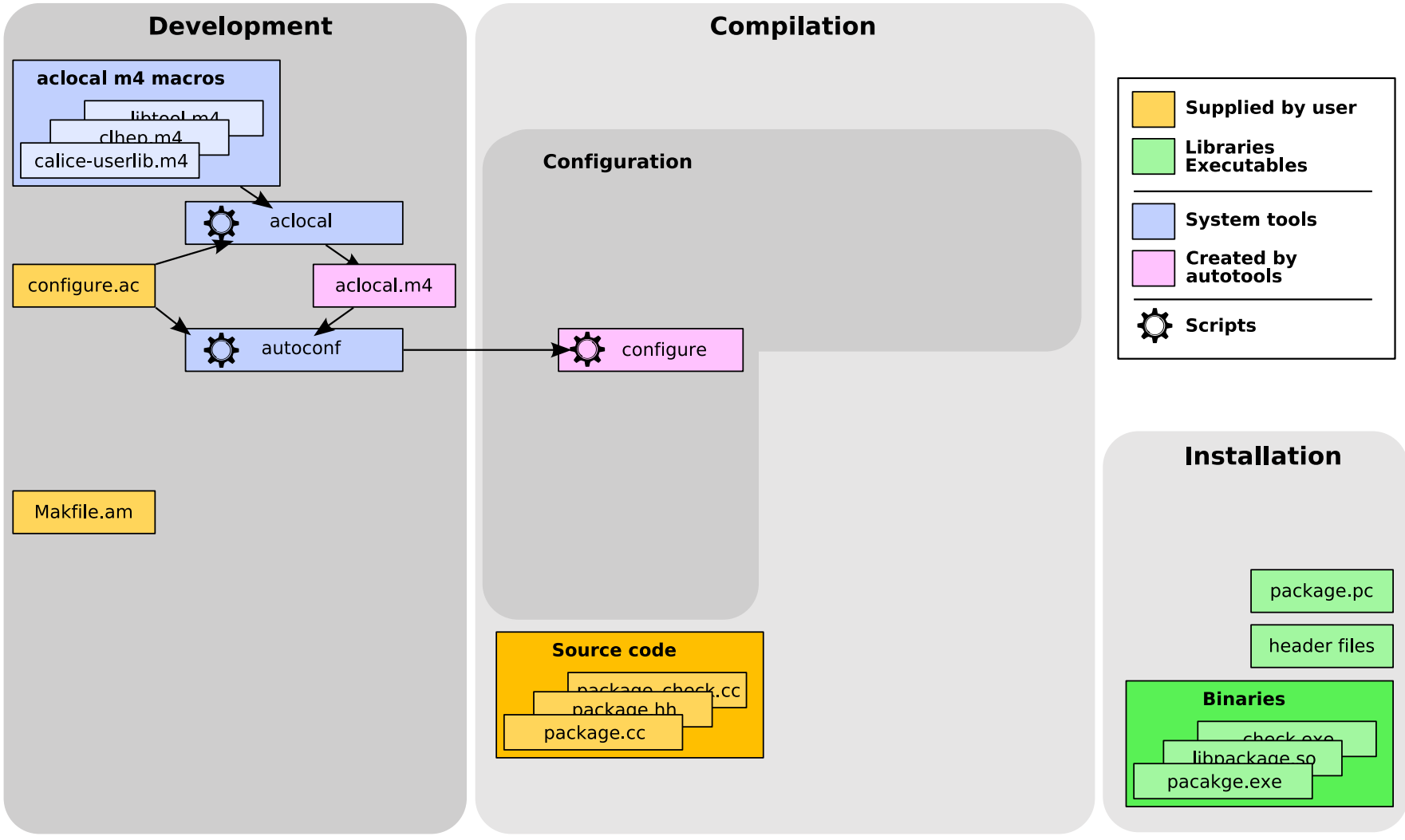
The GNU Autotools



Configuration described with high level macros.

