7 SCIENTIFIC HIGHLIGHT OF THE MONTH: "Harnessing the power of modern package management tools for a large Fortran-90-based project: the mutation of ABINIT"

Harnessing the power of modern package management tools for a large Fortran-90-based project: the mutation of ABINIT

Y. Pouillon[1,2,3], X. Gonze[1]

[1] European Theoretical Spectroscopy Facility (ETSF),
Université Catholique de Louvain - Louvain-la-Neuve - Belgium

[2] Universidad del País Vasco (UPV/EHU) - Donostia-San Sebastián - Spain

[3] European Theoretical Spectroscopy Facility (ETSF) - Donostia-San Sebastián - Spain

Abstract

ABINIT is a Fortran 90 free software application that allows the atomic-scale simulation of properties of matter, thanks to Density Functional Theory and Many-Body Perturbation Theory. It is used by more than thousand individuals, who enjoy the wide spectrum of properties that ABINIT allows to compute easily. Several dozen developers contribute to ABINIT from different parts of the world. In 2004, it was perceived that a change of the paradigm for source and package management was needed, in order to benefit from standard package management tools. Thus started a noticeable mutation of ABINIT.

Although the restructuration of the Fortran 90 source directories was needed, the biggest clarification arose from understanding the different kinds of people linked to ABINIT, *i.e.* end-users, developers and maintainers, and the parts of the package they should have access to or control of. Previously, everyone was modifying the source and build system, while further advances required more specialisation in the community, e.g. the management of external libraries, in growing number, which has to be done by skilled maintainers. To address the issues raised by Fortran compilers, and because the ABINIT developers are mostly scientists, it was decided to provide support beyond the GNU Autotools (nowadays the paradigm for binary/package generation) by developing a new build system on top of it. While building ABINIT is now much simpler for end-users, we have had to deal carefully with the additional complexity encountered by developers and maintainers. We discuss the issues that appeared during the mutation.

All these efforts now guarantee further extensibility and maintainability of ABINIT, and have nicely improved its visibility in different communities, with the integration of the packages into the Debian, Gentoo and Ubuntu Linux distributions. Being generic and portable, the new build system might be used in the future by other projects as well.

7.1 Introduction

ABINIT is a feature-full software package for the atomic-scale simulation of molecules and materials, based on Density Functional Theory (DFT) and Many-Body Perturbation Theory (MBPT). Though it is tuned to perform best on periodic systems like crystals, ABINIT is nevertheless able to deal with surfaces and molecules. Not only does it provide valuable information on the geometries of all these systems, it gives access to their electronic, dynamical and dielectric properties as well. Providing 16 tutorials and a lot of help, its companion website, `http://www.abinit.org/`, lets the newcomer step-in smoothly and discover progressively all of its features.

Started in 1997, on the basis of a legacy code, ABINIT has been being written by scientists for scientists, and delivered under the terms of the GNU General Public License (GPL) since 2000. Enjoying the freedom, openness and conviviality found within the project, several dozen developers have joined the community, bringing unexpected contributions as well. ABINIT counts now more than 1000 registered users and an average of 45 regular contributors from all around the world. In addition to its 500,000 lines of Fortran 90 contained in a thousand files, ABINIT features also about 600 automatic tests. For details about the scientific use of ABINIT, we refer the reader to Refs. [1, 2].

From the very start of the ABINIT project, many software engineering concepts were kept in mind, especially those related to portability, self-testing and self-documentation. As an example, every source file provides a header describing the purpose of the routine it contains, as well as its arguments, and other useful information, that may be processed by RoboDOC[1] to generate the HTML documentation for the source code[2]. Yet, despite the rigour that has been the basis of its development, ABINIT has not been originally designed to handle properly extensive code reuse. Its structure was quite intermixed and no much care was taken about packaging standards. Beyond the quite easily handled BLAS/LAPACK library for linear algebra operations, and the — already more difficult to address — MPI library for parallelism, the demand expanded a lot from 2004.

