# Configure Considered Harmful

*Reginald H. Beardsley*

When the first configure script was written, the computing world was very different. There were a large number of vendors competing for market share in the Unix arena. Today most of the protaganists in the Unix wars are long gone. Even Sun Microsystems, which arguably spawned the genre is gone.

There are still a few proprietary Unix implementations left, but open source implementations, Linux and to a lesser degree, BSD, dominate the Unix landscape. The reduction in the number of variants has made binary distribution of open source software practical. The many open source OS distributions largely compete on the basis of ease of installation and variety of binary packages available.

As a consequence, the proportion of people who build from source is much smaller than it once was. Despite this substantial change in circumstances, Gnu has popularized the use of configure as a means of setting compile time parameters.

Rather than simplifying building software, it has made the task staggeringly more difficult. It is no longer sufficient to know how to compile and link on your own system and where the various dependencies are located. Now all but the most trivial failure casts the prospective user into the computed goto from hell.

The FORTRAN feature which was the subject of Dijkstra's famous essay pales in complexity relative to even make which computes its actions based on time stamps which in a networked environment running NFS are all too often out of sync. In combination with autoconf, automake and libtool, this raises the level of complexity to truly dizzying heights. With files being executed after extensive processing with m4 or similar packages, it can become impossible to find where a change to the configuration should be made as tehre may be several layers of alteration between the initial specification and the actual evaluation.

One time software builds do not even need anything as complex as make. A simple shell script is quite sufficient. There is some small case to be made for using make in the event that the user is able and willing to modify the source code. However, that is a minor convenience. Anyone able to do that can quickly create a set of Makefiles from a script if needed. It seems quite unlikely that very many users are likely to make changes to packages that run into the millions of lines. The familiarization effort required is just too large.

If the vast body of available software cannot be compiled in a new environment, then computing is condemned to stagnation. No one can afford to reimplement all the things now readily available on Linux and BSD. However much work is expended on Linux or BSD, the model is 40 years old and was designed for a system with less computing resources than a typical cell phone. Anything that impedes the use of existing software in a new environment is harmful to the future of computing. Language standards arose to address this problem. Sadly, this is all too often ignored by programmers eager to use the very latest languages and features.

Conceptually, the configure and make model makes many assumptions about how software is transformed from source code to execuable objects. As a hypothetical instance, imagine a system that examines the contents of certain directories and automatically compiles and links any files it

finds for which it has the appropriate software to interpret the source files. Dependencies on libraries might be resolved by searching object symbol tables and in the event of conflicts consulting a precedence preference table or prompting the user.

A software package which applies elaborate, system specific transformations to the input before invoking a compiler will be very difficult to port to such a system. All the transformation rules will almost certainly be embedded in executables that will not run in the new environment and which would produce an erroneous result if they did.

The point here is not to introduce a new model for compiling and linking programs, but rather to highlight that a programmer who has never seen or heard of a system cannot divine how to build the software correctly. The human being who uses the system is far better able to do this. If an individual is not able to do that, they have no business attempting to install source level software until they do.

Because autoconf tries to resolve all the dependencies before any compilation takes place, it cannot possibly properly handle large complex systems with many external dependencies. As an example, consider 3 packages which each have options to interoperate with each other. When attempting to build the first, A, B and C are not available and configure will refuse to enable any library level options that reference those packages. When compiling B, it will now be possible to reference A, but C will still be inaccessible. In general, assuming that the programmers have not gotten too clever, it will require 2N-1 compilations to make all the features work. Current major scientific packages commonly have close to a dozen direct dependencies. Since many of those have dependencies of their own, the combinatorial complexity grows very rapidly.

There are however, more problems with the model posited by configure. It assumes that the dependencies are all static and that their features are known to the person creating the configure script. For example, if the person configuring a dependency on libsnafu is either unaware of or does not have it built using libfubar, when someone who has libsnafu built using libfubar tries to build the package, it will be difficult at best and may likely not be possible at all.

The current problem presented by configure is much the same as goto. It makes following the thread of execution difficult or impossible. The solution is also the same. It is simplicity. While David Parnas is famous for advocating information hiding, he was in fact arguing for making the important information easily discoverable by reducing the clutter of irrelevant detail.

When building a software package in a new environment, the builder is seldom, if ever, interested in details such as which objects get inserted into which libraries. Typical concerns evolve around compiler and linker options, dependencies and installation locations. The proper application of the principles espoused by Parnas require that details which are system specific be isolated into a single file and that this be completely separate from the internal construction details of the software package.

The computing environment of today is much simpler. We do not have dozens of system vendors using a dozen or more hardware variants. So most of the benefit offered by autoconf can be obtained by using a small number of include files which get selected based on the results returned by uname(1). Should a more detailed result be required, it’s simple to incorporate logic to vary Makefile symbols based on several possible compiler versions or similar. By isolating the setting of the variables to a single file, it becomes much easier for a human being to get things working quickly and easily. With some modest adherence to a standard naming convention, the requisite information can become part of the standard system environment and not even need to be supplied by the software package.