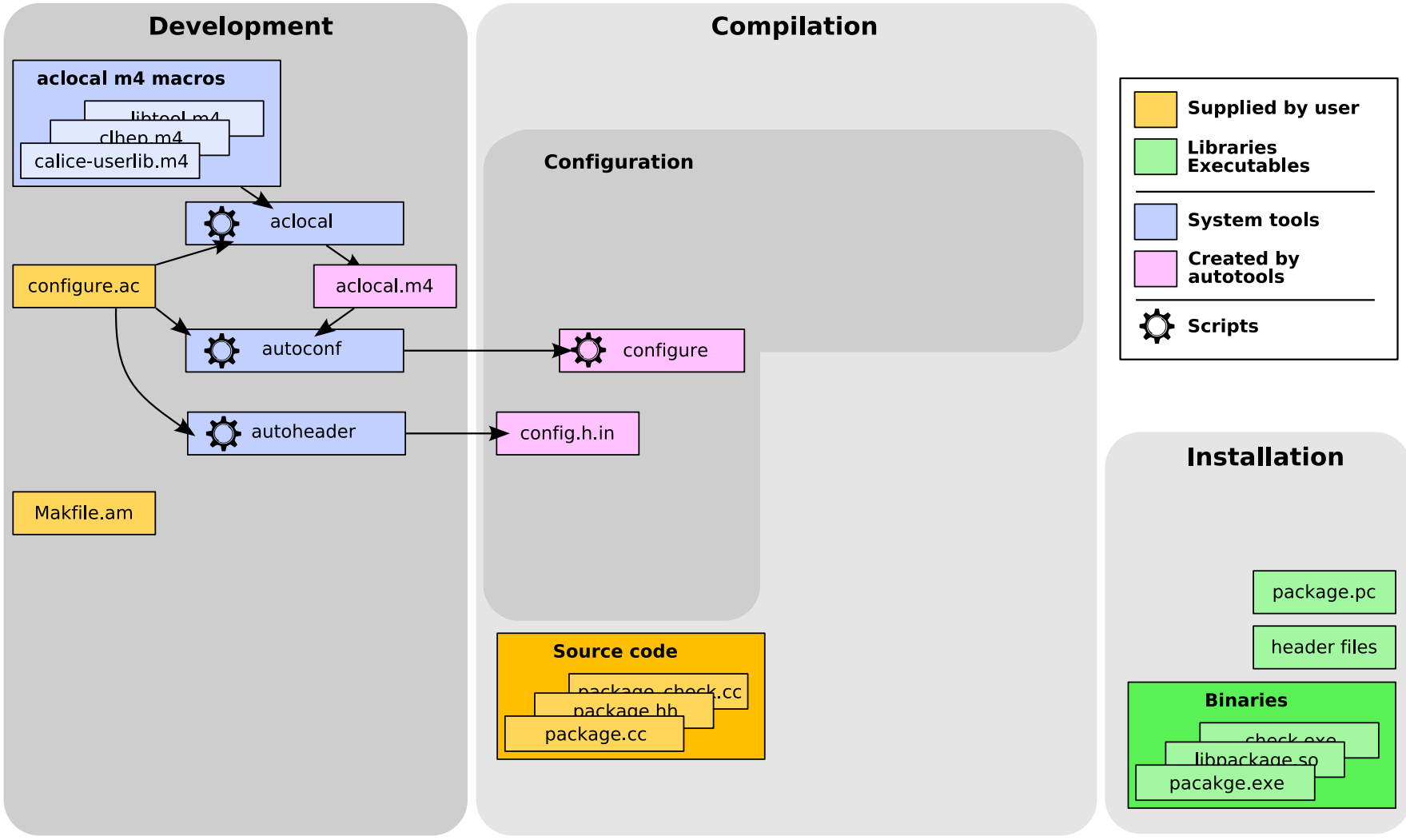
First step, collect definitions of all needed macros.

The GNU Autotools



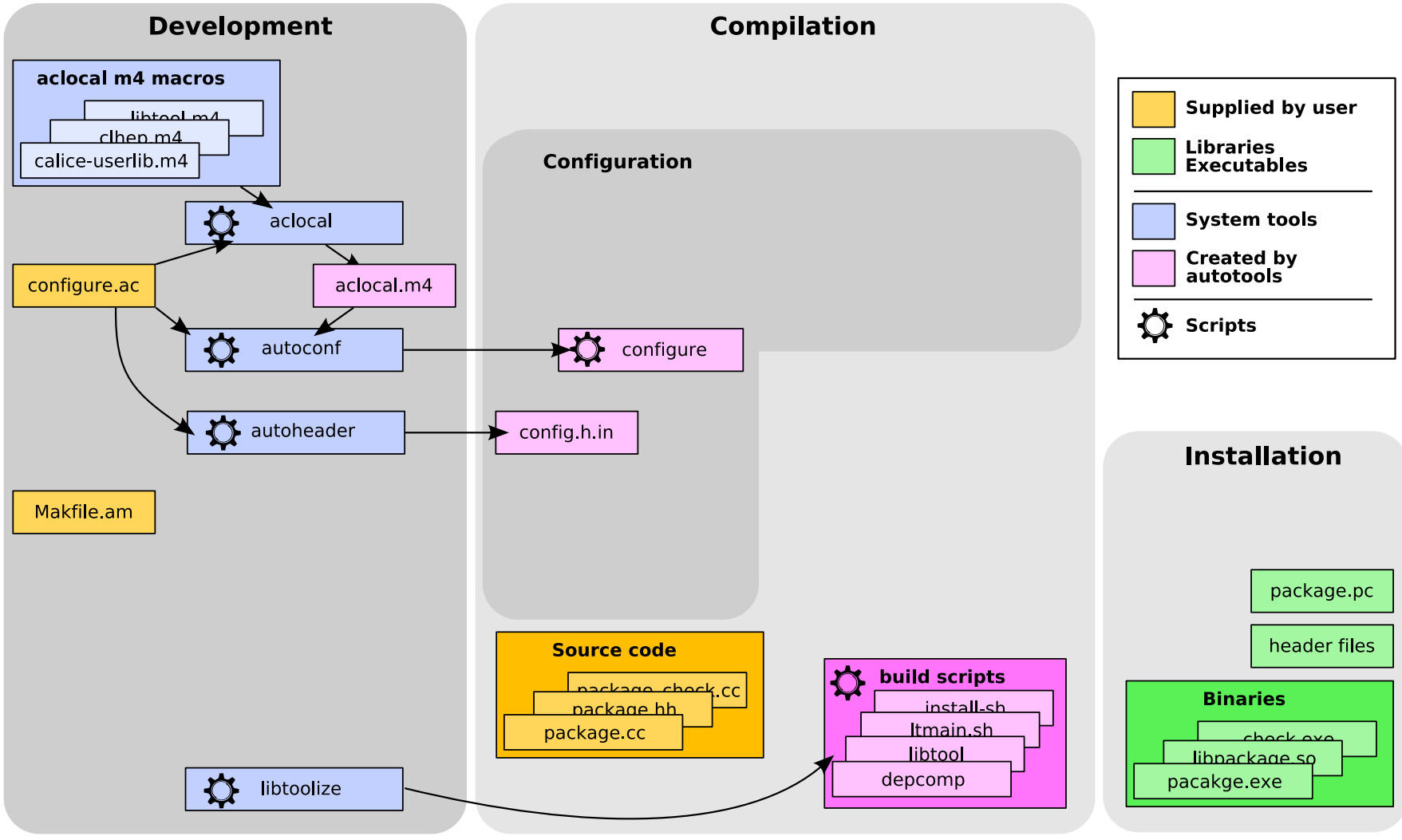
Then, macros are expand, the `configure` script is created.