Indeed, that year, the NANOQUANTA Network of Excellence (NoE), consisting in 10 research teams sponsored by the European Union (EU) was launched. One major goal of NANOQUANTA was the integration of the software developed within the network into an unified distribution of which ABINIT is a major component. The interoperability with several other codes guarantees that many kinds of complex calculations can be smoothly performed. Such efforts continue and expand within the European Theoretical Spectroscopy Facility (ETSF)[3]. As a consequence, it was needed to link ABINIT with NetCDF[4] for architecture-independent data exchange, XMLF90 for CML input/output, and the Nanoquanta exchange-correlation library, a C-based library

[1] RoboDOC home page: `http://www.xs4all.nl/~rfsber/Robo/robodoc.html`.
[2] See `http://www.abinit.org/package/robodoc/masterindex.html`.
[3] Website of the ETSF: `http://etsf.eu/`
[4] NetCDF home page: `http://www.unidata.ucar.edu/software/netcdf/`

coming from the Octopus code [3], providing routines that any DFT-based code may benefit from. These have been joined recently by a NetCDF-based platform-independent data-exchange library named ETSF I/O [4,5]. The latter implements the file format we described in the October 2007 highlight of this newsletter (#83). ABINIT has also been chosen as a basis by another EU-sponsored endeavour, codename BigDFT[5], the purpose of which is to eliminate the bottlenecks limiting the applicability of DFT to "large systems", i.e. containing more than 10000 atoms. Last but not least, the possibility of quantum transport calculations through the use of Wannier90 [6] started not much later afterwards.

As a consequence, 2004 became a turning point. When the support for external libraries became somewhat problematic, it felt obvious that the growth would soon become unmanageable if no action were undertaken quickly. Although very portable, the home-made build system was rather monolithic, e.g. with two shell-scripts generating a primitive Makefile for each directory containing the Fortran 90 routines, and was only able to handle Fortran properly. Its limits with respect to external libraries were obvious for some compilers and architectures.

As most users are running ABINIT in Unix-like environments, and because of the free-software-oriented philosophy of the development model, it has been decided to bring the code as close as possible to the GNU Coding Standards. Thus started an extensive and in-depth mutation of the whole package which is now reaching its final steps. Between its 4.4 and 5.0 versions, ABINIT underwent a series of preparatory minor modifications: enforcement of the strict programming rules, already known within the ABINIT developer community as the "abirules"; large enhancements within the subroutine headers, in particular addition of intents for all arguments; decompression of the sources into a *abinit-<version>* directory, instead of the current directory; complete restructuring of the source directory tree; adoption of a decentralised Version Control System (VCS): Bazaar[6]. While these preliminary steps were quickly addressed by a few selected developers within a small amount of time, the complete rewriting of the build system that followed, nicknamed "breaking the monolith", required many more efforts and a much broader culture in software programming and management. The former "monolith" was broken into many pieces, improving the distinction between the three levels of contributors along: end-users, developers, and maintainers (see Fig. 1).

7.2 Developing software in scientific environments

Before fully entering into the details, and in order to facilitate the understanding of our approach, let start with an overview of the situation. The ABINIT developer community is mostly made of physicists and chemists. As such, the typical ABINIT contributors do not have a very extended culture in computer programming, which means that in most cases their development expertise is limited to Fortran 90. Furthermore, since most ABINIT developers are hired to carry out scientific research, they cannot devote too much time in taking a software engineer's point of view. Another essential point to account for is the lack of comfort accompanying High-

[5]See `http://www-drfmc.cea.fr/sp2m/L_Sim/BigDFT/index.html` for details.
[6]See `http://bazaar-vcs.org/` for details

Performance Computing (HPC) environments. Working in a HPC environment may indeed imply a lot of constraints: restricted access, the need of a custom MPI installation for parallelism, unavailability of recent versions for software like Python and the Autotools, necessity for very specific skills, to cite the most obvious ones. In some cases, security issues linked with the potential military applications of the results finish to make things very involved.