The GNU Autotools



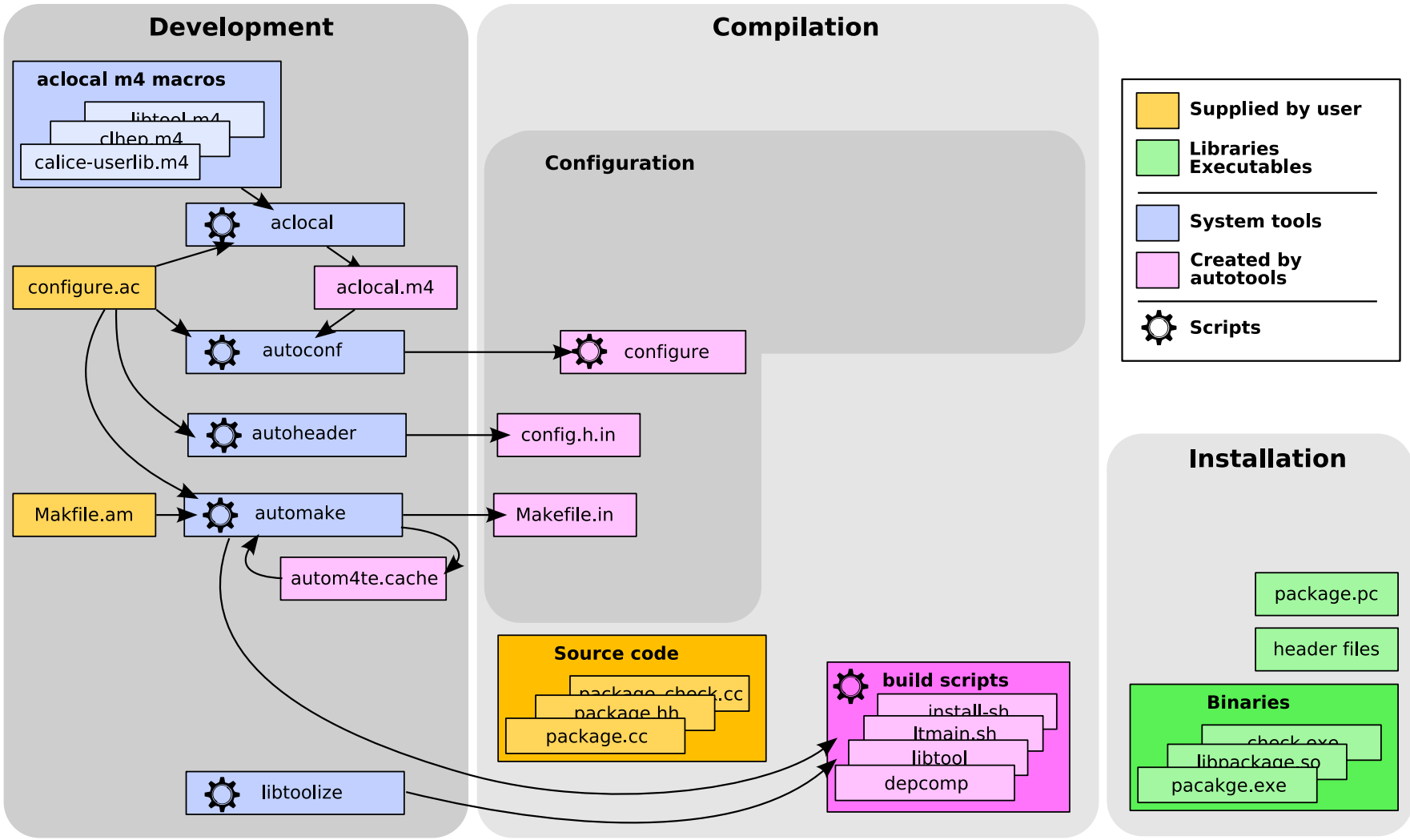
`configure.ac` also defines possible options for the build process (preprocessor definitions).

The GNU Autotools



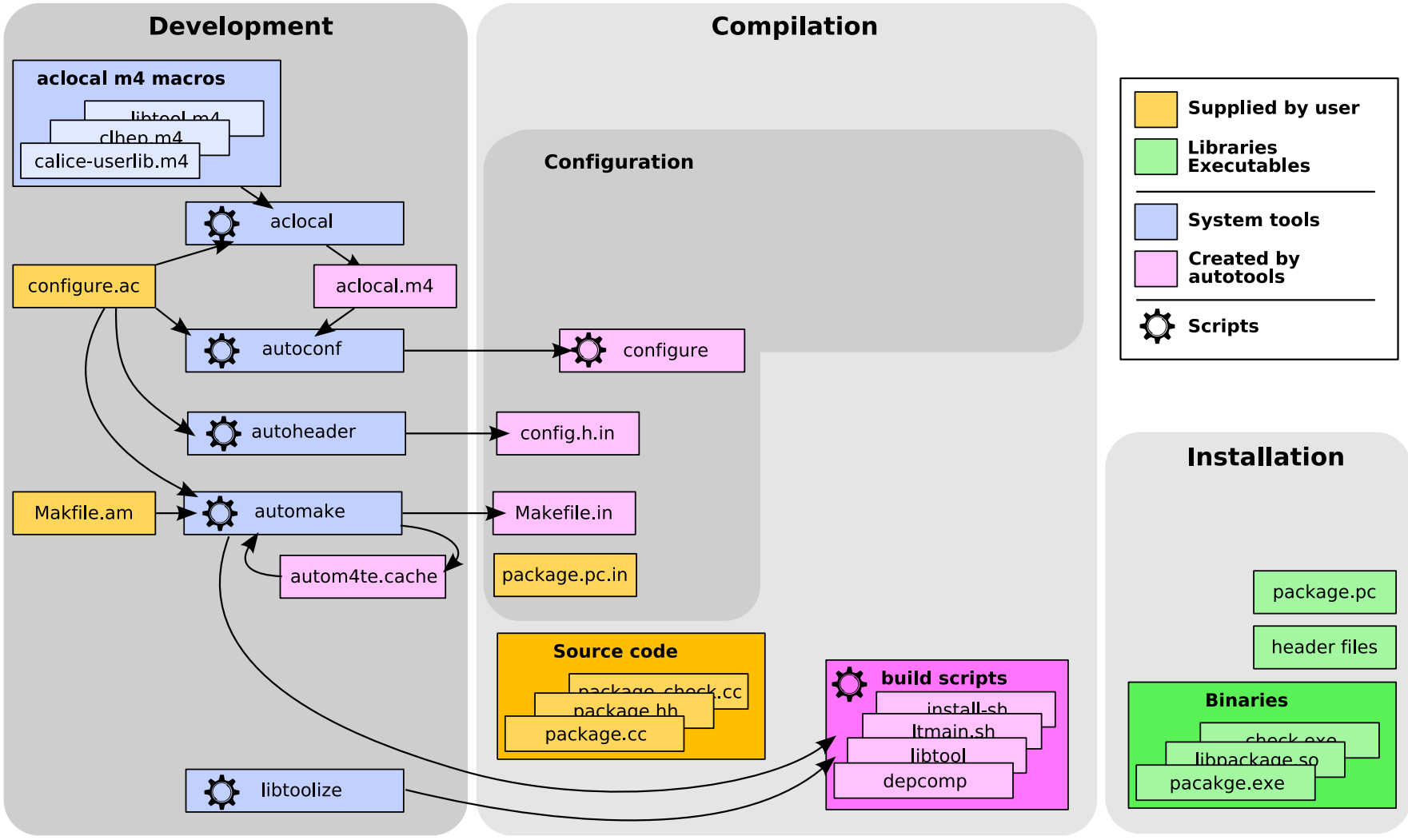
Autotools use scripts to build shared/static libraries → portability.