All this means that the build system has to take care of as many aspects as possible, far beyond what standard source management tools provide, and hide as much complexity as possible to the end-user. In particular, its interface has to be minimalistic and clear. One precision: by using "source management", we mean everything related to the build and distribution of the source code; other aspects, like version control, also play a major role in this respect, but will not be discussed here. Only maintainers should have to modify the build system, and that is why we have opted for a configuration-file-based approach for users and developers. Before everything else, we have defined minimum software requirements: Perl, Python, GNU Make or equivalent, GNU M4, GNU Autoconf, GNU Automake, and GNU Libtool, the three last ones forming the so-called "GNU Autotools". Perl and Python were chosen for performance and clarity, and they definitely constitute too large packages for us to provide any support for them. However, we were able to find a version combination of the Autotools which can be installed in a user's home directory, in case it would not be possible for them to have recent versions installed system-wide[7]. Alternative systems have been thought about as well, but they do not provide sufficient support for Fortran 90. The GNU Autotools are the only ones able to conciliate the growth of ABINIT with the preservation of its portability, both by the high number of their features and because of their ubiquity among the Free Software community.

7.3 The build system

The new build system of ABINIT constitutes a software layer above the Autotools. Let us first examine important characteristics of these.

One of the fundamental concepts of the Autotools has lead our way: the end-user does not have to care about the build system, in particular **it is not necessary to have the Autotools installed to build the source code**. A distributed source tree is fully autonomous and contains both the data and the code to be built out-of-the-box, by end-users, through the well-established '`configure; make; make install`' trilogy.

On the other hand, the developers and the maintainers have to install the Autotools on their development platforms, because the build system has to be kept synchronised with the source code, which means that some files (*e.g.* the *configure* script) have to be regenerated on a regular basis. To make an analogy between an Autotools-based build system and a scientific code, we can say that:

[7] One of us (Y.P.) wrote an installer script taking care of everything, and made the software bundle available on the ABINIT web site. See `http://www.abinit.org/gnu/` for details.

- the *configure.ac* is the source code of the main program;
- the M4 macros are the libraries (like BLAS, LAPACK, NetCDF, etc.);
- the *configure* script is the compiled code, Autoconf being the compiler;
- the *Makefile.am* files are the input data;
- the *Makefile.in* files are the preprocessed input data, Automake being the preprocessor;
- the *Makefile* files are the output data coming out from *configure*;
- the *make* program is a post-processor;
- distributing an Autotools-ready source package is like distributing a ready-to-use scientific code along with some example input data.

In order to facilitate future enhancements, improve modularity and simplify the build system, the physical and logical structures of the source directory tree have been first aligned, *i.e.* the directory tree has been split into 9 blocks (see Fig. 1), with corresponding global configuration files when necessary. In addition to these global files, one or two other local configuration files can be found in each of the core source libraries, as well as in some external libraries (see Fig. 2):

- *abinit.src* (A1), that lists the source files of a library, identifying Fortran modules, and controlling which files will be part of the libraries; this lets developers easily try and compare different ways to implement a feature;
- *abinit.amf* (A2), containing additions to *Makefile.am* files, such as explicit dependencies between object files, or lists of extra files to put in the source tarball; this is necessary when the build process cannot be fully automated.

Apart from the *configure.ac* file which has to be located in the top source directory, the whole build system is concentrated in one "pluggable" directory we have called *config/*. It contains the global configuration files mentioned above, as well as the scripts generating the files required to build ABINIT. These scripts take care of the pre-build stage shown in Fig. 2. Even if it cannot cover all situations, this design is both very flexible and extensible, and still usable by a relatively unexperienced developer, provided that the corresponding documentation is available.

Taking care of the build system itself has been attributed exclusively to the maintainers, in order to set the developers free from learning the internals of the Autotools and provide a better overall stability. As such, the developers of the code can be seen as the end-users of the build system, while the maintainers of the code are the developers of the build system. This also means that regular communication during the whole development process has become a key aspect, which was not the case in the version 4 of ABINIT. In other words, the developers now concentrate on the source code, while the maintainers provide source code management services.

Most of the issues we have to face when building ABINIT come from the Fortran compilers. In particular, their user interfaces are not-at-all standardised, and the compilation of Fortran modules produces the very annoying *.mod* files. These are binary, which means that, contrary to C header files, they are not platform-independent. Much worse, they are incompatible from one compiler to the other and may obey various conflicting naming conventions. Another painful issue is that a few compilers do not use the *'-I'* option to look for modules.