The GNU Autotools



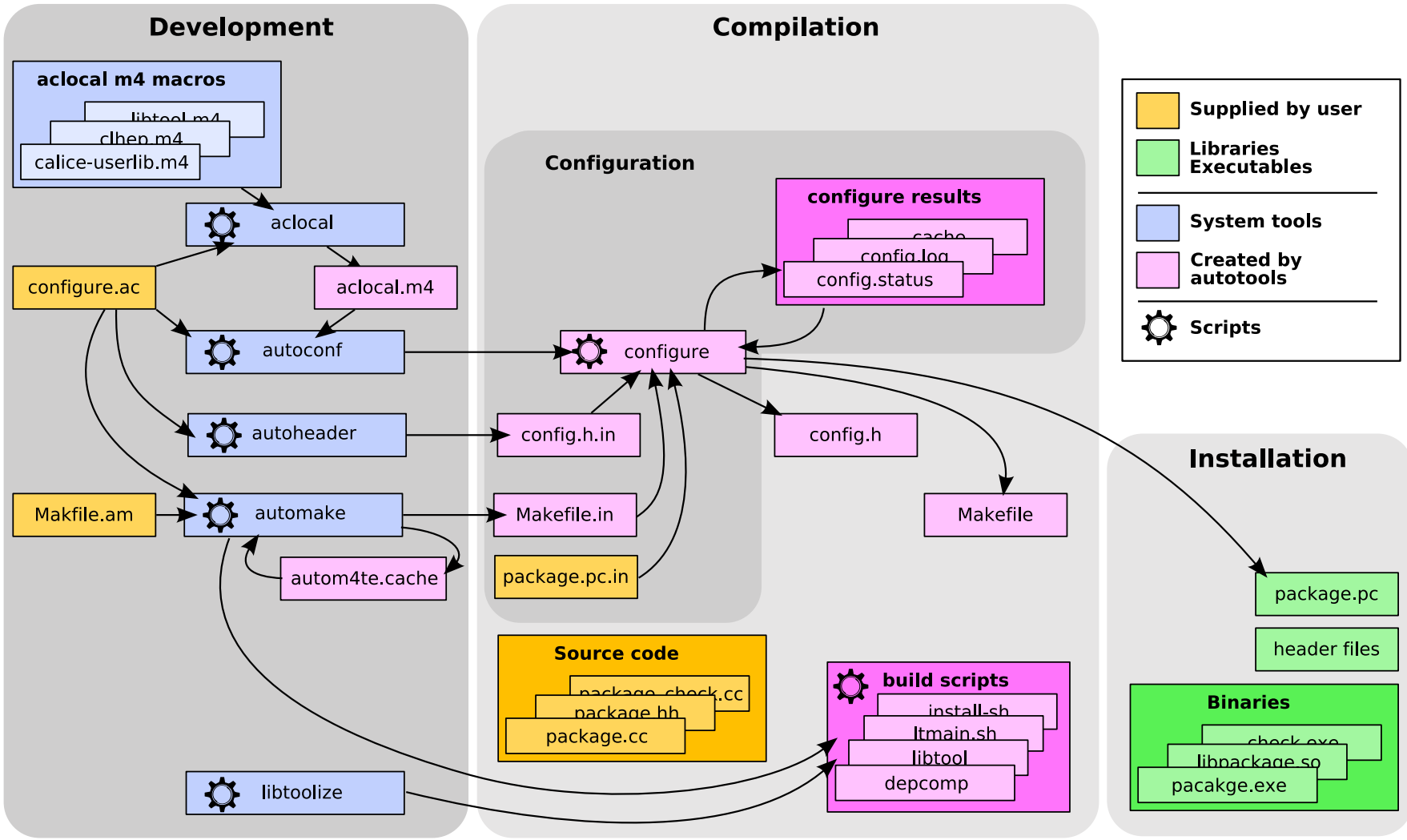
Makefile templates with standard build targets are create from `Makefile.am` files.

The GNU Autotools



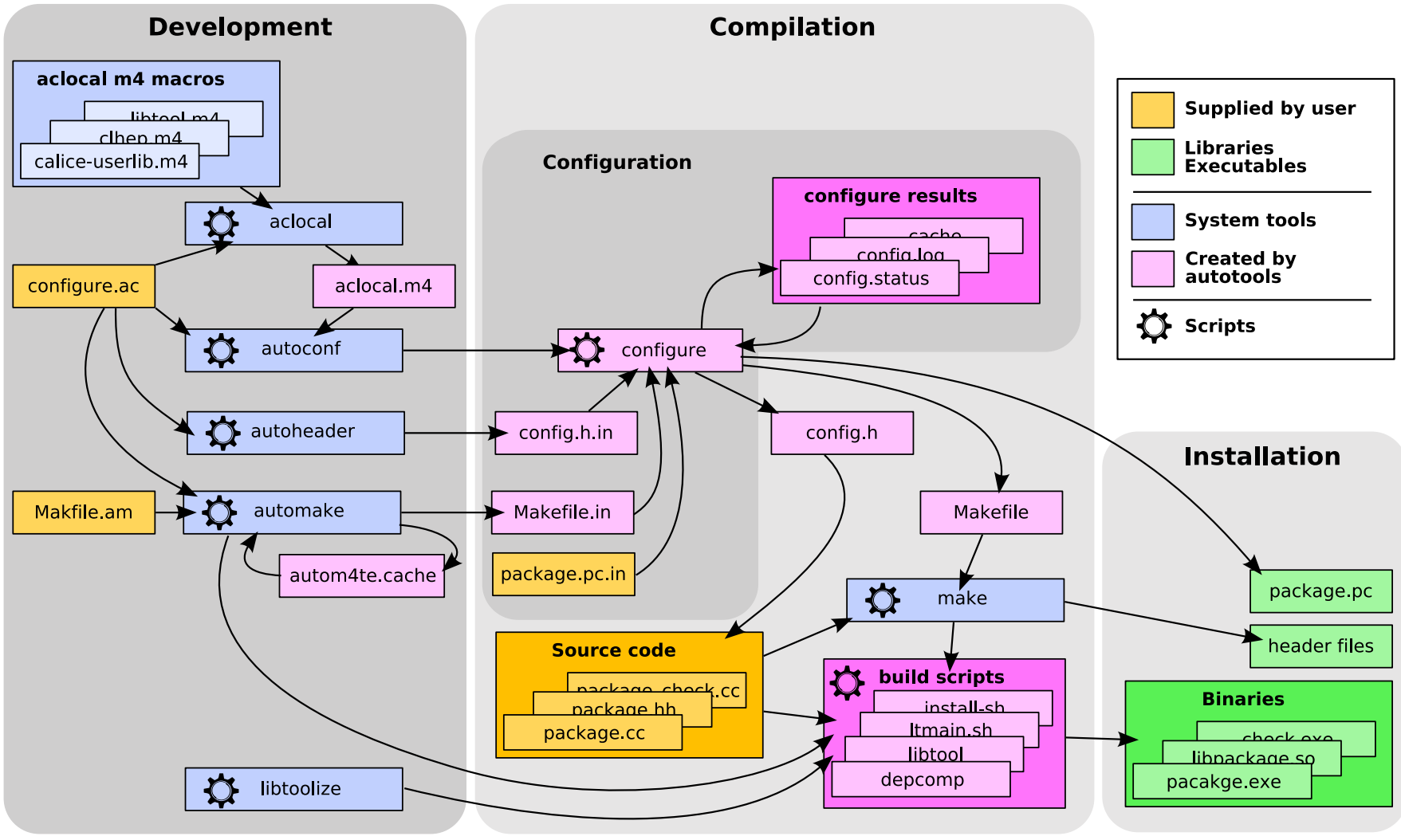
Developer has to provider template for pkg-config file. Will be filled by `configure`.