Needless to say, dealing nicely with such a situation is way beyond the average software-engineering skills of most scientists, and the Autotools do not provide any solution in this case. We have thus addressed these issues by providing compiler-vendor auto-detection, "tricks", and default optimisation flags for most Fortran compilers, at least those used within the ABINIT community. By "tricks", we mean workarounds one has to apply to have a Fortran compiler working properly, *e.g.* where to apply specific flags to obtain 64-bit objects.

Most build parameters are only machine-dependent, (*i.e.* not version-specific), and change quite rarely. Then it is very convenient for both users and developers to store them into config files. In order to have ABINIT built seamlessly on several machines sharing the same home directory — which is a relatively common situation in HPC environments, we have named these config files *$HOME/.abinit/build/<hostname>.ac*, i.e. the name of the machine plus an 'ac' extension to tell that the file is meant to be read by Autoconf's *configure* script.

One very delicate point is MPI support, both as it is critical for HPC and because there is no standard neither for its structure nor for its location. After a few failed attempts, we finally decided to push towards maximum flexibility, providing as many options as possible. Even if these options have to be set manually, it will typically be done every other year. It is possible either to let the build system auto-detect natively MPI-capable Fortran compilers or to use command-line parameters, e.g. specifying a prefix for the MPI installation. As time goes by, a more and more comprehensive database of examples is made available as well, saving developers' and users' time more and more efficiently.

7.4 Using the build system

End-users do not need know what the build system is, since they are not supposed to modify the sources. They may even ignore that it exists at all, as they will mostly download a tarball and build the code once in a while. For them, the mutation of ABINIT only means enhanced comfort. What they only have to know is that the build of ABINIT now follows the well-established '`configure; make; make install`' trilogy, and that they should store critical information into a config file. In most situations, once the latter has been set and proved to lead to reliable binaries, one may forget anything about the build parameters for quite a while. A fully documented template is provided with the source code of ABINIT to facilitate the setting of these parameters.

For developers, the new build system of ABINIT introduces the concept of a **pre-build stage**,

necessary to ensure a permanent consistency between the source tree itself and the metadata used to build it. The front-end script *makemake* performs all required steps that make the build of ABINIT possible (see Fig. 2):

1. Update the source tree according to the latest information available. Some routines are written by scripts and depend on the contents of other source files, like *e.g.* the routine checking the names of input variables. The most important and time-consuming part is dedicated to the parsing of all ABINIT source files and the subsequent generation of the Fortran interfaces for all the routines.

2. Update parts of the build system in order to follow accurately the evolution of the ABINIT source tree. This critical step consists in the writing of M4 macros by the build system itself. For instance, command-line options for the configure script are declared and a parser is written so that their validity and consistency can be checked at configure-time. Another example is the declaration of all the makefiles, which vary in number and location with time.

3. Generate prototype makefiles for Automake. Each of these files describes in a compact way how to build the contents of a directory. They highly depend on the auxiliary config files found in many source directories.

4. Run the Autotools in order to make the source tree autonomous and distributable. This step gathers all hand-written and script-written M4 macros into one file (running *aclocal*) and uses them in various places when creating the configure script (running *autoconf*). It also creates a C header input file (runing *autoheader*) that will store all preprocessing options later when processed by the *configure* script. Last but not least, it transforms the prototype makefiles into input data for the *configure* script (running *automake*).

Let's now have a look at the pre-configuration stage also described in Fig. 2. By having the ABINIT build system partly writing itself we suppress the need for developers to edit *configure.ac*, which is the most delicate part of any autotools-based build system. Moreover, having everything grouped under one command and performed systematically ensures that the source tree will always be in a consistent state. When developers add, move or delete a file or a directory, they have to update the corresponding metadata and run the *makemake* script. In case of a file, they mostly have to change one line in the *abinit.src* file (A1) found in the same directory; in the case of a new pre- or post-processing program, they have to edit the *config/specs/binaries.cf* (D1) file as well; for a directory containing a library, editing *config/specs/corelibs.cf* (D2) will be necessary. To facilitate the process, all these files are self-documented.

Providing the *configure.ac* script as well as a set of hand-made useful M4 macros is under the responsibility of the maintainers, who are the only ones who should edit the other global config files, as it will likely have consequences on the behaviour of the whole build system. Adding support for a new platform, compiler, or plug-in are also maintainer tasks, since such operations involve writing or modifying M4 macros, as well as editing the *configure.ac* script. Therefore, in contrast to what was happening until ABINIT 4, some communication is now necessary when

performing a certain number of actions, and this is precisely where the mutation of the code has involved a mutation of the community: roles have become specified more clearly and protocols have had to be defined.

7.5 Outcome

At present, several lessons have been drawn from the mutation. First, it is not yet possible to provide full auto-detection: the *<hostname>.ac* file is needed on most platforms, especially to deal with MPI. Now that there is a Fortran 90 compiler in the GNU Compiler Collection (GCC), the support of Fortran 90 by the GNU Autotools is greatly improving and the user interfaces of the Fortran compilers will hopefully tend to unify. This is however far from being the case for MPI, as system administrators are completely free to install it as they wish. Second, the jump from a home-made build system the developers were used to, towards a sophisticated build system based on the Autotools has generated some temporary frustration within the developer community. Indeed, the GNU Autotools generate automatically a large number of files defying human readability. During the transition, when the documentation on the new build system was still inexisting, most developers would try to solve their build-time problems by reading and hacking these files, which is both a nightmare for everybody and also the wrong way to proceed. Now, after a few years of practice, this issue has completely disappeared but took a long time to be eliminated. Third, while the usual developers could read, understand and modify the former build system according to their needs, a full handle of the new one is now only possible for maintainers. Though it is frustrating for some of the expert developers to lose the full understanding of the package they had previously, this is the price for the continued growth and complexification of the ABINIT package, as the Autotools are invisibly taking care of countless details. A good deal of the challenge is in the shift from independent individual contributions towards partnerships, and from control to mastery, with a particular emphasis on the critical importance of ***communicating before hacking***.

But one of the nicest results of this big mutation is that ABINIT has attracted the attention of several software packagers, who have also contributed in an interesting way to the improvement of its overall quality. ABINIT is now distributed with the Debian, Gentoo and Ubuntu Linux distributions, though no request or effort has been made on our side. This unexpected acknowledgment by the Free Software community is very encouraging and has a lot of nice side-effects. On one hand, the visibility and availability of the software are greatly improved, publicity is made for the related projects, and the user base expands faster. On the other hand, it puts an additional pressure on the developers and maintainers to produce higher-quality code and documentation. In any case, having standard packages available is a neat feature when it comes to deploy the software on a bunch of machines for schools or lectures.

Along with the 5.6 version of ABINIT, the new build system has reached a good level of quality and robustness. Its user interface is now frozen and most of the upcoming bug fixes and improvements will only appear as minor changes to end-users. We have successfully tested the automatic generation of Fortran 90 interfaces for all the subroutines and functions contained

in ABINIT, which was an extremely critical step. This has put a definitive end to the era of quick-and-very-dirty developments. In order to have the build system fitting the needs of both users and developers, we have allowed as many degrees of freedom as possible. As a bonus to our efforts, the set of Python scripts forming the new build system might be used by other intermixed C/C++/Fortran 90/Python software projects; actually the build systems of BigDFT and ETSF I/O have already imported some of our ideas.

References

[1] X. Gonze and *et al.* First-principle computation of material properties: the ABINIT software project. *Comput. Mater. Sci.*, 25:478–492, 2002.

[2] X. Gonze and *et al.* A brief introduction to the ABINIT software package. *Z. Kristallog.*, 220:558–562, 2005.

[3] A. Castro and *et al.* Octopus: a tool for the application of time-dependent density functional theory. *Phys. Stat. Sol. B*, 243:2465–2488, 2006.

[4] X. Gonze and *et al.* Specification of an extensible and portable file format for electronic structure and crystallographic data. *Comput. Mater. Sci.*, 43:1056–1065, 2008.