The GNU Autotools



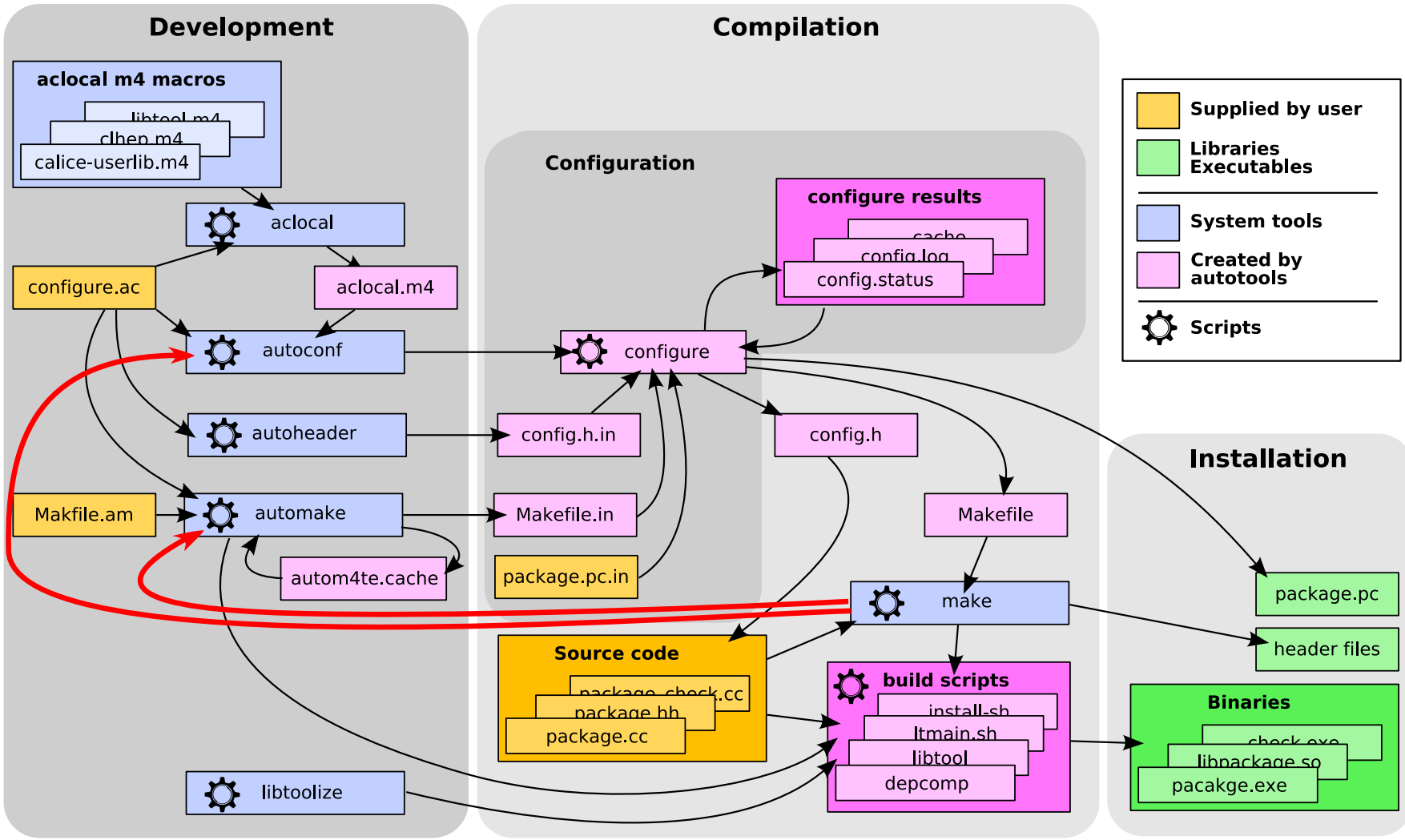
The configure script creates Makefiles, etc. from templates.
It fills in parameters specific to target system.

The GNU Autotools



Finally, make is used to compile, install etc.

The GNU Autotools



Once the Makefiles are created:

make will call autoconf, automake whenever necessary.

The GNU Autotools – From User Perspective

To build a package on an arbitrary host:

```
tar -xzf package-1.0.tar.gz
./configure
make
make install
```

- Fetch package, unpack it.
- Configure package for target system,
- compile and
- install the package in default location (/usr/local).

The GNU Autotools – From User Perspective

To build a package on an arbitrary host:

```
tar -xzf package-1.0.tar.gz
./configure
make
make install
```

- Fetch package, unpack it.
- Configure package for target system,
- compile and
- install the package in default location (/usr/local).

The GNU Autotools – From Developer Perspective

To participate in development process (or if checked out from CVS), need to bootstrap first:
(i.e. create configure script, Makefile templates etc.)

```
sh ./autogen.sh
```

Which runs:

```
libtoolize
aclocal
autoconf
autoheader
automake
```

Then only `configure` and `make` need to be called directly.

The GNU Autotools – From Developer Perspective

To participate in development process (or if checked out from CVS), need to bootstrap first:
(i.e. create configure script, Makefile templates etc.)

```
sh ./autogen.sh
```

Which runs:

```
libtoolize
aclocal
autoconf
autoheader
automake
```

Then only `configure` and `make` need to be called directly.

The GNU Autotools – From Developer Perspective

To participate in development process (or if checked out from CVS), need to bootstrap first:
(i.e. create configure script, Makefile templates etc.)

```
sh ./autogen.sh
```

Which runs:

```
libtoolize
aclocal
autoconf
autoheader
automake
```

Then only `configure` and `make` need to be called directly.

The autoconf input file - configure.ac

Tests are activated by high level macros mixed with korn shell scripts:

```
...

AC_CXX_NAMESPACES(, [config_error=yes])

if test x$config_error = xyes; then
  AC_MSG_ERROR([Need compiler with namespace...])
fi

AC_CXX_HAVE_STL([have_stl=yes],[have_stl=no])
AM_CONDITIONAL(USE_STL, test x$have_stl = xyes)

...
```

Automake – Makefile,am

Simple Makefile.am to create shared/static libraries:
`libpackage.so/.a` and `libpackage_ext.so/.a`

```
nobase_pkginclude_HEADERS=pacakge.hh

lib_LTLIBRARIES = libpackage.la \
                  libpackage_ext.la

libpackage_la_SOURCES = package_0.cc

if USE_STL
  libpackage_la_SOURCES += package_1.cc
endif

libpackage_ext_la_SOURCES = package_ext.cc
```

Automake – Makefile,am

to create executables:

```
...

bin_PROGRAMS = exe

exe_LDFLAGS = libpackage.la
exe_SOURCES =  main.cc

...
```

or process a subdirectory:

```
...

SUBDIRS=auxlib

...
```

Automake – Makefile.am

or to build unit tests:
(programs which are not installed to test e.g. libraries):

```
...

check_PROGRAMS = test_pacakge
test_package_LDFLAGS = libpackage.la
test_package_SOURCES = test_paclage.cc

...
```

Looks complicated - Why should I use this crap ?