[5] D. Caliste, Y. Pouillon, M. J. Verstraete, V. Olevano, and X. Gonze. Sharing electronic structure and crystallographic data with ETSF_IO. *Comput. Phys. Comm.*, 179:748–758, 2008.

[6] A. A. Mostofi, J. R. Yates, Y.-S. Lee, I. Souza, D. Vanderbilt, and N. Marzari. Wannier90: A Tool for Obtaining Maximally-Localised Wannier Functions. *Comput. Phys. Comm.*, 178:685–699, 2008.

Acknowledgments

This work was funded by the European Union's Sixth Framework Programme through the Nanoquanta Network of Excellence (NMP4-CT-2004-500198). Additional support was provided by the FRFC project #2.4502.05 and the EU project NEST-2003-1 ADVENTURE 511815 (BigDFT). We would like to warmly thank T. Deutsch, M. Mikami, M. Torrent, M. Verstraete, P.-M. Anglade and D. Caliste (by order of appearance) for their precious help and feedback during the development and early tests of the new build system. We are also very grateful to M. Marques and M. Oliveira for their help during the implementation of the support for the LibXC. Technical support from the high-performance computing and mass-storage facility (CISM) in Louvain-la-Neuve was greatly appreciated as well, with very special thanks to J.-M. Beuken. We thank all the ABINIT community members for their patience, understanding, and cooperation all along this great mutation too, in particular D.R. Hamann, M. Côté, J. Zwanziger, and M. Giantomassi. Y. Pouillon is also grateful to L. Ferraro and Y. Leroy for very helpful and interesting discussions.

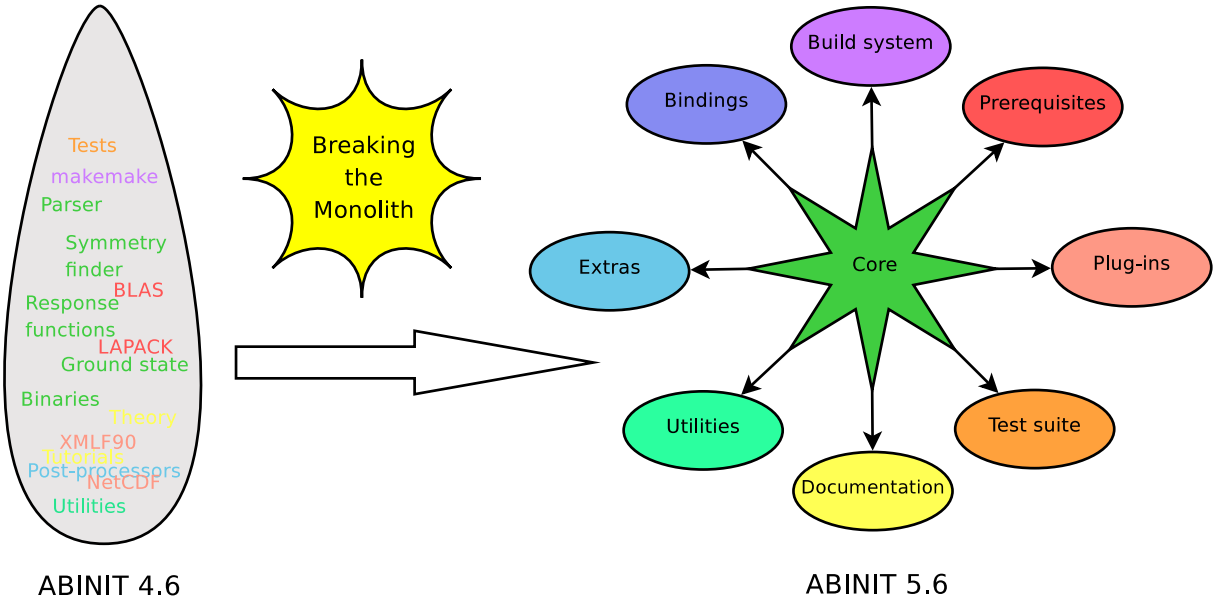

Figure 1: Overall view on the mutation of ABINIT, based on the restructuring of the code and the implementation of a new build system, providing better modularity and extensibility.

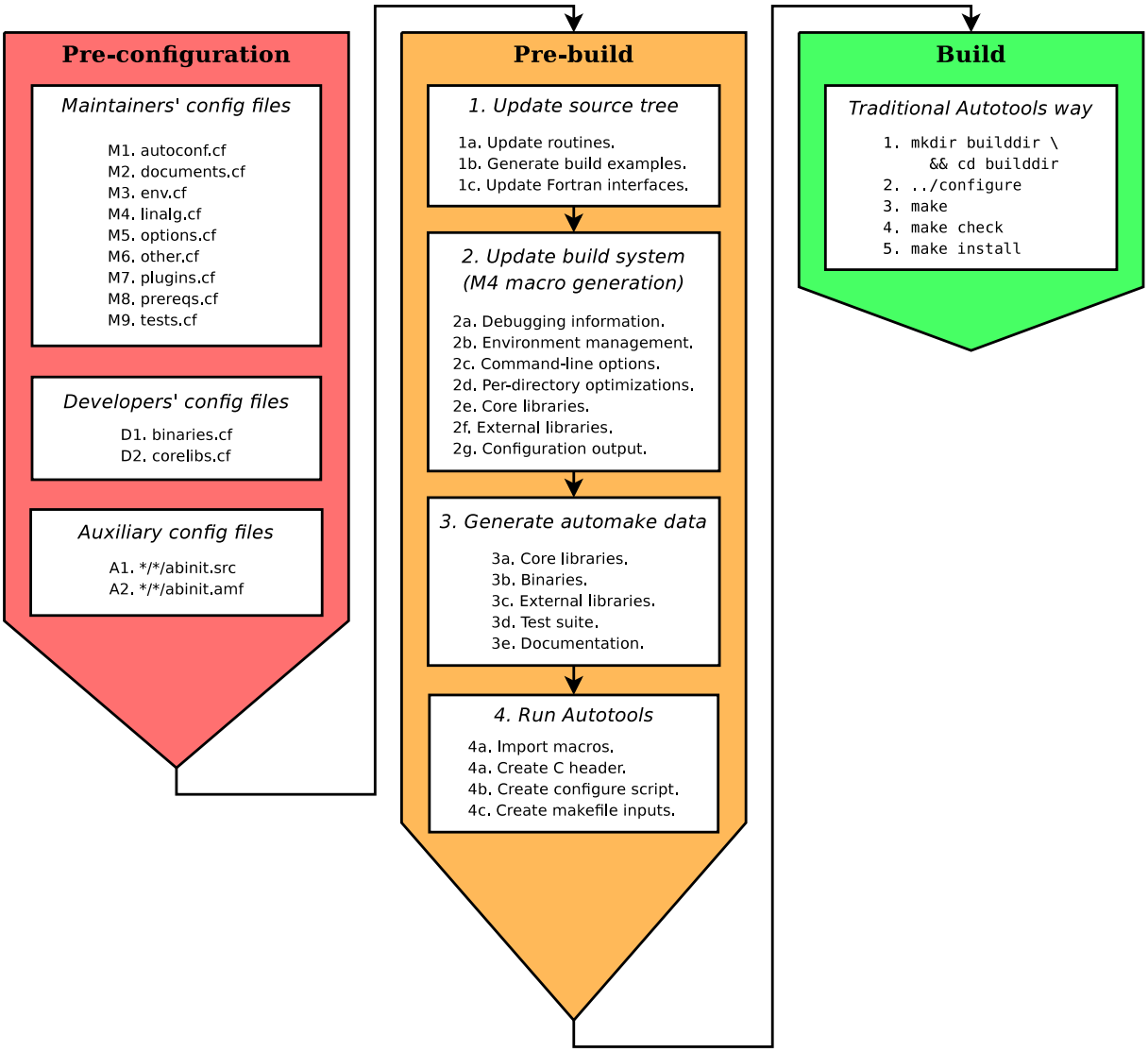

Figure 2: Steps involved in a build of ABINIT. As many files are generated automatically, the process requires different classes of contributors to operate at different steps and to co-operate with one another.