Steep learning curve – But once learned:

- Allow to create portable projects.
- Projects are easily extendable.and maintainable.
- Standardised Makefiles.

Looks complicated - Why should I use this crap ?

Steep learning curve – But once learned:

- Allow to create portable projects.
- Projects are easily extendable.and maintainable.
- Standardised Makefiles.

Aren't there less complicated alternatives ?

Alternatives, yes. Replacements for make and autotools:

- ant/maven – java based.
  No out of the box C, C++, F77 support.
- SCons – python based.
  Already mature ? Maintainable ?.
- CMake.
  ?

But: autotools widely adopted standard:

xorg, GNOME, MySQL, CLHEP, ...

$\longrightarrow$ good support.

Caveats of automatised solutions

Automatised solutions do a poor job if libraries etc. are not installed in default locations.

Way out:

- Provide pkg-config files and collect them in one location.
- Install libraries, header files in standard locations. For example:

Caveats of automatised solutions

Automatised solutions do a poor job if libraries etc. are not installed in default locations.

Way out:

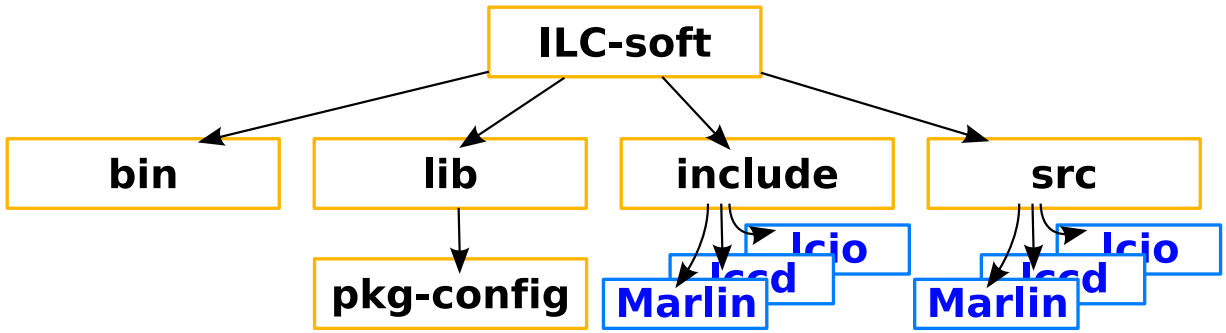
- Provide pkg-config files and collect them in one location.
- Install libraries, header files in standard locations. For example:

Caveats of automatised solutions

Automatised solutions do a poor job if libraries etc. are not installed in default locations.

Way out:

- Provide pkg-config files and collect them in one location.
- Install libraries, header files in standard locations. For example:



Summary

- I would suggest to provide pkg-config files for all libraries.
  (solves library dependency problem.)
- I would encourage the usage of the GNU autotools.
  (Results in portable, maintainable, standardised projects.)
- I think a UNIX like organisation of ILC-soft libraries and header files would be a good idea.
  (Easier to find libraries, header files, less configuration effort.)
- There are many packages which depend on each other. Do we need package managment?
  (e.g. apt-get install MarlinReco or emerge MarlinReco)

Summary

- I would suggest to provide pkg-config files for all libraries.
  (solves library dependency problem.)
- I would encourage the usage of the GNU autotools.
  (Results in portable, maintainable, standardised projects.)
- I think a UNIX like organisation of ILC-soft libraries and header files would be a good idea.
  (Easier to find libraries, header files, less configuration effort.)
- There are many packages which depend on each other. Do we need package managment?
  (e.g. apt-get install MarlinReco or emerge MarlinReco)

Summary

- I would suggest to provide pkg-config files for all libraries.
  (solves library dependency problem.)
- I would encourage the usage of the GNU autotools.
  (Results in portable, maintainable, standardised projects.)
- I think a UNIX like organisation of ILC-soft libraries and header files would be a good idea.
  (Easier to find libraries, header files, less configuration effort.)
- There are many packages which depend on each other. Do we need package managment?
  (e.g. apt-get install MarlinReco or emerge MarlinReco)

Summary

- I would suggest to provide pkg-config files for all libraries.
  (solves library dependency problem.)
- I would encourage the usage of the GNU autotools.
  (Results in portable, maintainable, standardised projects.)
- I think a UNIX like organisation of ILC-soft libraries and header files would be a good idea.
  (Easier to find libraries, header files, less configuration effort.)
- There are many packages which depend on each other. Do we need package managment?
  (e.g. apt-get install MarlinReco or emerge MarlinReco)